

---

# Enterprise Web Development

*Yakov Fain, Victor Rasputnis, Anatole Tartakovsky,  
and Viktor Gamov*

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo



## **Enterprise Web Development**

by Yakov Fain, Victor Rasputnis, Anatole Tartakovsky, and Viktor Gamov

Copyright © 2010 Yakov Fain, Victor Rasputnis, Anatole Tartakovsky, and Viktor Gamov. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editors:** Mary Treseler and Brian Anderson

**Indexer:**

**Production Editor:** Melanie Yarbrough

**Cover Designer:** Karen Montgomery

**Copyeditor:**

**Interior Designer:** David Futato

**Proofreader:** FIX ME!

**Illustrator:** Rebecca Demarest

July 2014: First Edition

### **Revision History for the First Edition:**

YYYY-MM-DD: First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449356811> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. !!FILL THIS IN!! and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-35681-1

[?]

---

# Table of Contents

Preface.....	xi
Introduction.....	xxiii

---

## Part I. Building Your Application

1. Mocking Up Save The Child Application.....	3
Mobile First?	4
Introducing Balsamiq Mockups	6
The Project Owner Talks to a Web Designer	7
First Mockups	7
From Mockups to a Prototype	11
Single Page Applications	11
Running Code Examples from WebStorm	12
Our First Prototype	13
Our Main Page JavaScript	18
The Footer section	22
The Donate Section	24
Adding Video	29
Adding the HTML5 Video Element	29
Embedding YouTube Videos	32
Adding Geolocation Support	34
Geolocation Basics	36
Integrating with Google Maps	39
Browser Features Detection With Modernizr	42
Search and Multi-Markers With Google Maps	47
Summary	51
2. Using AJAX and JSON.....	53
What's AJAX	53

---

What's JSON	54
How AJAX Works	55
Retrieving Data From the Server	56
AJAX: Good and Bad	59
Populating States and Countries from HTML Files	60
Using JSON	62
Populating States and Countries from JSON Files	64
Arrays in JSON	66
Loading Charity Events using AJAX and JSON	66
JSON in CMS	69
JSON in Java	71
Compressing JSON	72
Adding Charts to Save The Child	73
Adding Chart With Canvas Element	74
Adding Chart With SVG	78
Loading Data From Other Servers With JSONP	82
Beer and JSONP	83
Summary	86
<b>3. Introducing jQuery Library.....</b>	<b>87</b>
Getting Started With jQuery	88
Hello World	90
Selectors and Filters	91
Testing jQuery Code with JSFiddle	92
Filtering Elements	93
Handling Events	94
Attaching Events Handlers and Elements With the Method on()	95
Delegating Events	96
AJAX with jQuery	97
Handy Shorthand Methods	99
Save The Child With jQuery	100
Login and Donate	100
HTML States and Countries With jQuery AJAX	104
JSON States and Countries With jQuery AJAX	105
Submitting Donate Form	107
jQuery Plugins	113
Validating the Donate Form With Plugin	114
Adding Image Slider	116
Summary	118

---

## Part II. Enterprise Considerations

<b>4. Developing Web Applications in Ext JS Framework.</b>	<b>123</b>
JavaScript Frameworks	123
What This Chapter Covers	124
Why Ext JS?	124
Downloading and Installing Ext JS	125
Getting Familiar with Ext JS and Tooling	127
The First Version of Hello World	127
Generating Applications With Sencha CMD Tool	129
Which Ext JS Distribution to Use?	134
Declaring, Loading and Instantiating Classes	134
MVC in Ext JS	139
A Component's Lifecycle	146
Events	147
Layouts	148
Developing Save The Child With Ext JS	150
Setting Up Eclipse IDE and Apache Tomcat	150
Save The Child: The Top Portion of UI	156
Completing Save The Child	171
Summary	186
<b>5. Selected Productivity Tools for Enterprise Developers.</b>	<b>187</b>
Node.js, V8, and NPM	187
Automate Everything With Grunt	188
The Simplest Grunt File	188
Using Grunt to run JSHint Checks	189
Watching For the File Changes	191
Bower	192
Yeoman	196
Productive Enterprise Web Development with Ext JS and CDB	200
Ext JS MVC Application Scaffolding	200
Generating a CRUD application	205
Data Pagination	213
Summary	218
<b>6. Modularizing Large-Scale JavaScript Projects.</b>	<b>219</b>
Modularization basics	221
Roads To Modularization	223
The Module Pattern	223
CommonJS	226
Asynchronous Module Definition	228
Universal Module Definition	231
ECMAScript 6 Modules	232

Dicing the Save The Child Application Into Modules	236
Inside RequireJS configuration: config.js	240
Writing AMD Modules	241
Loading Modules On-Demand	242
RequireJS plugins	245
Using RequireJS Optimizer	245
Loosely-Coupled Inter-Module Communications With Mediator	249
Summary	255
<b>7. Test-Driven Development with JavaScript.....</b>	<b>257</b>
Why Test ?	258
Testing Basics	258
Unit Testing	259
Integration Testing	259
Functional Testing	259
Load Testing	260
Test Driven Development	263
Test-Driven Development With QUnit	264
Behavior-Driven Development With Jasmine	269
Multi-Browser Testing	280
Testing DOM	285
Save The Child With TDD	287
The Test-Driven ExtJS version of <i>SaveSickChild.org</i>	287
Setting up the IDE for TDD	294
Summary	299
References	299
<b>8. Upgrading HTTP To WebSocket.....</b>	<b>301</b>
Near Real Time Applications With HTTP	302
Polling	302
Long Polling	303
HTTP Streaming	304
Server-Sent Events	305
Introducing WebSocket API	306
WebSocket Interface	307
WebSocket Frameworks	314
The Portal	314
Atmosphere	315
Application-level Messages Format Considerations	316
CSV	317
XML	317
JSON	318

Google Protocol Buffers	318
WebSockets and Proxies	320
Adding an Auction to Save The Child	321
Monitor the WebSocket traffic with Chrome Developers Tools	329
Sniffing WebSocket frames with Wireshark	333
Save The Child Auction Protocol	338
Summary	341
References	341
<b>9. Introduction to Web Application Security.....</b>	<b>343</b>
HTTP vs HTTPS	344
Authentication and Passwords	345
Basic and Digest Authentication	346
Single Sign-on	347
Dealing With Passwords	348
Authorization	349
OAuth-Based Authentication and Authorization	350
Federated Identity with OpenID Connect and JSON Web Tokens	351
OAuth 2.0 Main Actors	353
Save The Child and OAuth	353
Top Security Risks	355
Injection	356
Cross-Site Scripting	357
Regulatory Compliance and Enterprise Security	359
Summary	361

---

## Part III. Responsive Web Design and Mobile

<b>10. Responsive Design: One Site Fits All.....</b>	<b>367</b>
One or Two Versions of Code?	367
How Many User Agents Are There	373
Back to Mockups	377
CSS Media Queries	381
How Many Breakpoints?	391
Fluid Grids	391
Moving Away From Absolute Sizing	392
Window as a Grid	392
Responsive CSS: The Good News	403
Making Save The Child Responsive	404
Fluid Media	415

Summary	417
<b>11. jQuery Mobile.....</b>	<b>419</b>
Where to get jQuery Mobile	419
Organizing the Code	420
How Will It Look on Mobile Devices?	423
Styling in jQuery Mobile	425
Adding Page Navigation	426
Persistent Toolbars	431
Save The Child with jQuery Mobile	437
Prototyping Mobile Version	438
The Project Structure and Navigation	453
Selected Code Fragments	461
Summary	480
<b>12. Sencha Touch.....</b>	<b>481</b>
Sencha Touch Overview	482
Code Generation and Distribution	482
Constructing UI	490
Save The Child With Sencha Touch	499
Building the Application	499
The Application Object	501
The Main View	504
Controller	510
The Other Views	513
Stores and Models	532
Dealing With Landscape Mode	534
Comparing jQuery Mobile and Sencha Touch	535
<b>13. Hybrid Mobile Applications.....</b>	<b>537</b>
Native Applications	537
Native vs. Web Applications	538
Hybrid Applications	539
Cordova and PhoneGap	539
Titanium	541
The Bottom Line	542
Intro to the PhoneGap Workflows	543
One More Hello World	544
Testing Applications on iOS Devices	549
Installing More Local SDKs	550
Using Adobe PhoneGap Build Service	551
Distribution of Mobile Applications	556

Save The Child with PhoneGap	559
How to Package Any HTML5 App With PhoneGap	559
Adding Camera Access to Save The Child	560
The Sever-Side Support for Photo Images	565
Summary	567
<b>14. Epilogue.....</b>	<b>569</b>
HTML5 is not a Rosy Place	569
Dart: A Promising Language	571
HTML5 is in Demand Today	572
<b>A. Appendix A. Advanced Introduction to JavaScript.....</b>	<b>573</b>
<b>B. Appendix B. Selected HTML5 APIs.....</b>	<b>637</b>
<b>C. Appendix C. Running Code Samples and IDE.....</b>	<b>679</b>
<b>Index.....</b>	<b>683</b>



---

# Preface

This book should help Web application developers and software architects in picking the right strategy for developing cross-platform applications that run on a variety of desktop computers as well as mobile devices. Primary audience is developers who are in a large organization and need to learn how to develop Web application using HTML5 stack.

## What's Enterprise Application?

This book has the word *Enterprise* in its title, and we'll explain you what we consider *enterprise applications* by giving you some examples. Creating a Web application that will process orders is not the same as creating a Web site to publish blogs. Enterprise applications include company-specific workflows, and usually they need to be integrated with number of internal systems, data sources and processes.

Google Doc is not an enterprise Web application. But Google appliance integrating search operating on company documents, databases, processes, tickets, and providing collaboration is - it integrates consumer-workforce front office with what the company does (back office).

Google Maps is not an enterprise application. But Google Maps integrated with the company site used by insurance agents to plan daily route, scheduling, doing address verification and geo-coding is.

Just using a Web application in some business doesn't make it an enterprise Web application. If you take Gmail as is, it won't be an enterprise application until you integrate it into another process of your business.

Is an online game an enterprise application? It depends on the game. A multi-player online roulette game hooked up to a payment system, and maintaining users' accounts is an enterprise Web application. But playing Sudoku online doesn't feel too enterprisey.

How about a dating Web site? If the site just offers an ability to display singles - it's just a publishing site as there is not much of a business there. Can you turn a dating Web site into an enterprise application? It's possible.

Some people will argue that an enterprise application must support multiple users, high data load, include data grids and dashboards, be scalable, have business and persistence layers, offer professional support et al. This is correct, but we don't believe that a Web application should do all this to qualify for the adjective *enterprise*.

Let's create a simple definition of an enterprise Web application:

*"An enterprise Web application is the one that helps an organization running its business online".*

## Why the Authors Wrote the book

The authors of this book have ninety years of combined experience in development of enterprise applications. During all these years we've been facing the same challenges regardless of what programming language we use:

- How to make the application code base maintainable
- How to make the application responsive by modularizing its code base
- How to minimize the number of production issues by proper testing on earlier stages of the project life cycle
- How to design UI that looks good and is convenient for users
- Which frameworks or libraries to pick
- Which design patterns to apply in coding

This list can be easily extended. Ten years ago we've been developing UI mainly in Java, five years ago in Adobe Flex, today in HTML5-related technologies. This book is an attempt to share with the readers our understanding of how to approach the above challenges in HTML5.

## Who This Book is For

Web application development with HTML5 includes HTML, JavaScript, CSS and dozens of JavaScript frameworks. The main goal of this book is to give you a hands-on overview of development Web applications that can be run on a variety of devices - desktops, tablets and smartphones. We expect the reader to have some experience with any programming language. Knowledge of basic HTML is also required. Understanding of the principles of object-oriented programming would be helpful too.

This book is intended for software developers, team leaders, and Web application architects who want to learn the following:

- How to write Web applications using some of the popular libraries and frameworks
- How to modularize the client's side of Web applications written in JavaScript
- How to test Web applications
- Is applying *responsive design* principles the right strategy for your application
- Which security vulnerabilities to watch for
- Why developing for mobile devices differs from developing for desktops
- What are the pros and cons of developing mobile application using HTML5 stack versus native languages



If you're new to programming in JavaScript language, start reading this book from Appendix A, which is an introduction to JavaScript.

## What the Book is And Why it's Important

This book has a lot of breadth, but for mastering some of the topics in depth be prepared to do additional studying. On the other hand, we give you a lot of working code samples to those who prefer studying by reading code.

This book can be important for busy professionals who don't have time for reading a separate book about each and every library and framework that exist in the HTML5 universe. This book will help them to narrow down the list of technologies and frameworks to be considered for the next project.

Enterprise server-side developers will also benefit from reading this book. Pretty often enterprise Java or .Net developers feel caught off guard when they get a task to create a new Web application with the cross-platform and cross-browser UI. These strong enterprise developers with good business knowledge may not have enough exposure to how things work in HTML5 domain. This book can be a time saver for all server-side developers who need to start working on the front end of Web applications.

Finally, this book is important because of the way it's written - you'll be working on the application that's introduced next.

# Introducing Save The Child Application

To make this book more practical, we decided to not just give you the unrelated code snippets illustration various syntax or techniques, but bring all of them together in a working application (just the UI portion). While learning various frameworks, libraries and approaches to building UI for Web application you'll be writing multiple versions of the same Web application - Save The Child. It's a sample charity application that may be used to collect donations for children who need medical attention.

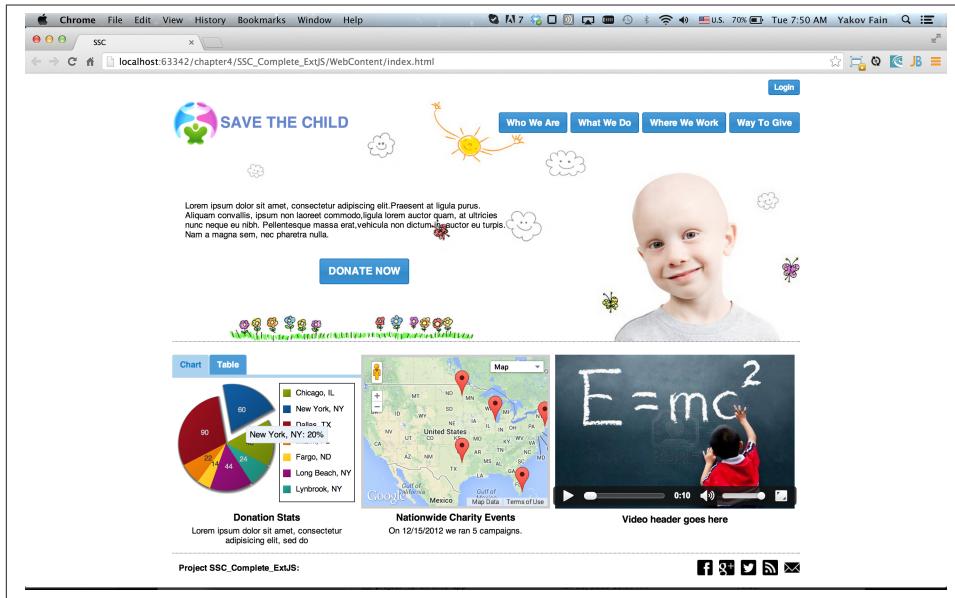


Figure P-1. Save The Child - a sample application

This Web application that will allow people to register, donate, find local kids that need help, match donors and recipients, upload images, videos and display statistics.

## Is This Even an Enterprise App?

By looking at the image above the reader may be thinking, "This doesn't look like an enterprise application". Let's see. Do you believe that an enterprise application has to consist of boring grey windows with lots of grids, forms, and some charts? True, but we got all of these elements in our application too:

- clicking on the Donate button will reveal a form that has to be filled out and sent to a payment processing system.

- The interactive live pie chart is something that many modern enterprise dashboards include.
- Clicking on the Table tab (right next to the Chart tab) shows the same donations stats in a grid (that one is greyish).
- Integration with the mapping API allows visually present the locations that run events important for this business or a non-profit organization.
- Under the hood this pretty window will use a high speed full duplex communication protocol WebSocket.

As a matter of fact, the company that employs authors of this book has a customer that is a non-for-profit organization that is in business of helping people fighting a certain disease. That application has two parts - consumer-facing and back-office. The former looks more colorful, while the latter has more grey grids indeed. Both parts process the same data and this organization can't operate if you remove any of these parts.

Would these feature make Save the Child an enterprise Web application? Yes, as it can help our imaginary non-for-profit organization to run its business: collecting donations for sick kids. Would you rather see a fully function Wall Street trading system? Maybe. But this book and our sample application incorporates all software components that you'd need to use for developing a financial application.

## How Are We Going to Build This App

Instead of giving a number of unrelated code samples, we decided to develop multiple versions of the same Web application built with different libraries, frameworks, and techniques. This approach will allow the reader compare apples to apples and make an educated decision which approach fits his or her needs the best.

First we'll show how to build this application in pure HTML/JavaScript, then we'll rewrite it using jQuery library, then with Ext JS framework. The users will be able to see where different charity events are being run (Google maps integration). The page will integrate the video player and display the chart with stats on donors by geographical location. One of the versions of this app shows how to modularize this application - this is a must for any enterprise system. Another version shows how to use WebSockets technology to illustrate the server-side data push while adding an auction to this Web application. The final chapters of the book show different ways of building different version of the same Save The Child application to run on mobile devices (Responsive Design, jQuery Mobile, Sencha Touch, and PhoneGap). We believe that this application will help you in comparing all these approaches and selecting those that fit your objectives.

# What the Goals of the Book Are

First, we want to say what's not the goal of this book. We are not planning to convince you that developing a cross-platform Web application is the right strategy for you. Don't be surprised if after reading this book you'll decide that developing applications in HTML5 is not the right approach for the tasks you have at hands.

This book should help decision makers in picking the right strategy for developing cross-platform applications that run on a variety of desktop computers as well as mobile devices.

## Technologies Used in This Book

This is HTML5 book, and the main programming language used here is JavaScript. We use HTML and CSS too. Most of the modern JavaScript development is done using various libraries and frameworks. The difference between a library and a framework is that the former does not dictate how to structure the code of your application - they simply offer a set of components that will spare you from writing lots of manual code. The goal of some frameworks is to help developers with testing of their applications. The goal of some frameworks is just to split the application into separate modules. There are tools just for building, packaging and running JavaScript applications. While many of the frameworks and tools will be mentioned in this book, the main technologies/libraries/tools/techniques/protocols used in this book are listed below:

- Balsamiq Mockups
- Modernizr
- jQuery
- jQuery Mobile
- Ext JS
- Sencha Touch
- RequireJS
- Jasmine
- Clear Data Builder
- WebSocket
- PhoneGap
- Grunt
- Bower
- WebStorm IDE

- Eclipse IDE

Although you can write your programs in any text editor, using specialized Integrated Development Environments is more productive, and we'll use Aptana Studio IDE by Appcelerator and WebStorm IDE by JetBrains.

## How the Book is Organized

Even though you may decide not to read some of the chapters we still recommend you to skim through them. If you're not familiar with JavaScript - start from Appendix A.

Chapters 1 and 2 are must read - if you can't read JavaScript code or are not familiar with CSS, AJAX or JSON, the rest of the book will be difficult to understand. On the other hand, if you're not planning to use, say Ext JS framework, you can just skim through Chapter 4. Following is a brief book outline:

**Introduction** includes a brief discussion of what's the difference between enterprise Web applications and Web sites. It also touches upon the evolution of HTML.

**Chapter 1** describes the process of mocking up the application Save The Child, which will support donations to the children, embed a video player, integrate with Google maps, and eventually will feature an online auction. We'll show you how to gradually build all the functionality of this Web application while explaining each step of the way. By the end of this chapter we'll have the Web design and the first prototype of the Save The Child application written using just HTML, JavaScript and CSS.

**Chapter 2** is about bringing external data to Web browsers by making asynchronous calls to server. The code from previous chapters uses only hard-coded data. Now it's time to learn how to make asynchronous server calls using AJAX techniques and consume the data in JSON format. The Save The Child application will start requesting the data from the external sources and sending them the JSON-formatted data.

**Chapter 3** shows how to use a popular jQuery library to lower the amount of manual coding in the Save The Child application. First, we'll introduce the jQuery Core library, and then re-build our Save The Child application with it. In the real world, developers often increase their productivity by using JavaScript libraries and frameworks.

**Chapter 4** is a mini tutorial of a comprehensive JavaScript framework called Ext JS. This is one of the most feature-complete frameworks available on the market. Sencha, the company behind Ext JS, managed to extend JavaScript to make its syntax closer to classical object-oriented languages. They also developed and extensive library of the UI components. Expect to see another re-write of the Save The child here.

**Chapter 5** is a review of productivity tools used by enterprise developers (NPM, Grunt, Bower, Yeoman, CDB). It's about build tools, code generators, and managing depen-

dencies (a typical enterprise application uses various software that need to work in harmony).

**Chapter 6** explains how to modularize large applications. Reducing the application startup latency and implementing lazy loading of certain parts of the application are the main reasons for modularization. We'll give you an example of how to build modularized Web applications that won't bring the large and monolithic code to the client's machine, but will rather load the code on as needed basis. You'll also see how to organize the data exchange between different programming modules in a loosely coupled fashion. The Save The Child application will be re-written with RequireJS framework, which will be loading modules on demand rather than the entire application.

**Chapter 7** is dedicated to test-driven development with JavaScript. To shorten the development cycle of your Web application you need to start testing it on the early stages of the project. It seems obvious, but many enterprise IT organizations haven't adopted agile testing methodologies, which costs them dearly. JavaScript is dynamically typed interpreted language - there is no compiler to help in identifying errors as it's done in compiled languages like Java. This means that a lot more time should be allocated for testing for JavaScript Web applications. We'll cover the basics of testing and will introduce to some of the popular testing frameworks for JavaScript application. Finally, you'll see how to test Save The Child application with Jasmine framework.

**Chapter 8** shows how to substantially speedup the interaction between the client and the server using WebSocket protocol introduced in HTML5. HTTP adds a lot of overhead for every request and response object that serve as wrappers for the data. You'll see how to introduce the WebSocket-based online auction to the new version of our Save The Child application. This is what Ian Hickson, the HTML5 spec editor from Google, said about why WebSocket protocol is important:

"Reducing kilobytes of data to 2 bytes is more than a little more byte efficient, and reducing latency from 150ms (TCP round trip to set up the connection plus a packet for the message) to 50ms (just the packet for the message) is far more than marginal. In fact, these two factors alone are enough to make WebSocket seriously interesting to Google."

**Chapter 9** is a brief introduction to Web application security. You'll learn about vulnerabilities of Web applications and will get references to the documents that contain recommendations on how to protect your application from attackers. This chapter concludes with some of the application-specific security considerations like the regulatory compliance that your business customers can't ignore.

**Chapter 10** opens up a discussion of how to approach creating Web applications that should run not only on desktops, but also on mobile devices. In this chapter you'll get familiar with the principles of responsive design, which allows to have a single code base that will be flexible enough to render the UI that looks good on the large and small

screens. You'll see the power of CSS *media queries* that will automatically re-allocate the UI components based on the screen width. The new version of the Save The Child application will demonstrate how to go about responsive design.

**Chapter 11** will introduce you to jQuery Mobile - the library that was specifically created for developing mobile Web applications. But main principles implemented in the larger jQuery library remain in place, and studying the materials from Chapter 3 is a prerequisite for the understanding of this chapter. Then you'll be creating the mobile version of the Save The Child with jQuery Mobile.

**Chapter 12** is about a little brother of Ext JS - Sencha Touch framework. This framework was developed for the mobile devices, and you'll need to read Chapter 6 to be able to understand the materials from this one. As usual, we'll develop another version of the mobile version of the Save The Child with Sencha Touch.

**Chapter 13** shows how you can create hybrid mobile applications, which are written with HTML/JavaScript/CSS, but can use the native API of the mobile devices. Hybrids are packaged as native mobile applications and can be submitted to the popular online app stores or market places the same way as if they were written in the programming language native for the mobile platform in question. This chapter will illustrate how to access the camera of the mobile device using the PhoneGap framework.

**Appendix A** is an introduction to programming with JavaScript. In about 60 pages we've covered main aspects of this language. No matter what framework you choose, a working knowledge the JavaScript is required.

**Appendix B** is a brief overview of selected APIs from HTML5 specification. They are supported by all modern Web browsers. We find these APIs important and useful for many Web applications. The following API will be reviewed in this chapter:

- Web Messaging
- Web Workers
- Application Cache
- Local Storage
- Indexed Database
- History API

**Appendix C** is a brief discussion of the Integrated Development Environments that are being used for HTML5 development in general and in this book in particular.

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

**Constant width**

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**Constant width bold**

Shows commands or other text that should be typed literally by the user.

*Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.



This element signifies a general note.



This element indicates a warning or caution.

## The Source Code of the Book Examples

The source code of all versions of the Save The Child application will be available for download from O'Reilly at <http://shop.oreilly.com/product/0636920028314.do>. There is also a [GitHub repository](#) where the authors keep the source code of the book examples.

The authors of this book also maintain [the Web site](#), where various versions of the sample Save The Child application are deployed so you can see them in action.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does

not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Book Title* by Some Author (O'Reilly). Copyright 2012 Some Copyright Holder, 978-0-596-xxxx-x.”

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Safari® Books Online



*Safari Books Online* is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **product mixes** and pricing programs for **organizations**, **government agencies**, and **individuals**. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens **more**. For more information about Safari Books Online, please visit us **online**.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
800-998-9938 (in the United States or Canada)  
707-829-0515 (international or local)  
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <http://www.oreilly.com/catalog/<catalog page>>.

To comment or ask technical questions about this book, send email to [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

## Acknowledgments

You see four names on this book cover. But this book a product of more than four people. It's a product of our company - Farata Systems.

In particular, we'd like to thank Alex Maltsev, who was playing a role of Jerry-The-Designer starting from Chapter 3 and onward. Alex created all UI prototypes for the sample Web application "Save Sick Child" that is designed, re-designed, developed, and re-developed several times in this book. He also developed a number of code samples for the book and all CSS files.

Our big thanks to Anton Moiseev who developed the Ext JS and Sencha Touch versions of our sample application.

Our hats off to the creators of the [Asciidoc text format](#) - the drafts of this book were prepared in this format with the subsequent generation of PDF, EPUB, MOBI, and HTML documents.

In our sample application we've used two images from the [iStockPhoto](#) collection: the smiling boy by the user *jessicaphoto* and the logo by the user *khalus*. Thank you, guys!

Finally, our thanks to the O'Reilly editors for being so patient while we were trying to hit lots of moving and evolving targets that together represent the universe known as HTML5.

---

# Introduction

During the last decade the authors of this book worked on many enterprise Web applications using variety of programming languages and frameworks: HTML, JavaScript, Java, and Flex to name a few. Apache Flex and Java produce compiled code that runs in a well known and predictable virtual machine (JVM and Flash Player respectively).

This book is about developing software using what's known as HTML5 stack. But only the second chapter of this book will offer you an overview of the selected HTML5 tags and APIs. The first chapter is an advanced introduction to JavaScript. The rest of the chapters are about designing, re-designing, developing, and re-developing a sample Web site Save The Child. You'll be learning whatever is required for building this Web application on the go.

You'll be using dynamic HTML (DHTML), which is HTML5, JavaScript, and Cascading Style Sheets (CSS). We'll add to the mix the XMLHttpRequest object that lives in a Web browser and communicates with the server without the need to refresh the entire Web page (a.k.a. AJAX). JSON will be our data format of choice for data exchange between the Web browser and the server.

## From DHTML to HTML5

DHTML stands for Dynamic HTML. Back in 1999 Microsoft introduced XMLHttpRequest object to allow the Web version of their mail client Outlook to update the browser's window without the need to refresh the entire Web page, and several years later it was substituted with a more popular acronym AJAX. The market share of IE5 was about 90% at the time, and in enterprises it was literally the only approved browser.

Many years passed by and today's Internet ecosystems changed quite a bit. Web browsers are a lot smarter and performance of JavaScript improved substantially. The browsers support 6-8-12 simultaneous connections per domain (as opposed to 2 five years ago), which gave a performance boost to all AJAX applications. At least one third of all Web requests is being made from smart phones or tablets. Apple Inc. started their war against

all browser's plugins hence using embedded Java VM or Flash Player is not an option there. The growing need to support a huge variety of mobile devices gave another boost for HTML5 stack, which is supported by all devices.

But choosing HTML5 as the least common denominator that works in various devices and browsers means lowering requirements for your enterprise project - the UI may not be pixel-perfect on any particular device, but it'll be made somewhat simpler (comparing to developing for one specific VM, device or OS) and will have the ability to adapt to different screen sizes and densities. Instead of implementing specific to device features, the functional specification will include requirements to test under several Web browsers, in many different screen sizes and resolutions. HTML5 developers spend a lot more time in the debugger than people who develop for a known VM. You'll have to be ready to solve problems like a dropdown not showing any data in one browser while working fine in the others. Can you imagine a situation when the click event is not always generated while working in Java, Flex, or Silverlight? Get ready for such surprises while testing your HTML5 application.

You'll save some time because there is no need to compile JavaScript, but you'll spend more time testing the running application during development and QA phases. The final deliverable of an HTML5 project may have as low as half of the functionality comparing to the same project developed for a VM. But you'll gain a little better Web adaptability, easier implementation of full text search and the ability to create **mash-ups**. The integration with other technologies will also become easier with HTML/JavaScript. If all these advantages are important to your application choose HTML5.

JavaScript will enforce its language and tooling limitations on any serious and complex enterprise project. You can develop a number of fairly independent windows, but creating a well tested and reliable HTML5 takes time. This can be done significantly easier by the use of libraries or a framework.

In this book we'll be using some JavaScript frameworks - there are dozens of those on the market. Several of them promise to cover all the needs of your Web application. Overall, there are two main categories of frameworks:

- a) Those that allow you to take an existing HTML5 Web site and easily add new attributes to all or some page elements so they would start shining, blinking, or do some other fun stuff. Such frameworks don't promote component-based development. They may not include navigation components, grids, trees, which are pretty typical for any UI of the corporate tasks. JQuery is probably the best representative of this group- it's light (30Kb), extendable, and easy to learn.
- b) Another group of frameworks offers rich libraries of high-level components and allow you to extend them. But overall, such components are supposed to be used together becoming a platform for your Web UI. These components process some events, offer support of the Model-View-Controller paradigm (or offshoot of that), have a proprietary way of laying out elements on the Web page, organize navigation et al. Ext JS from Sencha belongs to this group.

Dividing all frameworks into only two category is an oversimplification, of course. Frameworks like Backbone, Angular and Ember have no “components” in terms of the UI sense and some don’t even quite dictate how you build your application (as in, they are not full-stack like Sencha). Some of the frameworks are less intrusive and some more. But our goal is not compare and contrast all HTML5 frameworks, but rather show you some selected ones.

We’ll use both JQuery and Ext JS and will show you how to develop Web applications with each of them. JQuery is good for improving an existing JavaScript site. JQuery can be used to program about 80% of a Web site functionality, but for the UI components you’ll need to use another framework, e.g. jQuery UI. You should use jQuery for the look and feel support, which is what it’s meant for. But you can’t use it for building your application component model. The component model of Ext JS becomes a fabric of the Web site, which includes an application piece rather than just being a set of Web pages. Typically it’s a serious view navigator or a wizard to implement a serious business process or a workflow. Besides, ExtJS comes with a library of the rich UI components.

JavaScript frameworks are hiding from software developers all incompatibilities and take care of the cases when a Web browser doesn’t support some HTML5, CSS3, or JavaScript features yet.

High-level UI components and a workflow support are needed for a typical enterprise application where the user needs to perform several steps to complete the business process. And these 20% of an application’s code will require 80% of the project time of complex development. So choosing a framework is not the most difficult task. The main problem with DHTML projects is not how to pick the best JavaScript framework for development, but finding the right software developers. Lack of qualified developers increases the importance of using specialized frameworks for code testing. The entire code base must be thoroughly tested over and over again. We’ll discuss this subject in the Chapter 5 dedicated to test-driven development.

A JavaScript developer has to remember all unfinished pieces of code. Many things that we take for granted with compiled languages simply don’t exist in JavaScript. For example, in Java or C# just by looking at the method signature you know what are the data types of the method’s parameters. In JavaScript you can only guess if the parameter names are self descriptive. Take the Google’s framework GWT that allows developers write code in Java with auto-generation of the JavaScript code. Writing code in one language with further conversion and deployment in another one is a controversial idea unless the source and generated languages are very similar. We’re not big fans of GWT, because after writing the code you’ll need to be able to debug it. This is when a Java developer meets a foreign language JavaScript. The ideology and psychology of programming in JavaScript and Java are different. A person who writes in Java/GWT has to know how to read and interpret deployed JavaScript code. On the other hand, using TypeScript or CoffeeScript to produce JavaScript code can be a time saver.

The Ext JS framework creators decided to extend JavaScript introducing their version of classes and more familiar syntax for object-oriented languages. Technically they are extending or replacing the constructs of the JavaScript itself extending the alphabet. Ext JS recommends creating objects using `ext.create` instead of the operator `new`. But Ext JS is still a JavaScript framework.

JQuery framework substantially simplifies working with browser's DOM elements and there are millions of small components that know how to do one thing well, for example, **sliding effects**. But it's still JavaScript and requires developers to understand the power of JavaScript functions, callbacks, and closures.

## Should we develop in HTML5 if its standard is not finalized yet?

The short answer is yes. If you are planning to develop mainly for the mobile market, it's well equipped with the latest Web browsers and if you'll run into issues there, they won't be caused by the lack of HTML5 support. In the market of the enterprise Web applications, the aging Internet Explorer 8 is still being widely used and they don't support some of the HTML5 specific features. But it's not a show stopper either. If you are using one of the JavaScript frameworks that offers cross-browser compatibility, most likely, they take care of IE8 issues.

Remember that, even if you rely on a framework that claims to offer cross-browser compatibility, you will still need to test your application in the browsers you intend to support to ensure it functions as intended. The chances are that you may need to be fixing the framework's code here and there. Maintaining compatibility is a huge challenge for any framework's vendor, which in some cases can consist of just one developer. Spend some time working with the framework, and then work on your application code. If you can, submit your fixes back to the framework's code base - most of them are open source.

If you are planning to write pure JavaScript, add the tiny framework Modernizr (see Chapter 1) to your code base, which will detect if a certain feature is supported by the user's Web browser, and if not - provide an alternative solution. We like the analogy with TV sets. People with latest 3D HD TV sets and those who have 50-year old black and white televisions can watch the same movie even though the quality of the picture will be drastically different.

## Challenges of the Enterprise Developer

If you are an enterprise developer starting working on your first HTML5 enterprise project, get ready to solve the same tasks as all UI software developers face regardless of what programming language they use:

- Reliability of the network communications. What if the data never arrive from/to the server? Is it possible to recover the lost data? Where they got lost? Can we resend the lost data? What to do with duplicates?
- Modularization of your application. If your application has certain rarely used menus don't even load the code that handles this menu.
- Perceived performance. How quickly the main window of your application is loaded to the user's computer? How heavy is the framework's code base?
- Should you store the application state on the server or on the client?
- Does the framework offer a rich library of components?
- Does the framework support creation of loosely coupled application components? Is the event model well designed?
- Does the framework of your choice cover most of the needs of your application, or you'll need to use several frameworks?
- Is well written documentation available?
- Does the framework of your choice locks you in? Does it restrict your choices? Can you easily replace this framework with another one if need be?
- Is there an active community to ask for help with technical questions?
- What is the right set of tools to increase your productivity (debugging, code generation, build automation, dependency management)?
- What are the security risks that need to be addressed to prevent expose sensitive information to malicious attackers?

We could continue adding items to this list. But our main message is that developing HTML5 applications is not just about adding tag video and canvas to a Web page. It's about serious JavaScript programming. In this book we'll discuss all of the listed above challenges.

## Summary

HTML5 is ready for the prime time. There is no need to wait for the official release of its final standard - all modern Web browsers support most of the HTML5 features and API's for a couple of years now. To be productive, you'll need to use not just HTML, JavaScript, and CSS, but a number of third-party libraries, frameworks and tools. In this book we'll introduce you to a number of them, which will help you to make the final choice of the right set of productivity tools that work for your project the best.



## PART I

---

# Building Your Application

This book has three parts. In Part 1 we'll start building Web applications. We'll be building and re-building a sample application titled Save The Child.

We assume that the readers know how to write programs in JavaScript. If you are not familiar with this language, study the materials from [Appendix A](#) first. You'll find a fast paced introduction to JavaScript there.

In Chapter 1 we'll start working with a Web designer. We'll create a mockup, and will start development in pure JavaScript. By the end of this chapter the first version of this application will be working using hard-coded data.

Chapter 2 will show you how to use AJAX techniques to allow Web pages communicate with external data sources without the need to refresh the page. We'll also cover JSON - a de facto standard data format when it comes to communication between the Web browsers and servers.

Chapter 3 shows how to minimize the amount of manually written JavaScript by introducing popular jQuery library. We'll rebuild the Save The Child application with jQuery.

After reading of this part you'll be ready to immerse into a more heavy-duty tools and frameworks that are being used by enterprise developers.



# Mocking Up Save The Child Application

Let's start working on our Web application Save The Child. It's going to be a Web application that will contain a form for donations to sick children and embeded video player, will integrate with Google maps, will have charts, and more. The goal is to gradually build all the functionality of this Web application while explaining each step of the way and giving you the reasons why we are building it the way we do. By the end of this chapter we'll have the Web design and the first prototype of the Save The Child.

The proliferation of mobile devices and Web applications require new skills for development of what was known as boring-looking enterprise applications. In the past, design of the user interface of most of the enterprise applications was done by developers to the best of their artistic abilities: a couple of buttons here, and a grid there on a gray background. The business users were happy cause they did not see any better. The application allowed to process business data – what else to wish for? Enterprise business users used to be happy with any UI as long as the application helped them to take care of their business.

But today's business users are spoiled by nice looking consumer-facing applications, and more often than not new development starts with inviting a Web designer who should create a prototype of the future application. For example, we've seen some excellent (from the UI perspective) functional specifications for boring financial applications made by professional designers. Business users slowly but surely becoming more demanding in the area of the UI design solutions. The trend is clear: developer's art does not cut it anymore.

In enterprise IT shops the Web design is usually done by a professional Web designer. The software developers are not overly familiar with the tools that Web designers are using. But to make this book useful even for a smaller shops that can't afford professional Web design, we'll illustrate the process of design and prototyping of the UI of a Web application.

Our Web designer, let's call him Jerry, is ready to start working on the *mockup* (a.k.a. *wireframes*) - a set of images depicting various views of the future Save The Child application. We expect him to deliver images with comments that would briefly explain what should change in a view if a user will take certain actions, e.g. clicks on the button. You can also think of a UI of an application as a set of states, and the user's action result in your application transitioning from one state to the other. As nerds and mathematicians say, the UI of your application is a **finite state machine**“, which at any given point of time is in one of the finite number of states, for example, in the view state Donate Form or Auction.

## Mobile First?

While starting working on the design of a new Web application keep in mind that most likely some users will access it from mobile devices. Will the proposed UI look good on the mobile devices with smaller screens? Some people suggest using so-called “Mobile First” approach, which means that from the very early stages of Web application development you should do the following:

- Ensure that your Web application (design and layout) looks on smaller screens
- Differentiate the content to be shown on large vs small screens (start with small screens and enhance for the larger ones)
- Test your application on the slow (3G-like) networks and minimize the “weight” of the landing page.
- Decide on using responsive design (see Chapter 12) vs HTML5 mobile framework vs native vs hybrid (see Chapter 15) approach.
- If you are planning to use geographical location services decide on the API to be used for mobile devices, but don't forget about the desktops too.



The users of iOS and Android devices are used to the fact that they can find the closest restaurant or a gas station based on their current location. Do you know that location feature can be available on the desktops too? Google maps API is just one of the services that can find the location of the user's desktop based on its IP address, Wi-Fi router's ID or proximity to the cell towers. Zeroing in on your device may not be as precise as with the smartphone's GPS, but it may be good enough. So why not plan for adding this feature to all versions of your Web application? Finding the closest charity event or a The Child can be done knowing an approximate location of your desktop computer.

Let's consider pointing devices. At the time of this writing, vast majority of the desktop users work with pixel-perfect mouse pointers or track pads. SmartPhone or tablet users work with fingers. One finger touch can cover a square with 100 pixels. The CNN site shows lots of news links located very close to each other on the screen. A finger may cover more than one link, and Android devices offer you a larger popup allowing you to select the link you really wanted to touch. Having the "Mobile first" state of mind doesn't mean that CNN would need to keep the larger distance between the links for all the users. But this means that they should foresee the issues or innovate using the features offered by modern mobile devices.

In Chapter 10 we'll discuss the *Responsive Design* techniques that allow to create the UI for Web applications that automatically re-allocate the screen content based on the screen size of the user's device. Although this chapter is about the desktop version of the Save The Child Web application, its screen will consist of several rectangular areas that can be allocated differently (or even hidden) on smartphones or tablets.



Before writing this book we've discussed how our application should look/work on mobile devices. But strictly speaking, since the work on multiple chapters was done in parallel, this was not a mobile first approach.



Consider reading Chapter 12 now to understand what you will need to deal with developing Web applications that look good on desktop monitors as well as on mobile screens. Understanding of the responsive design principles will help you in communications with your Web designer.

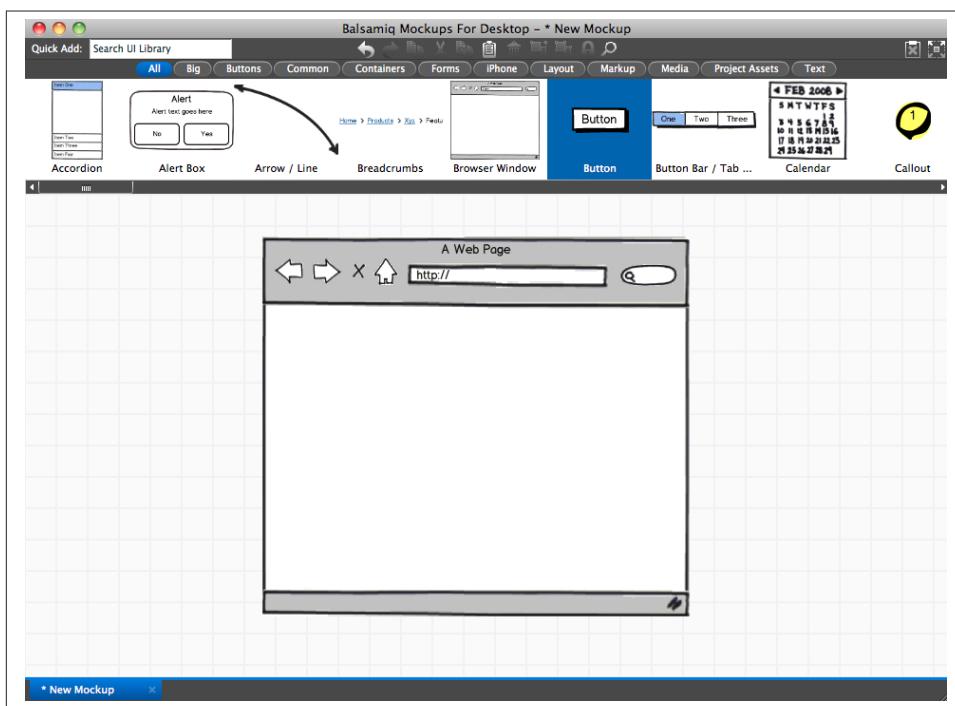
One of the constraints that the mobile users have is the relatively slow speed of the mobile Internet. This means that even though your desktop users will use fast LAN connection lines, your Web application has to be modularized and only the minimal number modules has to be loaded initially. Often mobile providers charge the user based on the amount of consumed data too.

The chances are slim that the desktop users will lose the Internet connection for a long period of time. On the other hand, the mobile users may stay in the area with no or spotty connection. In this case the mobile first thinking can lead to introducing an offline mode with limited functionality.

Thinking upfront of the minimal content to be displayed on a small mobile screens may force you to change the design of the desktop Web pages too. In our sample Save The Child application we need to make sure that there is a space for the Donate button even on the smallest devices.

# Introducing Balsamiq Mockups

Visualize a project owner talking to Jerry in a cafeteria, and Jerry is drawing sketches of the future Web site on a napkin. Well, in the 21st Century he'll use an electronic napkin so to speak - an excellent prototyping tool called **Balsamiq Mockups**. This easy to use program gives you a working area where you create a mockup of your future Web application by dragging and dropping the required UI components from the toolbar onto the image of the Web page (see [Figure 1-1](#)).



*Figure 1-1. The working area of Balsamiq Mockups*

If you can't find the required image in Balsamiq's library, add your own by dragging and dropping it onto the top toolbar. For example, the mockup in Chapter 10 uses our images of the iPhone that we've added to Balsamiq assets.



If you prefer using free tools, consider using [Mockflow](#).

When the prototype is done, it can be saved as an image and sent to the project owner. Another option is to export the Balsamiq project into XML, and if both the project owner and Web designer have Balsamiq installed, they can work on the prototype in collaboration. For example, the designer exports the current states of the project, the owner imports it and makes corrections or comments, then exports it again and sends it back to the designer.

## The Project Owner Talks to a Web Designer

During the first meeting Jerry talks to the project owner discussing the required functionality and then creates the UI to be implemented by Web developers. The artifacts produced by the designer vary depending on the qualifications of the designer. This can be a set of images representing different states of the UI with little *callouts* explaining the navigation of the application. If the Web designer is familiar with HTML and CSS, developers may get a working prototype in the form of HTML and CSS files, and this is exactly what Jerry will create by the end of this chapter.

Our project owner said to Jerry: “*The Save The Child Web application should allow people to make donations to The Children. The users should be able to find these children by specifying a geographical area on the map. The application should include a video player and display statistics about the donors and recipients. The application should include an online auction with proceeds going to charity. We’ll start working on the desktop version of this Web first, but your future mockup should include three versions of the UI supporting desktops, tablets, and smartphones*”.

After the meeting Jerry launched Balsamiq and started to work. He decided that the main window will consist of four areas laid out vertically:

1. The header with the logo and several navigation buttons
2. The main area with the Donate support plus the video player
3. The area with the Find Local Kid, statistics, and charts.
4. The footer with several house-holding links plus the icons for Twitter and Face-Book.

## First Mockups

The first deliverable of our Web designer (see [Figure 1-2](#) and [Figure 1-3](#)) depicted two states of the UI: before and after clicking the button *Donate Now*. The Web designer suggested that on the button click the Video Player would turn into a small button revealing the donation form.

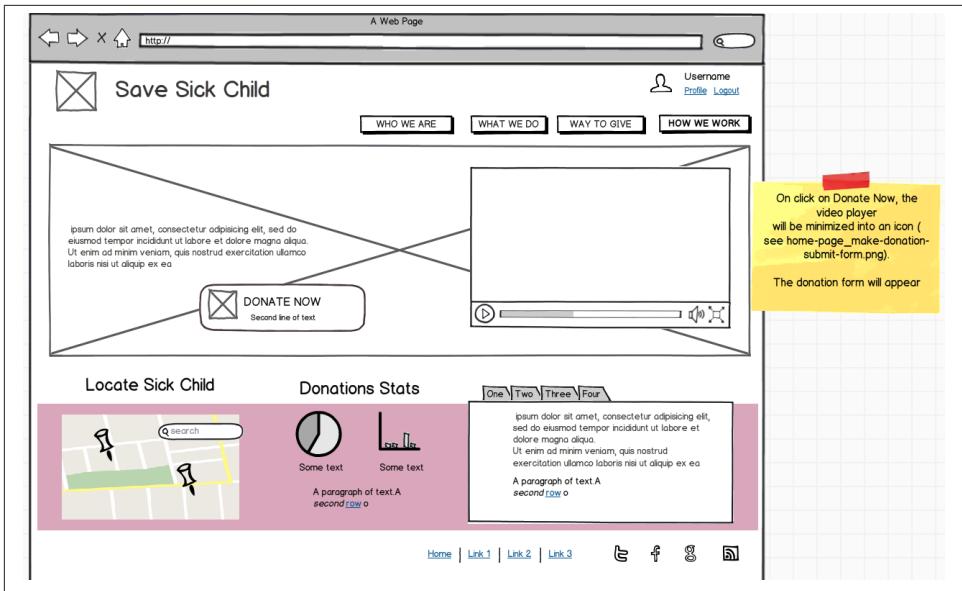


Figure 1-2. The main view before clicking *Donate Now*



*Figure 1-3. The main view after clicking Donate Now*

The project owner suggested that turning the video into a Donate button may not be a the best idea. We shouldn't forget that the main goal of this application is collecting donation, so they decided to keep the user's attention on the Donate area and move the video player to the lower portion of the window.

After that they went over the mockups of the authorization routine. The view states in this process can be : 1. Not Logged On 2. The Login Form 3. Wrong ID/Password 4. Forgot Password 5. Successfully Logged On

The Web designer has created mockups of some of these states as shown on [Figure 1-4](#) and [Figure 1-5](#).

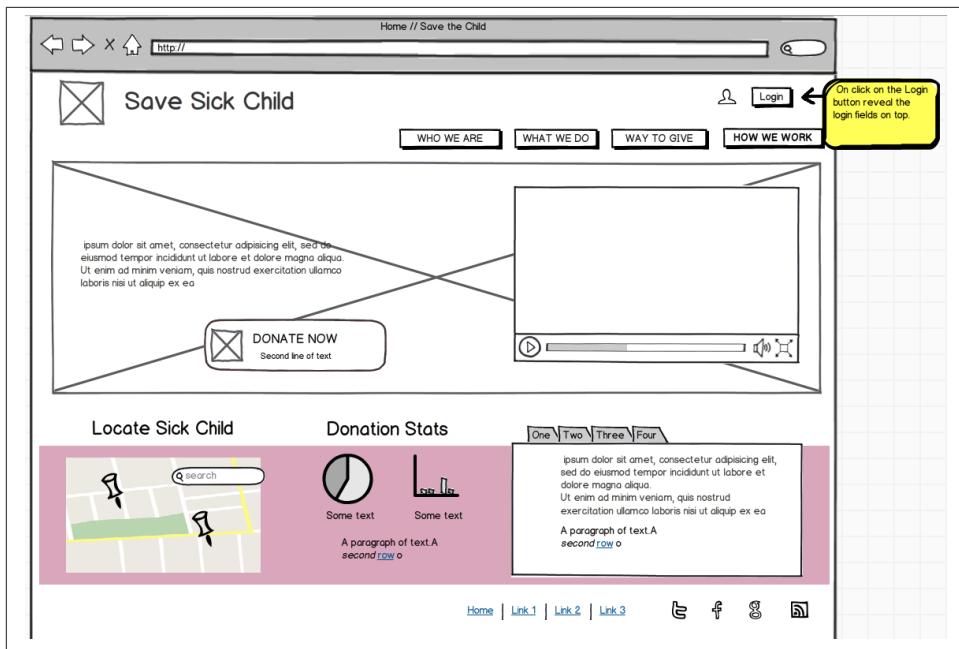
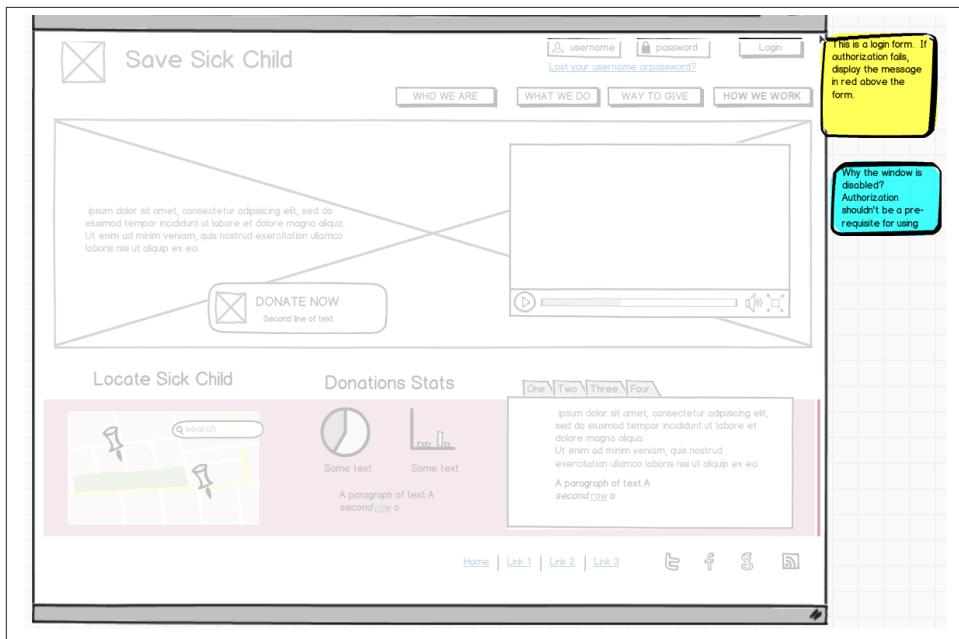


Figure 1-4. The user haven't clicked on the Login button

The latter shows different UI states should the user decide to log in. The project owner reviewed the mockups and return them back to Jerry with some comments. The project owner wanted to make sure that the user doesn't have to log on to the application to access the Web site. The process of making donations has to be as easy as possible, and forcing the donor to log on may scare some people away, so the project owner left his comment as shown on Figure 1-5.



*Figure 1-5. The user haven't clicked on the Login button*

This is enough of a design for us to build a working prototype of the app and start getting the feedback from business users. In the real world when a prospective client (including business users from your enterprise) approaches you asking for a project estimate, provide them with the document with a detailed work breakdown and the screenshots made by Balsamiq or a similar tool.

## From Mockups to a Prototype

We are lucky - Jerry knows HTML and CSS. He's ready to turn the still mockups into the first working prototype. It'll use only hard-coded data but the layout of the site will be done in CSS and we'll use HTML5 markup. We'll design this application as a Single-Page Application (SPA).

## Single Page Applications

A Single Page Web Application (SPA) is an architectural approach that doesn't require the user going through multiple pages to navigate the site. The user enters the URL in the browser, which brings the Web page that remains open on the screen until the user stop working with this application. The portion of the user's screen may change as the user navigate the application, the new data comes in using the AJAX techniques (see Chapter 2), or the new DOM elements will need to be created during the runtime, but

the main page itself doesn't gets reloaded. This allows building so-called fat client applications that can remember its state. Besides, most likely your HTML5 application will use some JavaScript framework, which in SPA gets loaded only once when the home page gets created by the browser.

Have you ever seen a monitor of a trader working for a Wall Street firm? They usually have three or four large monitors, but let's just look at one of them. Imagine a busy screen with lots and lots constantly changing data grouped in dedicated areas of the window. This screen shows the constantly changing prices from financial markets, the trader can place orders to buy or sell products, and notifications on completed trades are also coming to the same screen. If this would be a Web application it would live in the same Web page. No menus to open another windows.

The price of Apple share was \$590.45 just a second ago and now it's \$590.60. How can this be done technically? Here's one of the possibilities: every second an AJAX call is being made to the remote server providing current stock prices and the JavaScript code finds in the DOM the HTML element responsible for rendering the price and then modifies its value with the latest price.

Have you seen a Web page showing the content of the input box of Google's Gmail? It looks like a table with a list of rows representing the sender, subject, and the date of when each email arrived. All of a sudden you see a new row in bold on top of the list - the new email came in. How was this done technically? A new object(s) was created and inserted into a DOM tree. No page changes, no needs for the user to refresh the browser's page - an undercover AJAX call gets the data and JavaScript changes the DOM. The content of DOM changed - the user sees an updated value.

## Running Code Examples from WebStorm

The authors of this book use WebStorm IDE 7 from JetBrains for developing real-world projects. Appendix C explains how to run code samples in WebStorm.

This chapter will include lots of code samples illustrating how the UI is gradually being built. We've created a number small Web applications. Each of them can be run independently. Just download and open in WebStorm (or any other) IDE the directory containing samples from Chapter 1. After that you'll be able to run each of these examples by right-clicking on the index.html and selecting *Open in Browser* menu of WebStorm.



We assume that the users of our Save The Child application work with the modern versions of Web browsers (two year old or younger). The real world Web developers need to deal with finding workarounds to the unsupported CSS or HTML5 features in the old browsers, but modern IDE generate HTML5 boilerplate code that include large CSS files providing different solutions to older browsers.

JavaScript frameworks implement workarounds (a.k.a. polyfills) for features unsupported by old browsers too, so we don't want to clutter the text providing several versions of the code just to make book samples work in outdated browsers. This is especially important when developing Enterprise apps in situations where the majority of users are locked in a particular version of and older Web browser.

## Our First Prototype

In this section you'll see several projects that show how the static mockup will turn into a working prototype with the help of HTML, CSS, and JavaScript. Jerry, the designer, decided to have four separate areas on the page hence he created the HTML file index.html that has the tag `<header>` with the navigation tag `<nav>`, two `<div>` tags for the middle sections of the page and a `<footer>`:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Save The Child | Home Page</title>
    <link rel="stylesheet" href="css/styles.css">
  </head>
  <body>
    <div id="main-container">
      <header>
        <h1>Save The Child</h1>
        <nav>
          <ul>
            <li>
              <a href="javascript:void(0)">Who we are</a>
            </li>
            <li>
              <a href="javascript:void(0)">What we do</a>
            </li>
            <li>
              <a href="javascript:void(0)">Way to give</a>
            </li>
            <li>
              <a href="javascript:void(0)">How we work</a>
            </li>
          </ul>
        </nav>
    </div>
  </body>
</html>
```

```

</header>
<div id="main" role="main">
    <section>
        Donate section and Video Player go here
    </section>
    <section>
        Locate The Child, stats and tab folder go here
    </section>
</div>
<footer>
    <section id="temp-project-name-container">
        <b>project 01</b>: This is the page footer
    </section>
</footer>
</div>
</body>
</html>

```

Note that the above HTML file includes the CSS file shown below using the `<link>` tag. Since there is no content yet for the navigation links to open, we use the syntax `href="javascript:void(0)"` that allows to create a live link that doesn't load any page, which is fine on the prototyping stage.

```

/* Navigation menu */
nav {
    float: right
}
nav ul li {
    list-style: none;
    float: left;
}
nav ul li a {
    display: block;
    padding: 7px 12px;
}

/* Main content
   #main-container is a wrapper for all page content
*/
#main-container {
    width: 980px;
    margin: 0 auto;
}
div#main {
    clear: both;
}

/* Footer */
footer {
    /* Set background color just to make the footer standout*/
    background: #eee;
    height: 20px;
}

```

```
}

footer #temp-project-name-container {
    float: left;
}
```

The above CSS controls not only the styles of the page content, but also that sets the page layout. The `<nav>` section should be pushed to the right. If an unordered list is placed inside the `<nav>`, it should be left aligned. The width of the HTML container with ID `main-container` should be 980 pixels, and it has to be automatically centered. The footer will be 20 pixels high and should have a gray background. The first version of our Web page is shown on [Figure 1-6](#). Run `index.html` from the `project-01-get-started`.



In Chapter 10 you'll see how to create Web pages with more flexible layouts that don't require specifying absolute sizes in pixels.



*Figure 1-6. Working prototype. Take 1: Getting Started*

The next version of our prototype is more interesting, and it will contain a lot more code. First of all, the CSS file will become fancier, the layout of the four page sections will properly divide the screen real estate. We'll add a Logo and a nicely styled Login button to the top of the page. This version of the code will also introduce some JavaScript supporting user's authorization. Run the `project-02-login`, and you'll see a window similar to [Figure 1-7](#).

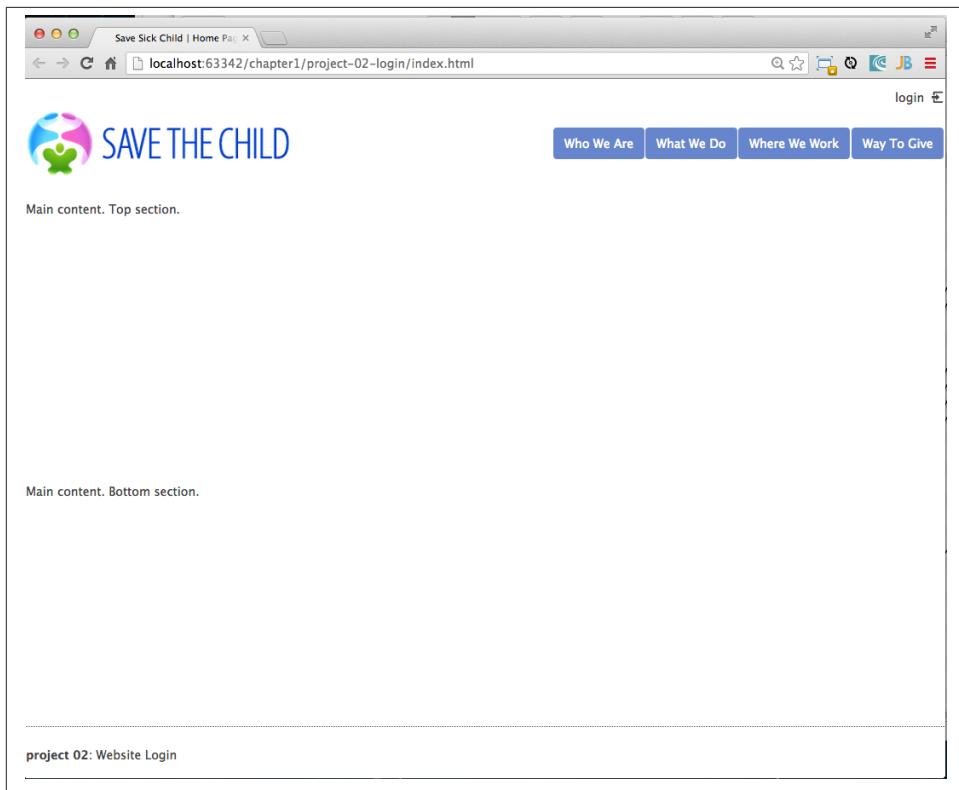


Figure 1-7. Working prototype. Take2: Login

This project has several directories to keep JavaScript, images, CSS, and fonts separately. We'll talk about special icon fonts later in this section, but first things first - let's take a close look at the HTML code.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title>Save The Child</title>
    <link rel="stylesheet" href="assets/css/styles.css">

  </head>
  <body>
    <div id="main-container">
      <header>

        <h1 id="logo"><a href="javascript:void(0)">Save The Child</a></h1>

        <nav id="top-nav">
```

```

<ul>
  <li id="login">
    <div id="authorized">
      <span class="icon-user authorized-icon"></span>
      <span id="user-authorized">admin</span>
      <br/>
      <a id="profile-link" href="javascript:void(0);">profile</a> |
      <a id="logout-link" href="javascript:void(0);">logout</a>
    </div>

    <form id="login-form">
      <span class="icon-user login-form-icons"></span>
      <input id="username" name="username" type="text"
        placeholder="username" autocomplete="off" />
      &nbsp; <span class="icon-locked login-form-icons"></span>
      <input id="password" name="password"
        type="password" placeholder="password"/>
    </form>
    <a id="login-submit" href="javascript:void(0)">login &nbsp;
      <span class="icon-enter"></span> </a>

    <div id="login-link" class="show-form">login
      &nbsp; <span class="icon-enter"></span></div>

      <div class="clearfix"></div>
    </li>
    <li id="top-menu-items">
      <ul>
        <li>
          <a href="javascript:void(0)">Who We Are</a>
        </li>
        <li>
          <a href="javascript:void(0)">What We Do</a>
        </li>
        <li>
          <a href="javascript:void(0)">Where We Work</a>
        </li>
        <li>
          <a href="javascript:void(0)">Way To Give</a>
        </li>
      </ul>
    </li>
  </ul>
</nav>
</header>

<div id="main" role="main">
  <section id="main-top-section">
    <br/>
    Main content. Top section.
  </section>
  <section id="main-bottom-section">

```

```

        Main content. Bottom section.
    </section>
</div>
<footer>
    <section id="temp-project-name-container">
        <b>This is the footer</b>
    </section>
</footer>
</div>
<script src="assets/js/main.js"></script>
</body>
</html>

```

Usually, the logos on multi-page Web sites are clickable - they bring up the home page. That's why Jerry placed the anchor tag in the logo section. But we are planning to build a single-page application so having a clickable logo won't be needed.

Run this project in WebStorm and click on the button Login, and you'll see that it reacts. But looking at the login-related `<a>` tags in the `<header>` section you'll find nothing but `href="javascript:void(0)"`. So why the button reacts? Read the code in the main.js shown below, and you'll find there line `loginLink.addEventListener('click', showLoginForm, false);` that invokes the callback `showLoginForm()`. That's why the Login button reacts. This seems confusing because the anchor component was used here just for styling purposes. In this example a better solution would be to replace the anchor tag `<a id="login-link" class="show-form" href="javascript:void(0)">` with another component that doesn't make the code confusing, for example `<div id="login-link" class="show-form">`.

We do not want to build Web applications the old way when a server-side program prepares and sends UI fragments to the client. The server and the client send to each other only the data. If the server is not available, we can use the local storage (the offline mode) or a mockup data one the client.

## Our Main Page JavaScript

Now let's examine the JavaScript code located in main.js. This code will self-invoke the anonymous function, which creates an object - encapsulated namespace ssc (short for Save Sick Child). This avoids polluting the global namespace. If we wanted to expose anything from this closure to the global namespace we could have done this via the variable `ssc` as described in [Appendix A](#) in section Closures.

```

// global namespace ssc
var ssc = (function() {
    // Encapsulated variables

    // Find login section elements
    // You can use here document.querySelector()
    // ❶

```

```

// instead of getElementById ()
var loginLink = document.getElementById("login-link");
var loginForm = document.getElementById("login-form");
var loginSubmit = document.getElementById('login-submit');
var logoutLink = document.getElementById('logout-link');
var profileLink = document.getElementById('profile-link');
var authorizedSection = document.getElementById("authorized");

var userName = document.getElementById('username');
var userPassword = document.getElementById('password');

// Register event listeners // ②
loginLink.addEventListener('click', showLoginForm, false);
loginSubmit.addEventListener('click', logIn, false);
logoutLink.addEventListener('click', logOut, false);
profileLink.addEventListener('click', getProfile, false);

function showLoginForm() {
    loginLink.style.display = "none"; // ③
    loginForm.style.display = "block";
    loginSubmit.style.display = "block";
}

function showAuthorizedSection() {
    authorizedSection.style.display = "block";
    loginForm.style.display = "none";
    loginSubmit.style.display = "none";
}

function logIn() {
    //check credentials
    var userNameValue = userName.value;
    var userNameValueLength = userName.value.length;
    var userPasswordValue = userPassword.value;
    var userPasswordLength = userPassword.value.length;

    if (userNameValueLength == 0 || userPasswordLength == 0) {
        if (userNameValueLength == 0) {
            console.log("username can't be empty");
        }
        if (userPasswordLength == 0) {
            console.log("password can't be empty");
        }
    } else if (userNameValue != 'admin' ||
               userPasswordValue != '1234') {
        console.log('username or password is invalid');
    } else if (userNameValue == 'admin' &&
               userPasswordValue == '1234') {
        showAuthorizedSection(); // ④
    }
}

```

```

        }
    }

    function logOut() {
        userName.value = '';
        userPassword.value = '';
        authorizedSection.style.display = "none";
        loginLink.style.display = "block";
    }

    function getProfile() {
        console.log('Profile link clicked');
    }

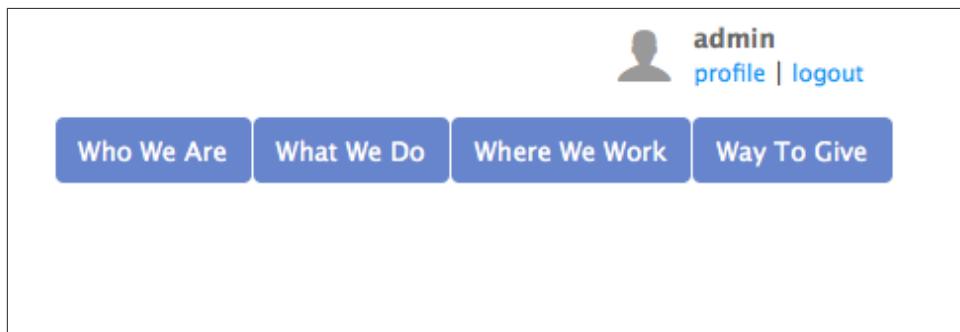
})();

```

- ➊ First query the DOM to get references to login-related HTML elements.
- ➋ Register event listeners for the clickable login elements.
- ➌ To make a DOM element invisible set its `style.display="none"`. Hide the login button and show the login form having two input fields for entering the user id and the password.
- ➍ If the user is `admin` and the password is `1234`, hide the `loginForm` and make the top corner of the page look as in [Figure 1-8](#).



We keep the user ID and password in this code just for the illustration purposes. Never do this in your applications. Authentication has to be done in a secure way on the server side.



*Figure 1-8. After successful login*

## Where to put JavaScript

We recommend placing the `<script>` tag with your JavaScript at the end of your HTML file as in our index.html above. If you move the line `<script src="js/main.js"></script>` to the top of the `<body>` section and re-run index.html the screen will look as in [Figure 1-7](#), but clicking on the Login won't display the login form as it should. Why? Because registering of the event listeners in the script main.js failed cause the DOM components (`login-link`, `login-form` and others) were not created yet by the time this script was running. Open the Firebug, Chrome Developer Tools, or any other debugging tool and you'll see an error on the console that will look similar to the following:

`"TypeError: loginLink is null loginLink.addEventListener(click, showLoginForm, false);"`

Of course, in many cases your JavaScript code could have tested if the DOM elements exist before using them, but in this particular sample it's just easier to put the script at the end of the HTML file. Another solution would be to load the JavaScript code located in main.js in a separate handler function that would run only when the window's `load` event is dispatched by the browser indicated that the DOM is ready: `window.addEventListener('load', function() {...})`. You'll see how to do this in the next version of main.js.

## The CSS of our Main Page

After reviewing the HTML and JavaScript code let's spend a little more time with the CSS that supports the pages shown in [Figure 1-7](#). The difference between the screen shots shown in [Figure 1-6](#) and [Figure 1-7](#) is substantial. First, the top left image is nowhere to be found in index.html. Open the styles.css file and you'll see the line `background: url(..../img/logo.png) no-repeat;` in the header `h1#logo` section.

The page layout is also specified in the file styles.css. In this version the sizes of each section is specified in pixels (px), which won't make your page fluid and easily resizable. For example, the HTML element with `id="main-top-section"` is styled like this:

```
#main-top-section {  
    width: 100%;  
    height: 320px;  
    margin-top: 18px;  
}
```

Jerry styled the main top section to take the entire width of the browser's window and to be 320 pixels tall. If you'll keep in mind the "Mobile First" mantra, this may not be the best approach cause 320 pixels mean difference size (in inches) on the displays with different screen density. For example, 320 pixels on the iPhone 5 with retina display will look a lot smaller than 320 pixels on the iPhone 4. You may consider switching from px to em units: 1em is equal to the current font height, 2em means twice the size et al. You can read more about creating scalable style sheets with `em` units at <http://www.w3.org/WAI/GL/css2em.htm>.

What looks a Login button on [Figure 1-7](#) is not a button, but a styled `div` element. Initially it was a clickable anchor `<a>`, and we've explained this change right after the listing shown `index.html` above. The CSS fragment supporting the Login button looks like this:

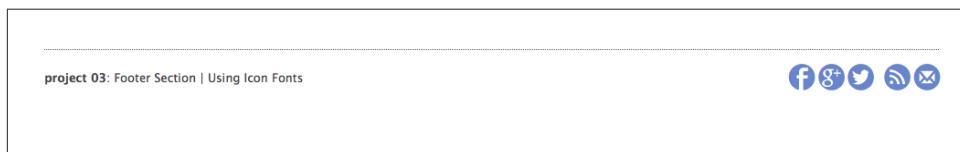
```
li#login input {  
    width: 122px;  
    padding: 4px;  
    border: 1px solid #ddd;  
    border-radius: 2px;  
    -moz-border-radius: 2px;  
    -webkit-border-radius: 2px;  
}  
}
```

The `border-radius` element makes the corners rounded of the HTML element it applied to. But why we repeat it three times with additional prefixes `-moz-` and `-webkit-`? These are so called *CSS vendor prefixes*, which allow the Web browser vendors to implement experimental CSS properties that haven't been standardized yet. For example, `-webkit-` is the prefix for all WebKit-based browsers: Chrome, Safari, Android, iOS. Microsoft uses `-ms-` for Internet Explorer, Opera uses `-o-`. These prefixes are temporary measures, which make the CSS files heavier than they need to be. The time will come when the CSS3 standard properties will be implemented by all browser vendors and you won't need to use these prefixes.

As a matter of fact, unless you want this code to work in the very old versions of Firefox, you can remove the line `-moz-border-radius: 2px;` from our `styles.css` because Mozilla has implemented the property `border-radius` in most of their browser. You can find a list of CSS properties with the corresponding vendor prefixes in [this list](#) maintained by Peter Beverloo.

## The Footer section

The footer section comes next. Run the project called `project-03-footer` and you'll see a new version of the Save The Child page with the bottom portion that looks as in [Figure 1-9](#). The footer section shows several icons linking to Facebook, Google Plus, Twitter, RSS feed, and e-mail.



*Figure 1-9. The footer section*

The HTML section of our first prototype is shown below. At this point it has a number of [tags](#), which have the dummy references `href="javascript:void(0)"` that don't redirect the user to any of these social sites.

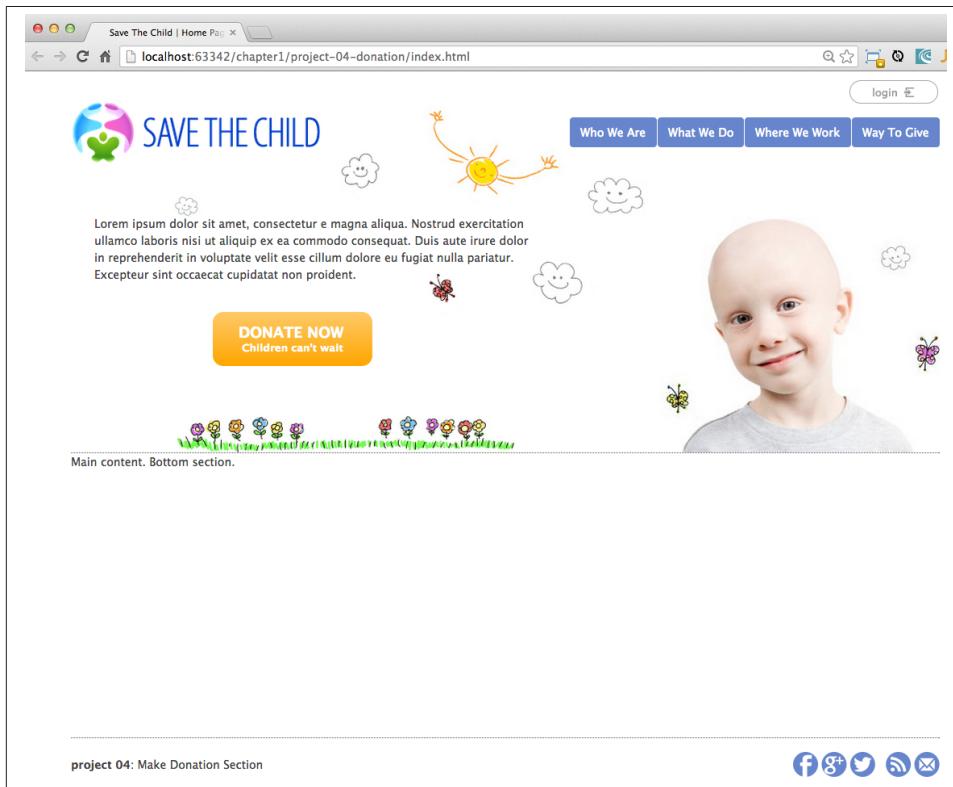
```
<footer>
  <section id="temp-project-name-container">
    <b>project 03</b>: Footer Section | Using Icon Fonts
  </section>
  <section id="social-icons">
    <a href="javascript:void(0)" title="Our Facebook page">
      <span aria-hidden="true" class="icon-facebook"></span></a>
    <a href="javascript:void(0)" title="Our Google Plus page">
      <span aria-hidden="true" class="icon-gplus"></span></a>
    <a href="javascript:void(0)" title="Our Twitter">
      <span aria-hidden="true" class="icon-twitter"></span></a> &nbsp;
    <a href="javascript:void(0)" title="RSS feed">
      <span aria-hidden="true" class="icon-feed"></span></a>
    <a href="javascript:void(0)" title="Email us">
      <span aria-hidden="true" class="icon-mail"></span></a>
    </section>
  </footer>
```

Each of the above anchors is styled using vector graphics icon fonts that we've selected and downloaded from <http://icomoon.io/app>. Vector graphics images are being re-drawn using vectors (strokes) as opposed to raster graphics, which are pre-drawn in certain resolution images. The raster graphics can give you these boxy pixelated images if the size of the image needs to be increased. We use the vector images for our footer section that are treated as fonts. They will look as good as originals on any screen size, besides you can change their properties (e.g. color) as easy as you'd do with any other font. The images that you see on [Figure 1-9](#) are located in the fonts directory of the project-03-footer. The IcoMoon web application will generate the fonts for you based on your selection and you'll get a sample html file, fonts, and CSS to be used with your application. Our icon fonts section in styles.css will look as follows:

```
/* Icon Fonts */
@font-face {
  font-family: 'icomoon';
  src: url('../fonts/icomoon.eot');
  src: url('../fonts/icomoon.eot?#iefix') format('embedded-opentype'),
       url('../fonts/icomoon.svg#icomoon') format('svg'),
       url('../fonts/icomoon.woff') format('woff'),
       url('../fonts/icomoon.ttf') format('truetype');
  font-weight: normal;
  font-style: normal;
}
```

## The Donate Section

The section with the Donate button and the donation form will be located in the top portion of page right below the navigation area. Initially, the page will open up with the background image of a sick but smiley boy on the right and a large Donate button on the left. The image shown on [Figure 1-10](#) is taken from a large collection of photos at [iStockphoto](#) Web site. We're also using two more background images here: one with the flowers, and the other with the sun and clouds, and you can find the references to these images in the styles.css file. Run the project-04-donation and you'll see the new version of or Save The Child page that will look as on [Figure 1-10](#).



*Figure 1-10. The initial view of the Donate section*

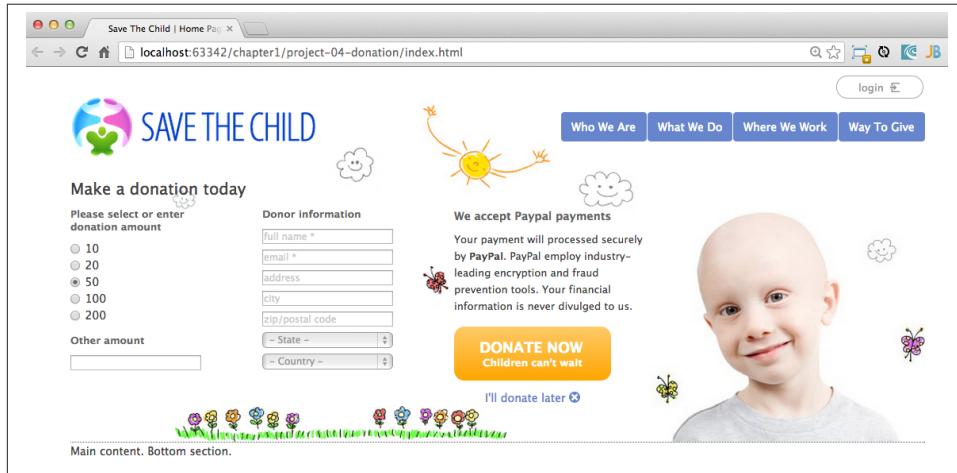
Lorem Ipsum is a dummy text widely used in printing, typesetting, and Web design. It's used as a placeholder to indicate the text areas that should be filled with a real content later on. You can read about it at <http://www.lipsum.com>. This is how the HTML fragment supporting [Figure 1-10](#) looks like (no CSS is shown for brevity).

```

<div id="donation-address">
    <p class="donation-address">
        Lorem ipsum dolor sit amet, consectetur e magna aliqua.
        Nostrud exercitation ullamco laboris nisi ut aliquip ex
        ea commodo consequat.
        Duis aute irure dolor in reprehenderit in voluptate velit
        esse cillum dolore eu fugiat nulla pariatur.
        Excepteur sint occaecat cupidatat non proident.
    </p>
    <button class="donate-button" id="donate-btn">
        <span class="donate-button-header">Donate Now</span>
        <br/>
        <span class="donate-2nd-line">Children can't wait</span>
    </button>
</div>

```

Clicking the button Donate should reveal the form where the user should be able to enter her name, address and the donation amount. Instead of opening a popup window we'll just change the content on the left revealing the form, and move the button Donate to the right. **Figure 1-11** shows how the top portion of our page will look like after the user clicks the Donate button.



*Figure 1-11. After clicking on Donate button*

The HTML of the donation form shown on **Figure 1-11** is shown below. When the user clicks on the Donate button the content of the form should be sent to PayPal or any other payment processing system.

```

<div id="donate-form-container">
    <h3>Make a donation today</h3>
    <form name="_xclick" action="https://www.paypal.com/cgi-bin/webscr" method="post">

```

```

<div class="donation-form-section">
    <label class="donation-heading">Please select or enter
        <br/> donation amount</label>
    <input type="radio" name = "amount" id= "d10" value = "10"/>
    <label for = "d10">10</label>
    <br/>
    <input type="radio" name = "amount" id = "d20" value="20" />
    <label for = "d20">20</label>
    <br/>
    <input type="radio" name = "amount" id="d50" checked="checked" value="50" />
    <label for="d50">50</label>
    <br/>
    <input type="radio" name = "amount" id="d100" value="100" />
    <label for="d100">100</label>
    <br/>
    <input type="radio" name = "amount" id="d200" value="200" />
    <label for="d200">200</label>
    <label class="donation-heading">Other amount</label>
    <input id="customAmount" name="amount" value=""
        type="text" autocomplete="off" />
</div>
<div class="donation-form-section">
    <label class="donation-heading">Donor information</label>
    <input type="text" id="full_name" name="full_name"
        placeholder="full name *" required>
    <input type="email" id="email_addr" name="email_addr"
        placeholder="email *" required>
    <input type="text" id="street_address" name="street_address"
        placeholder="address">
    <input type="text" id="city" name="scty" placeholder="city">
    <input type="text" id="zip" name="zip" placeholder="zip/postal code">
    <select name="state">
        <option value="" selected="selected"> - State - </option>
        <option value="AL">Alabama</option>
        <option value="WY">Wyoming</option>
    </select>
    <select name="country">
        <option value="" selected="selected"> - Country - </option>
        <option value="United States">United States</option>
        <option value="Zimbabwe">Zimbabwe</option>
    </select>
</div>

<div class="donation-form-section make-payment">
    <h4>We accept Paypal payments</h4>
    <p>
        Your payment will processed securely by <b>PayPal</b>.
        PayPal employ industry-leading encryption and fraud prevention tools.
        Your financial information is never divulged to us.
    </p>
    <button type="submit" class="donate-button donate-button-submit">

```

```

        <span class="donate-button-header">Donate Now</span>
        <br/>
        <span class="donate-2nd-line">Children can't wait</span>
    </button>
    <a id="donate-later-link" href="javascript:void(0);">I'll donate later
    <span class="icon-cancel"></span></a>
</div>
</form>
</div>

```

The JavaScript code supporting the UI transformations related to the button Donate is shown below. It's the code snippet from the main.js from project-04-donation. The click on the Donate button invokes the event handler `showDonationForm()`, which simply hides the `<div id="donation-address">` with *Loem Ipsum* and displays the donation form:

```
<form name="_xclick" action="https://www.paypal.com/cgi-bin/webscr" method="post">">.
```

When the form field loses focus or after the user clicked on the Submit button, the data from the form `_xclick` must be validated and sent to paypal.com. If the user clicks on "I'll donate later", the code hides the form and shows the Loem Ipsum from the `<div id="donation-address">` again.

Not including proper form validation is a sign of a rookie developer. This can easily irritate users. Instead of showing error messages like "Please include only numbers in the phone number field" use **regular expressions** to programmatically strip non-digits away.

Two `select` dropdowns in the code above contain hard-coded values of all states and countries. For brevity, we've listed just a couple of entries in each. In Chapter 2 we'll populate these dropdowns using the external data in JSON format.



Don't show all the countries in the dropdown unless your application is global. If the majority of the users of your country live in France, display on top of the list France and not Afghanistan (the first country in alphabetical order).

## Assigning Function Handlers. Take 1.

The next code fragment is an extract of JavaScript file main.js provide by Jerry. This code contains function handlers that process user clicks in the Donate section.

```

(function() {
    var donateButton = document.getElementById('donate-button');
    var donationAddress = document.getElementById('donation-address');
    var customAmount = document.getElementById('customAmount');
    var donateForm = document.forms['_xclick'];

```

```

var donateLaterLink = document.getElementById('donate-later-link');
var checkedInd = 2;

function showDonationForm() {
    donationAddress.style.display = "none";
    donateFormContainer.style.display = "block";
}

// Register the event listeners
donateButton.addEventListener('click', showDonationForm, false);
customAmount.addEventListener('focus', onCustomAmountFocus, false);
donateLaterLink.addEventListener('click', donateLater, false);
customAmount.addEventListener('blur', onCustomAmountBlur, false);

// Uncheck selected radio buttons if the custom amount was chosen
function onCustomAmountFocus() {
    for (var i = 0; i < donateForm.length; i++) {
        if (donateForm[i].type == 'radio') {
            donateForm[i].onclick = function() {
                customAmount.value = '';
            }
        }
        if (donateForm[i].type == 'radio' && donateForm[i].checked) {
            checkedInd = i;
            donateForm[i].checked = false;
        }
    }
}

function onCustomAmountBlur() {

    if (isNaN(customAmount.value)) {
        // The user haven't entered valid number for other amount
        donateForm[checkedInd].checked = true;
    }
}

function donateLater(){
    donationAddress.style.display = "block";
    donateFormContainer.style.display = "none";
}
})();

```

The code above contains an example of an inefficient code that in a loop assigns a click event handler to each radio button should the user click any radio button after visiting the Other Amount field. This was a Jerry's understanding of how to reset the value of the `customAmount` variable. Jerry was not familiar with the capture phase of the events that can intercept the click event on the radio buttons container's level and simply reset the value of `customAmount` regardless of which specific radio button is clicked.

## Assigning Function Handlers. Take 2.

Let's improve the code from the previous section. The idea is to intercept the click event during the capture phase (see the DOM Events section in [Appendix A](#)) and if the Event.target is any radio button, perform `customAmount.value = '';`

```
var donateFormContainer = document.getElementById('donate-form-container');

// Intercept any click on the donate form in a capturing phase
donateFormContainer.addEventListener("click", resetCustomAmount, true);
function resetCustomAmount(event){

    // reset the customAmount
    if (event.target.type=="radio"){
        customAmount.value = '';
    }
}
```

The code of the `onCustomAmountFocus()` doesn't need to assign function handlers to the radio buttons any longer:

```
function onCustomAmountFocus() {
    for (var i = 0; i < donateForm.length; i++) {
        if (donateForm[i].type == 'radio' && donateForm[i].checked) {
            checkedInd = i;
            donateForm[i].checked = false;
        }
    }
}
```

In the Donate section we started working with event handlers. You'll see many more examples of event processing throughout the book. In particular, Appendix A has a section DOM Events providing more details on the subject.

## Adding Video

In this section we'll add a video player to our Save The Child application. The goal is to play a short animation encouraging kids to fight the disease. We've hired a professional animation artist Yuri who has started working on the animation. Meanwhile let's take care of embedding the video player showing any sample video file.

### Adding the HTML5 Video Element

Let's run the project called `project-05-html5-video` to see the video playing, and after that we'll review the code. The new version of the Sick The Child app should look as in [Figure 1-12](#). The users will see an embedded video player on the right that can play the video located in the assets/media folder of the project `project-05-html5-video`.

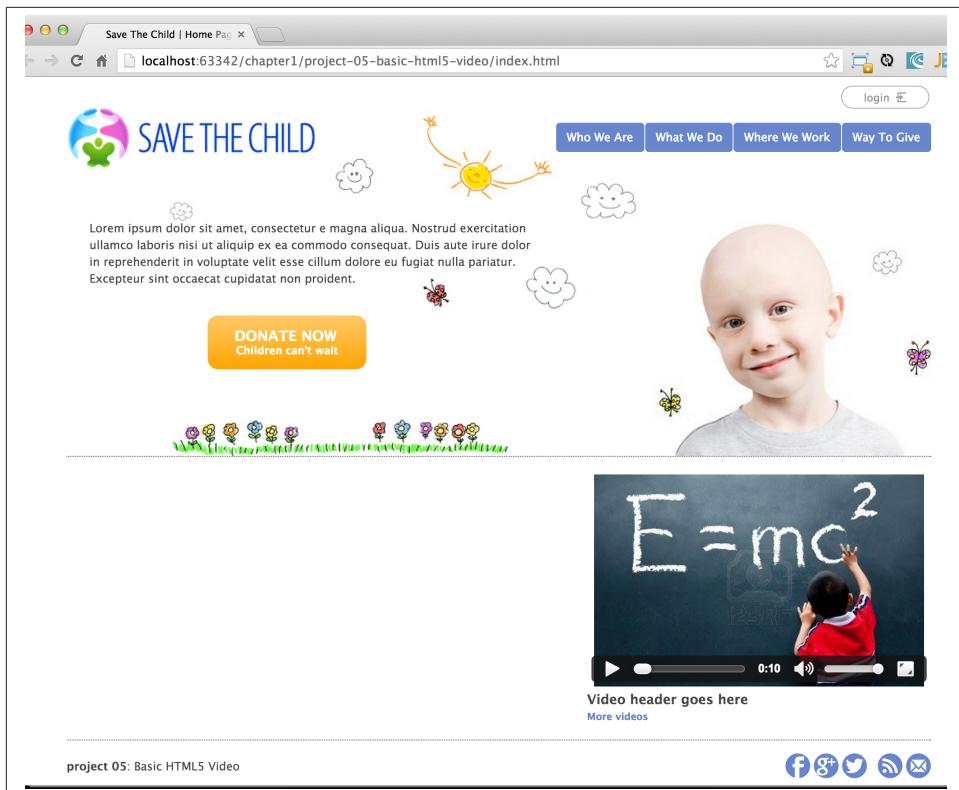


Figure 1-12. The video player is embeded

Let's see how our index.html has changed since its previous version. The bottom part of the main section includes the `<video>` tag. In the past, the videos in Web pages were played predominantly by the browser's Flash Player plugin (even older popular plugins included RealPlayer, MediaPlayer, and QuickTime). For example, you could have used the HTML tag '`<embed src="myvideo.swf" height="300" width="300">`' and if the user's browser supports Flash Player, that's all you needed for basic video play. While there were plenty of open source video players, creation of the enterprise-grade video player for Flash videos became an important skill for some software developers. For example, HBO, an American cable network offers an advanced multi-featured video player embedded into [www.hbogo.com](http://www.hbogo.com) for their subscribers.

In today's world most of the modern mobile Web browsers don't support Flash Player, and the video content providers prefer broadcasting videos in formats that are supported by all the browsers and can be embedded into Web page using the standard HTML5 element `<video>` (see its current working draft is published at <http://www.w3.org/wiki/HTML/Elements/video>).

The following code fragment illustrates how we've embedded the video into the bottom portion of our Web page (index.html). It includes two `<source>` elements, which allows to provide alternative media resources. If the Web browser supports playing video specified in the first `<source>` element, it'll ignore the other versions of the media. For example, the code below offers two versions of the video file: intro.mp4 (in H.264/MPEG-4 format natively supported by Safari and Internet Explorer) and intro.webm (WebM format for Firefox, Chrome, and Opera).

```
<section id="main-bottom-section">
  <div id="video-container">
    <video controls poster="assets/media/intro.jpg"
      width="390px" height="240" preload="metadata">

      <source src="assets/media/intro.mp4" type="video/mp4">
      <source src="assets/media/intro.webm" type="video/webm">
      <p>Sorry, your browser doesn't support video</p>
    </video>

    <h3>Video header goes here</h3>
    <h5><a href="javascript:void(0);">More videos</a></h5>
  </div>
</section>
```

The boolean property `controls` asks the Web browser to display the video player with controls (the play/pause buttons, the full screen mode, et al.) You can also control the playback programmatically in JavaScript. The `poster` property of the `<video>` tag specifies the image to display as a placeholder for the video - this is the image you see on [Figure 1-12](#). In our case `preload=metadata` instructs the Web browser to pre load just the first frame of the video and its metadata. Should we used `'preload="auto"`, the video would start loading in the background as soon as the Web page was loaded unless the user's browser doesn't allow it (e.g. Safari on iOS) for saving the bandwidth.

All major Web browsers released in 2011 and later (including Internet Explorer 9) come with their own embedded video players that support the '`<video>`' element. It's great that your code doesn't depend on the support of the Flash Player, but browsers' video players look different.

If neither .mp4 nor .webm files can be played, the content in the `<p>` tag displays the fallback message "Sorry, your browser doesn't support video". If you need to support older Web browsers that don't support HTML5 video, but support Flash Player, you can replace this `<p>` tag with the `<object>` and `<embed>` tags that embed another media file that Flash Player understands. Finally, if you believe that some users may have the browsers that support neither the `<video>` tag nor Flash Player, just add the links to the files listed in the `<source>` tags right after the closing `</video>` tag.

## Embedding YouTube Videos

Another way to include videos in your Web application is by uploading them to YouTube first and then embedding if into your Web page. This provides a number of benefits:

- The videos are hosted on Google's servers and use their bandwidth.
- The users can either watch the video as a part of your application's Web page or, by clicking on the YouTube logo on the status bar of the video player you can continue watching the video from its original YouTube URL.
- YouTube is streaming videos in the compressed form and the user can watch it as the bytes come in - it doesn't require a video to be fully preloaded to the user's device.
- YouTube stores videos in several formats and automatically selects the best one based on the user's Web browser (user agent).
- The HTML code to embed a YouTube video is generated for you by pressing the Share and then Embed link by the video itself.
- You can enrich your Web application by incorporating extensive video libraries by using the **YouTube Data API**. You can create fine tuned searches retrieving channels, playlists, videos, manage subscriptions, and authorize user requests.
- Your users can save the YouTube videos on their local drive using free Web Browsers add-ons like DownloadHelper extension for Firefox or a RealDownloader.

Embedding a YouTube video into your HTML page is simple. Find the page with the video on YouTube and press the links Share and Embed located right under the video. Then select the size of your video player and HTTPS encryption if needed (see Chapter 9 on Web security for reasoning). When this is done, copy the generated **iframe** section into your page.

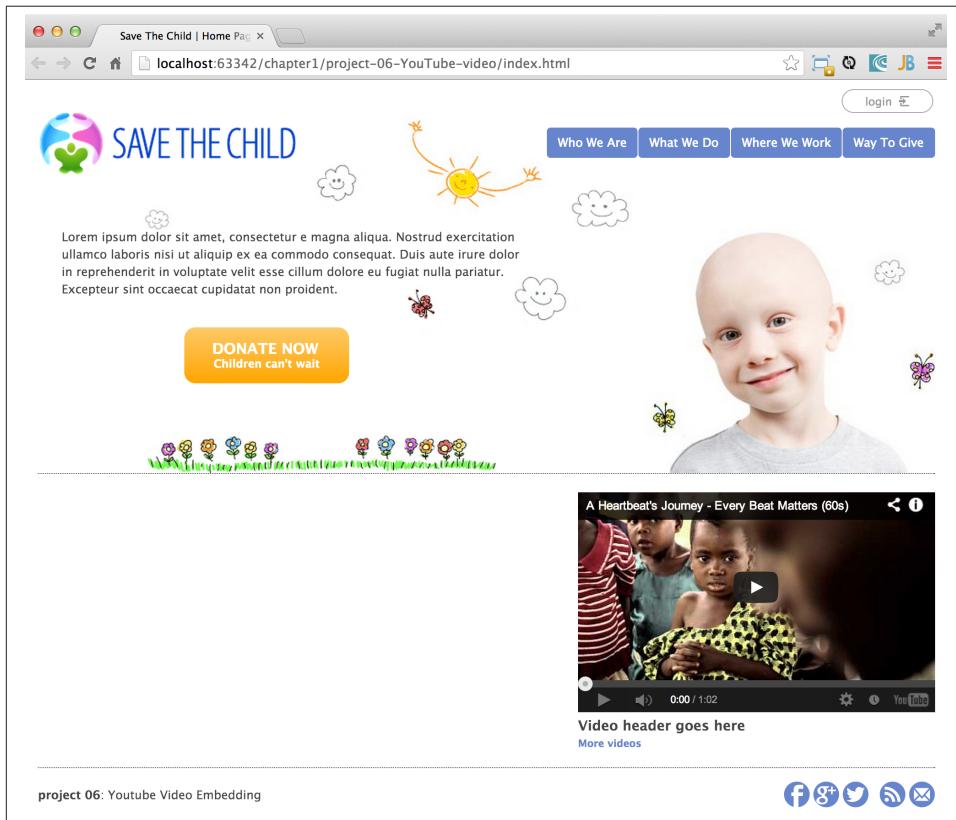
Open the file index.html in the project-06-YouTube-video and you'll see there a code that replaces the **<video>** tag of the previous project. It should look like this:

```
<section id="main-bottom-section">
  <div id="video-container">
    <div id="video-container">
      <iframe src="http://www.youtube.com/embed/VGZcer0hCuo?wmode=transparent&hd=1&vq=hd720"
              frameborder="0" width="390" height="240"></iframe>

      <h3>Video header goes here</h3>
      <h5><a href="javascript:void(0);">More videos</a></h5>
    </div>
  </div>
</section>
```

Note that the initial size of our video player is 390x240 pixels. The **<iframe>** wraps the URL of the video, which in this example ends with parameters **hd=1** and **vq=hd720**. This

is how you can force YouTube to load video in HD quality. Run the project-06-YouTube-video and it shows you a Web page that looks as in [Figure 1-13](#).



*Figure 1-13. The YouTube player is embeded*

Now let's do yet another experiment. Enter the URL of our video directly in your Web browser, turn on the Firebug or Chrome Developer Tools as explained in Appendix A. We did it in Firebug under Mac OS and selected the Net tab. Then HTML Response looked as in [Figure 1-14](#). YouTube recognized that this Web browser is capable of playing Flash content (FLASH\_UPGRADE) and picked the QuickTime as a fallback (QUICKTIME\_FALLBACK).

*Figure 1-14. HTTP Response object from YouTube*



Youtube offers an Opt-In Trial of HTML5 video, which allows the users to request playing most of the videos using HTML 5 video (even those recorded for Flash Player). Try to experiment on your own with and see if Youtube streams HTML5 videos in your browser.

Our brief introduction to embedding videos in HTML is over. Let's keep adding new features to the Save The Child Web application. This time we'll get familiar with the HTML5 Geolocation API.

# Adding Geolocation Support

HTML5 includes the Geolocation API that allows programmatically figure out the latitude and longitude of the user's device. Most of the people are accustomed to the non-Web GPS applications in cars or mobile devices that display maps and calculate distances

based on the current coordinates of the user's device or motor vehicle. But why do we need the Geolocation API in a desktop Web application?

The goal of this section is to demonstrate a very practical feature - finding registered Save The Children events based on the user's location. This way the users of this application can not only donate, but participate in such event or even find the needy children in a particular geographical area. In this chapter you'll just learn the basics of HTML5 GeoLocation API, but we'll continue improving the location feature of the Save The Child in the next chapter.



The World Wide Web Consortium has published proposed recommendation of the [Geolocation API Specification](#), which can become a part of HTML5 spec soon.

Does your old desktop computer have a GPS hardware? Most likely it doesn't. But its location can be calculated with varying degree of accuracy. If your desktop computer is connected to the network it has an IP address or your local Wi-Fi router may have an SSID given by the router vendor or your Internet provider so the location of your desktop computer is not a secret, unless you change the SSID of your Wi-Fi router. Highly populated areas have more Wi-Fi routers and cell towers so the accuracy increases. In any case, properly designed applications must always ask the user's permission to use the current location of her computer or other connected device.



The GPS signals are not always available. There are various location services that help identifying the position of your device. For example, Google, Apple, Microsoft, Skyhook and other companies use publicly broadcast Wi-Fi data from the wireless access point. Google Location Server uses Media Access Control (MAC) address to identify any device connected to the network.

Every Web Browser has a global object `window`, which includes the `navigator` object containing the information about the user's browser. If the browser's `navigator` object includes the property `geolocation`, geolocation services are available. While the Geolocation API allows you to get just a coordinate of your device and report the accuracy of this location, most applications use this information with some user-friendly UI, for example, the mapping software. In this section our goal is to demonstrate the following:

1. How to use Geolocation API
2. How to integrate the Geolocation API with Google Maps.
3. How to detect if the Web browser supports geolocation services



To respect people's privacy, Web browsers will always ask for permission to use Geolocation API unless the user changes the settings one the browser to always allow it.

## Geolocation Basics

The next version of our application is called project-07-basic-geolocation., where we simply assume that the Web browser supports the Geolocation. The Save The Child page will get a new container in the middle of the bottom main section, where we are planning to display the map of the current user location. But for now we'll show there just the coordinates: latitude, longitude, and the accuracy. Initially, the map container is empty, but we'll populate it from the JavaScript code as soon as the position of the computer is located.

```
<div id="map-container">  
/</div>
```

The following code snippet from main.js makes a call to the `navigator.geolocation` object to get the current position of the user's computer. In many code samples we'll use `console.log()` to print debug data in the Web browser's console.

```
var mapContainer = document.getElementById('map-container');           // ①  
  
function successGeoData(position) {  
    var successMessage = "We found your position!";                      // ②  
    successMessage += '\n Latitude = ' + position.coords.latitude;  
    successMessage += '\n Longitude = ' + position.coords.longitude;  
    successMessage += '\n Accuracy = ' + position.coords.accuracy +  
        console.log(successMessage);  
  
    var successMessageHTML = successMessage.replace(/\n/g, '<br />');  
    var currentContent = mapContainer.innerHTML;  
    mapContainer.innerHTML = currentContent + "<br />"  
        + successMessageHTML;                                              // ③  
}  
  
function failGeoData(error) {                                            // ④  
    console.log('error code = ' + error.code);  
  
    switch(error.code) {  
        case error.POSITION_UNAVAILABLE:  
            errorMessage = "Can't get the location";  
            break;  
        case error.PERMISSION_DENIED:  
            errorMessage = "The user doesn't want to share location";  
            break;
```

```

        case error.TIMEOUT:
            errorMessage = "Timeout - Finding location takes too long";
            break;
        case error.UNKNOWN_ERROR:
            errorMessage = "Unknown error: " + error.code;
            break;
    }
    console.log(errorMessage);
    mapContainer.innerHTML = errorMessage;
}

if (navigator.geolocation) {
    var startMessage = 'Your browser supports geolocation API :';
    console.log(startMessage);
    mapContainer.innerHTML = startMessage;
    console.log('Checking your position...');

    navigator.geolocation.getCurrentPosition(successGeoData,           // ⑤
                                              failGeoData,
                                              {maximumAge : 60000,           // ⑥
                                               enableHighAccuracy : true,
                                               timeout : 5000
                                              });
}

} else {
    mapContainer.innerHTML ='Your browser does not support geolocation';
}

```

- ① Get a reference to the DOM element `map-container` to be used for showing the results.
- ② The function handler to be called in case of the successful discovery of the computer's coordinates. If this function will be called it'll get a `position` object as an argument.
- ③ Display the retrieved data on the Web page (see [Figure 1-15](#)).
- ④ This is the error handler callback.
- ⑤ Invoke the method `getCurrentPosition()` passing it two callback function as arguments (for success and failure) and an object with optional parameters for this invocation.
- ⑥ Optional parameters: accept the cached value if not older than 60 seconds, retrieve the best possible results and don't wait for results for more than 5 seconds. You may not always want the best possible results to lower the response time and the power consumption.

If you run the project-07-basic-geolocation, the Browser will show a popup (it can be located under the toolbar) asking you a question similar to "Would you like to share

your location with 127.0.01?" Allow this sharing and you'll see a Web page, which will include the information about your computer's location similar to [Figure 1-15](#).



If you don't see the question asking permission to share location, check the privacy settings of your Web browser - most likely you've allowed using your location some time in the past.

A screenshot of a web browser displaying a charity website for "SAVE THE CHILD". The page features a smiling child, butterflies, and clouds. A yellow button labeled "DONATE NOW Children can't wait" is visible. On the left, there is placeholder text and a "DONATE NOW" button. Below the main content area, a message indicates the browser supports geolocation and provides the found position: Latitude = 40.71808689999996, Longitude = -74.01508919999999, and Accuracy = 102 meters. To the right, a video player shows a child writing "E=mc²" on a chalkboard, with a play button and a timestamp of 0:10. A caption below the video says "Video header goes here" and "More video link".

localhost:63342/chapter1/project-07-basic-geolocation/index.html

Who We Are | What We Do | Where We Work | Way To Give | login

SAVE THE CHILD

Who We Are | What We Do | Where We Work | Way To Give | login

DONATE NOW  
Children can't wait

Browser supports geolocation API  
Checking your position...  
We found your position!  
Latitude = 40.71808689999996  
Longitude = -74.01508919999999  
Accuracy = 102 meters

E=mc<sup>2</sup>

Video header goes here  
More video link

*Figure 1-15. The latitude and longitude are displayed*



If you want to monitor the position as it changes (the device is moving) use `geolocation.watchPosition()`, which implements internal timer and checks the position. To stop monitoring position use `geolocation.clearWatch()`.

## Integrating with Google Maps

Knowing the device coordinates is very important, but let's make the location information more presentable by feeding the device coordinates to **Google Maps API**. In this version of Save The Child we'll replace the gray rectangle from <>FIG3-15> with the Google maps container. We want the user to see a familiar map fragment with a pin pointing at the location of her Web browser. To follow our "Show and Tell" style let's see it working first. Run the project-08-geolocation-maps and you'll see a map with your current location as shown on **Figure 1-16**.

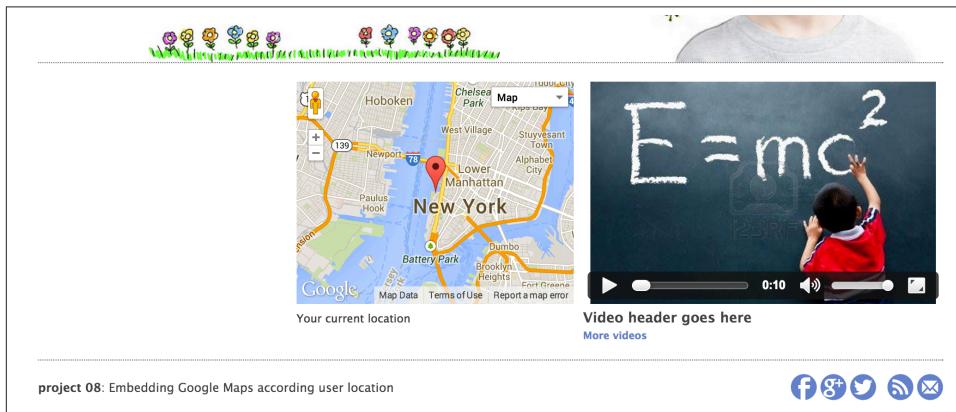


Figure 1-16. Showing your current location

Now comes the "Tell" part. First of all, take a look at the bottom of the index.html file. It loads Google's JavaScript library with their Map API (`sensor=false` means that we are not using a sensor like GPS locator):

```
<script src="http://maps.googleapis.com/maps/api/js?sensor=false"></script>
```

In the past Google required developers to obtain an API key and include it in the above URL. Although some Google's tutorials still mention the API key, it's not a must.



An alternative way of adding the `<script>` section to HTML page is by creating a script element. This gives you a flexibility of postponing the decision of which JavaScript to load. For example,

```
var myScript=document.createElement("script");
myScript.src="http://.....somelibrary.js";
document.appendChild(myScript);
```

Our main.js will invoke the function for Google's library as needed. The code that finds the location of your device is almost the same as in the section Geolocation Basics. We've replaced the call to `with geolocation.watchPosition()` so this program can modify

the position if your computer, tablet, or a mobile phone is moving. We store the returned value of the `watchPosition()` in the variable `watcherID` in case if you decide to stop watching the position of the device by calling `clearWatch(watcherID)`. Also, we lowered the value of the `maximumAge` option so the program will update the UI more frequently, which is important if you are running this program while in motion.

```
(function() {

    var locationUI = document.getElementById('location-ui');
    var locationMap = document.getElementById('location-map');
    var watcherID;

    function successGeoData(position) {
        var successMessage = "We found your position!";
        var latitude = position.coords.latitude;

        var longitude = position.coords.longitude;
        successMessage += '\n Latitude = ' + latitude;
        successMessage += '\n Longitude = ' + longitude;
        successMessage += '\n Accuracy = ' + position.coords.accuracy
            + ' meters';
        console.log(successMessage);

        // Turn the geolocation position into a LatLng object.
        var locationCoordinates =
            new google.maps.LatLng(latitude, longitude); // ①

        var mapOptions = {
            center : locationCoordinates,
            zoom : 12,
            mapTypeId : google.maps.MapTypeId.ROADMAP, // ②
            mapTypeControlOptions : {
                style : google.maps.MapTypeControlStyle.DROPDOWN_MENU,
                position : google.maps.ControlPosition.TOP_RIGHT
            }
        };

        // Create the map
        var map = new google.maps.Map(locationMap, mapOptions); // ③

        // set the marker and info window
        var contentString = '<div id="info-window-content">' +
            'We have located you using HTML5 Geolocation.</div>';

        var infowindow = new google.maps.InfoWindow({ // ④
            content : contentString,
            maxWidth : 160
        });

        var marker = new google.maps.Marker({ // ⑤
            position : locationCoordinates,
            map : map
        });

        infowindow.open(map, marker);
    }

    function errorGeoData(error) {
        var errorMessage = "An error occurred while trying to find your position: " + error.message;
        console.log(errorMessage);
    }

    watcherID = navigator.geolocation.watchPosition(successGeoData, errorGeoData, {
        maximumAge : 1000
    });
});
```

```

        map : map,
        title : "Your current location"

    });

    google.maps.event.addListener(marker, 'click', // 6
        function() {
            infowindow.open(map, marker);
        }
    );

    // When the map is loaded show the message and
    // remove event handler after the first "idle" event
    google.maps.event.addListenerOnce(map, 'idle', function(){
        locationUI.innerHTML = "Your current location";
    })
}

// error handler
function failGeoData(error) {
    clearWatch(watcherID);
    //the error processing code is omitted for brevity
}

if (navigator.geolocation) {
    var startMessage =
        'Browser supports geolocation API. Checking your location...';
    console.log(startMessage);

    var currentContent = locationUI.innerHTML;
    locationUI.innerHTML = currentContent + ' '+startMessage;

    watcherID = navigator.geolocation.watchPosition(successGeoData, // 7
        failGeoData,
        {
            maximumAge : 1000,
            enableHighAccuracy : true,
            timeout : 5000
        });
}

} else {
    console.log('browser does not support geolocation :(');
}
})();

```

- ① Google API represents a point in geographical coordinates (latitude and longitude) as a `LatLng` object, which we instantiate here.

- ② The object `google.maps.MapOptions` is an object that allows you to specify various parameters of the map to be created. In particular, the map type can be one of the following: HYBRID, ROADMAP, SATELLITE, TERRAIN. We've chosen the ROADMAP, which displays a normal street map.
- ③ The function constructor `google.maps.Map` takes two arguments: the HTML container where the map has to be rendered and the `MapOption` as parameters of the map.
- ④ Create an overlay box that will show the content describing the location (e.g. a restaurant name) on the map. You can do it programmatically by calling `InfoWindow.open()`.
- ⑤ Place a marker on the specified position on the map.
- ⑥ Show the overlay box when the user clicks on the marker on the map.
- ⑦ Invoke the method `watchPosition()` to find the current position of the user's computer.

This is a pretty basic example of the integrating GeoLocation with the mapping software. Google Maps API consists of dozens JavaScript objects and supports various events that allow you to build interactive and engaging Web pages that include maps. Refer to the [Google Maps JavaScript API Reference](#) for the complete list of available parameters (properties) of all objects used in project-08-geolocation-maps and more. In Chapter 2 you'll see a more advanced example of using Google maps - we'll read the JSON data stream containing coordinates of the children so the donors can find them based on the specified postal code.



For a great illustration of using Google Maps API look at the [PadMapper](#) Web application. We use it for finding rental apartments in Manhattan.

## Browser Features Detection With Modernizr

Now we'll learn how to use the detection features offered by a JavaScript library called [Modernizr](#). This is a must have feature detection library that helps your application to figure out if the user's browser supports certain HTML5/CSS3 features. Review the code of index.html from the project-08-1-modernizr-geolocation-maps. Note that the index.html includes two `<script>` sections - the Modernizr's JavaScript gets loaded first, while our own main.js is loaded at the end of the `<body>` section.

```
<!DOCTYPE html>

<html class="no-js lang=en">
```

```

<head>
  <meta charset="utf-8">

  <title>Save The Child | Home Page</title>
  <link rel="stylesheet" href="assets/css/styles.css">

  <script src="js/libs/modernizr-2.5.3.min.js"></script>

</head>
<body>
  <!-- Most of the HTML markup is omitted for brevity --!

    &lt;script src="js/main.js"&gt;&lt;/script&gt;
  &lt;/body&gt;
&lt;/html&gt;
</pre>

```

Modernizr is an open source JavaScript library that helps your script to figure out if the required HTML or CSS features are supported by the user's browser. Instead of maintaining complex cross-browser feature matrix to see if, say `border-radius` is supported in the user's version of Firefox, the Modernizer queries the `<html>` elements to see what's supported and what's not.

Note the following fragment on the top of `index.html`: `<html class="no-js" lang="en">`. For Modernizr to work, your HTML root element has to include the class named "no-js". On page load, the Modernizr will replace the `no-js` class with its extended version that lists all detected features, and those that are not supported will get a prefix `no-`. Run `index.html` from `project-08-1-modernizr-geolocation-maps` in Chrome and you'll see using Developer Tools Panel that the values of the `class` property of the `html` element are different now, and you can see from [Figure 1-17](#) that our version of Chrome doesn't support touch events (`no-touch`) and flexbox (`no-flexbox`).

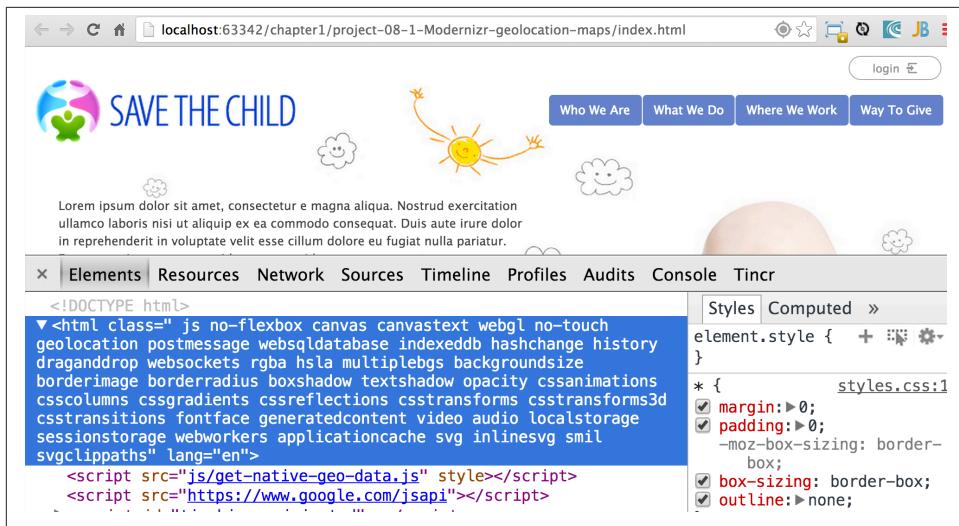


Figure 1-17. Modernizr changed the HTML's class property

For example, there is a new way to do page layouts using called CSS Flexible Box Layout Module. This feature is not widely supported yet, and as you can see from Figure 1-17, our Web browser doesn't support it at the time of this writing. If the CSS file of your application will implement two class selectors `.flexbox` and `.no-flexbox` then the browsers that support flexible boxes will use the former and the older browsers - the latter.

When Modernizr loads it creates a new JavaScript object `window.Modernizr` with lots of boolean properties indicating if a certain feature is or is not supported. Add the `Modernizr` object as a Watch Expression in the Chrome Developer Tools panel and see which properties have the `false` value (see Figure 1-18).

The screenshot shows the Chrome DevTools Elements tab open on a page from 'SAVE THE CHILD'. The page features a logo of two children holding hands, a sun, and clouds. The navigation bar includes 'Who We Are', 'What We Do', and 'Where We'. Below the navigation, there's a toolbar with icons for copy, paste, find, etc. The main pane shows the 'Sources' tab selected, displaying a tree view of resources. On the right, the 'Elements' panel shows the properties of the 'window.Modernizr' object. The properties listed include:

- visitsCursor: <not available>
- Modernizr: Object
  - \_cssomPrefixes: Array[4]
  - \_domPrefixes: Array[4]
  - \_prefixes: Array[6]
  - \_version: "2.5.3"
  - addTest: function (a,b){if(typeof a=="string")a=""+a;var c=typeof b=="function"?b:a;return function(d){c.call(this,d);var e=d.documentElement,f=d.createElement("div");f.setAttribute("class",a);e.appendChild(f);var g=f.offsetWidth==0?\_testWidth(a):null;f.parentNode.removeChild(f);return g}}();
  - applicationcache: true
  - audio: Boolean
  - backgroundsize: true
  - borderimage: true
  - borderradius: true
  - boxshadow: true
  - canvas: true
  - canvastext: true
  - cssanimations: true
  - csscolumns: true
  - cssgradients: true
  - cssreflections: true
  - csstransforms: true
  - csstransforms3d: true
  - csstransitions: true
  - draganddrop: true
  - flexbox: false
  - fontface: true
  - generatedcontent: true

Figure 1-18. `window.Modernizr` object

Hence your JavaScript code can test if certain features are supported or not.

What if the Modernizer detects that a certain feature is not supported yet by a user's older browser? You can include polyfills in your code that replicate the required functionality. You can write such a polyfill on your own or pick one from the collection that is located at [Modernizr's Github repository](#).



Addy Osmani published [The Developer's Guide To Writing Cross-Browser JavaScript Polyfills](#)

The Development version of Modernizr weighs 42Kb and can detect lots of features. But you can make it smaller by configuring the detection of only selected features. Just visit [Modernizr](#) and press the red Production button that will allow to configure the build specifically for your application. For example, if you're just interested to detect the HTML5 video support, the size of the generated Modernizr library will be reduced to under 2Kb.

Let's review the relevant code from project-08-1-modernizr-geolocation-maps that illustrate the use of Modernizr. In particular, Modernizr allows you to load one or the other JavaScript code based on the result of some tests.



Actually, the Modernizr loader internally utilizes a tiny (under 2Kb) resource loader library [yepnope.js](#), which can load both JavaScript and CSS. This library is integrated in Modernizr, but we just wanted to give a proper recognition to yepnope.js, which can be used as an independent resource loader too.

```
(function() {  
    Modernizr.load({  
        test: Modernizr.geolocation,  
  
        yep: ['js/get-native-geo-data.js', 'https://www.google.com/jsapi'],  
  
        nope: ['js/get-geo-data-by-ip.js', 'https://www.google.com/jsapi'],  
  
        complete : function () {  
            google.load("maps", "3",  
                {other_params: "sensor=false", 'callback':init});  
        }  
    });  
})();
```

The code above invokes the function `load()`, which can take different arguments, but our example uses as an argument a specially prepared object with five properties: `test`, `yep`, `nope`, `complete`. The `load()` function will test the value of `Modernizr.geolocation` and if it's true, it'll load the scripts listed in the `yep` property. Otherwise it'll load the code listed in the `nope` array. The code in the `get-native-geo-data.js` gets the user's location the same way as it was done earlier in the section Integrating with Google Maps.

Now let's consider the “`nope`” case. The code of the `get-geo-data-by-ip.js` has to offer an alternative way of getting the location for the browsers that don't support HTML5 Geolocation API. We found the GeoIP JavaScript API offered by [MaxMind, Inc.](#). Their service returns country, region, city, latitude and longitude, which can serve as a good

illustration of how a workaround of a non-supported feature can be implemented. The code in get-geo-data-by-ip.js is very simple for now.

```
function init(){

  var locationMap = document.getElementById('location-map');
  locationMap.innerHTML="Your browser does not support HTML5 geolocation API. ";

  // The code to get the location by IP from http://j.maxmind.com/app/geoip.js
  // will go here. Then we'll pass the latitude and longitude values to
  // Google Map API for drawing the map.

}
```

Most likely your browser supports HTML5 geolocation API, and you'll see the map created by the script get-native-geo-data.js. But if you want to test a non-supported geolocation (the nope branch) either try this code in the older browser or change the test condition to look like this: `Modernizr.fakegeolocation`,

Google has several JavaScript APIs, for example, Maps, Search, Feeds, Earths et al. Any of these APIs can be loaded by [Google AJAX Loader](#) `google.load()`. This is more generic way of loading any APIs comparing to loading maps from <http://maps.googleapis.com/maps/api> in the previous section on integrating geolocation and maps. The process of loading of the Google code with Google AJAX Loader consists of two steps:

1. Load Google's common loader script from <https://www.google.com/jsapi>
2. Load the concrete module API specifying its name, version and optional parameters. In our example we are loading the maps API of version 3 passing an object with two properties: `sensor=false` and the name of the callback function to invoke right after the mapping API completes loading: `'callback':init`.



If you want to test your Web page in the specific old version of a particular Web browser, you can find their distributions at [old-apps.com](http://old-apps.com). For example, you can find all the old version of Firefox for [Mac OS](#) and for [Windows](#).

## Search and Multi-Markers With Google Maps

We've prepared for you a couple of more examples just to showcase the features of Google Maps API. The working examples will be included in the code accompanying the book, and we'll provide very brief explanations below.

The project-09-map-and-search is an example of address search using Google Maps API. [Figure 1-19](#) shows a fragment of the Save The Child page after we've entered the address "26 Broadway ny ny" in the search field. You can do a search by city or a zip

code too. This can be a useful feature if you'd want to allow the users search for the children living in a particular geographical area so their donation would be directed to specific people.

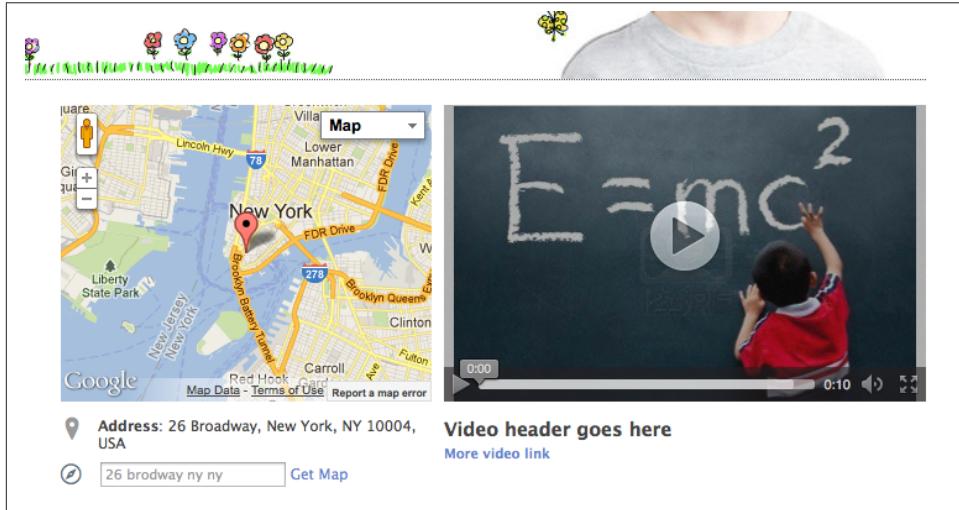


Figure 1-19. Searching by Address

Our implementation of the search is shown in the code fragment from main.js. It uses geocoding, which is a process of converting an address into geographic coordinates (latitude and longitude). If the address is found, the code places a marker on the map.

```
var geocoder = new google.maps.Geocoder();

function getMapByAddress() {
  var newaddress = document.getElementById('newaddress').value;

  geocoder.geocode(                                     // ①
    {'address' : newaddress,
     'country' : 'USA'
   },

  function(results, status) {                         // ②
    console.log('status = ' + status);

    if (status == google.maps.GeocoderStatus.OK) {

      var latitude = results[0].geometry.location.lat();           // ③
      var longitude = results[0].geometry.location.lng();

      var formattedAddress = results[0].formatted_address;
      console.log('latitude = ' + latitude +
                  ' longitude = ' + longitude);
    }
  }
}
```

```

        console.log('formatted_address = ' + formattedAddress);

        var message = '<b>Address</b>: ' + formattedAddress;
        foundInfo.innerHTML = message;

        var locationCoordinates =
            new google.maps.LatLng(latitude, longitude);           // ④
        showMap(locationCoordinates, locationMap);

    } else if (status == google.maps.GeocoderStatus.ZERO_RESULTS) { // ⑤
        console.log('geocode was successful but returned no results. ' +
        'This may occur if the geocode was passed a non-existent ' +
        'address or a latlng in a remote location.');
    }

    } else if (status == google.maps.GeocoderStatus.OVER_QUERY_LIMIT) {
        console.log('You are over our quota of requests.');

    } else if (status == google.maps.GeocoderStatus.REQUEST_DENIED) {
        console.log('Your request was denied, ' +
        'generally because of lack of a sensor parameter.');

    } else if (status == google.maps.GeocoderStatus.INVALID_REQUEST) {
        console.log('Invalid request. ' +
        'The query (address or latlng) is missing.');
    }
});
}

```

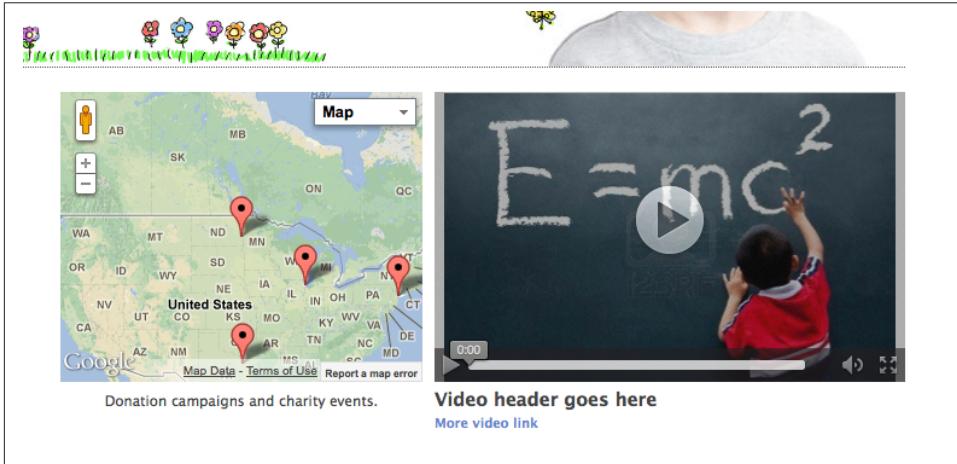
- ➊ Initiate request to the Geocoder object providing the GeocodeRequest object with the address and a function to process the results. Since the request to the Google server is asynchronous, the function is a callback.
- ➋ When the callback will be invoked, it'll get an array with results.
- ➌ Get the latitude and longitude from the result.
- ➍ Prepare the LatLng object and give it to the mapping API for rendering.
- ➎ Process errors.

The Geocoding API is simple and free to use until your application reaches a certain number of requests. Refer to [Google Geocoding API documentation](#) for more details. If your application is getting the OVER\_QUERY\_LIMIT you ned to contact Google Maps API for Business sales team for information on licensing options.

## Adding Multiple Markers on the Map

Jerry has yet another idea: show multiple markers on the map reflecting several donation campaigns and charity events that are going on at various locations. If we display this information on the Save The Child page more people may participate with their donations or other ways. We've just learned how to do an address search on the map, and

if the application has an access to the data about charity events, we can display them as the markers on the map. Run the project-10-maps-multi-markers and you'll see a map with multiple markers as in [Figure 1-20](#)



*Figure 1-20. Multiple markers on the map*

The JavaScript fragment below displays the map with multiple markers. In this example the data is hard-coded in the array `charityEvents`, but in the next chapter we'll modify this example and will get the data from a file in a JSON form. The for-loop creates a marker for each of the event listed in the array `charityEvents`. Each element of this array is also an array containing the name of the city and state, the latitude and longitude, and the title of the charity event. You can have any other attributes of the charity events stored in such an array and display them when the user clicks on a particular marker in an overlay by calling `InfoWindow.open()`.

```
(function() {  
  
    var locationUI = document.getElementById('location-ui');  
    var locationMap = document.getElementById('location-map');  
  
    var charityEvents = [[['Chicago, IL', 41.87, -87.62, 'Giving Hand'],  
        ['New York, NY', 40.71, -74.00, 'Lawyers for Children'],  
        ['Dallas, TX', 32.80, -96.76, 'Mothers of Asthmatics '],  
        ['Miami, FL', 25.78, -80.22, 'Friends of Blind Kids'],  
        ['Miami, FL', 25.78, -80.22, 'A Place Called Home'],  
        ['Fargo, ND', 46.87, -96.78, 'Marathon for Survivors']]  
    ];  
  
    var mapOptions = {  
        center : new google.maps.LatLng(46.87, -96.78),  
        zoom : 3,  
    };  
});
```

```

        mapTypeId : google.maps.MapTypeId.ROADMAP,
        mapTypeControlOptions : {
            style : google.maps.MapTypeControlStyle.DROPDOWN_MENU,
            position : google.maps.ControlPosition.TOP_RIGHT
        }
    });

var map = new google.maps.Map(locationMap, mapOptions);

var infowindow = new google.maps.InfoWindow();

var marker, i;

// JavaScript forEach() function is deprecated,
// hence using a regular for loop
for ( i = 0; i < charityEvents.length; i++) {
    marker = new google.maps.Marker({
        position : new google.maps.LatLng(charityEvents[i][1],
                                            charityEvents[i][2]),
        map : map
    });

    google.maps.event.addListener(marker, 'click', (function(marker, i) {
        return function() {
            var content = charityEvents[i][0] + '<br/>' + charityEvents[i][3];
            infowindow.setContent(content);
            infowindow.open(map, marker);
        }
    })(marker, i));

    google.maps.event.addListenerOnce(map, 'idle', function(){
        locationUI.innerHTML = "Donation campaigns and charity events.";
    })
}

})();

```

## Summary

This chapter has described the process of mocking the future Web site on by our Web Designer Jerry, who went a lot further than creating a number of images with short descriptions. Jerry created a working prototype of the Save The Child page. Keep in mind that Jerry and his fellow Web designers like creating good looking Web pages.

But us, Web developers, need to worry about other things like making Web pages responsive and light weight. The first thing you need to do after receiving the prototype of your Web application from Jerry is run it through Google Developer Tools or Firebug (see the Debugging JavaScript section in Appendix A for details) and measure the total size of the resources being downloaded from the server. If it loads 1Mb or more worth of images, ask Jerry to review the images and minimize their size.

The chances are that you don't need to download all the JavaScript code at once - we'll discuss modularization of large applications in Chapter 6.

The next phase of improving this prototype is to remove the hard-coded data from the code and place them into external files. The next chapter will cover the JSON data format and how to fill our single-page application with the data using a set of techniques called AJAX.

## CHAPTER 2

# Using AJAX and JSON

This chapter is about bringing the external data to HTML Web pages. In the previous chapters we've been using only hard-coded data - our goal was to see how to layout the Web page and how to change the layout in case some events have happened, e.g. the user clicked on the menu button. Now we'll make sure that our single-page application Save The Child can request the data from the external sources and send them the data too. This comes down to two questions:

1. How can an HTML Web page exchange the data with Web servers?
2. What format to use for presenting the application data?

While there could be different answers to these questions, we'll be using AJAX techniques as an answer to the first question and JSON data format as an answer to the second one. We'll start this chapter with explaining why AJAX and JSON are appropriate choice for the Save The Child Web application and many others.

## What's AJAX

In the early days of the Internet, every new page, whether on the same or separate web site, required a new request and response to the web server. This, in turn, would re-render the entire contents of the new page. If a user points her Web browser to one URL and then changes it to another, the new request will be sent to the new URL, and the new page will arrive and will be rendered by the browser. The URL may have been changed not because the user decided to go to visit a different Web site, but simply because the user selected a menu item that resulted in bringing a new Web page from the same domain. This was pretty much the only way to design Web sites in the 90th.

Back in 1999, Microsoft decided to create a Web version of Outlook - their popular eMail application. Their goal was to be able to modify the Input folder as the new emails arrive, but without refreshing the entire content of the Web page. They created an Ac-

tiveX control called `XMLHTTP` that lived inside Internet Explorer 5 and could make requests to the remote servers receiving data without the need to refresh the entire Web page. Such a Web Outlook client would make periodic requests to the mail server, and if the new mail arrived the application would insert the new row on the top of the Inbox by direct change of the DOM object from JavaScript.

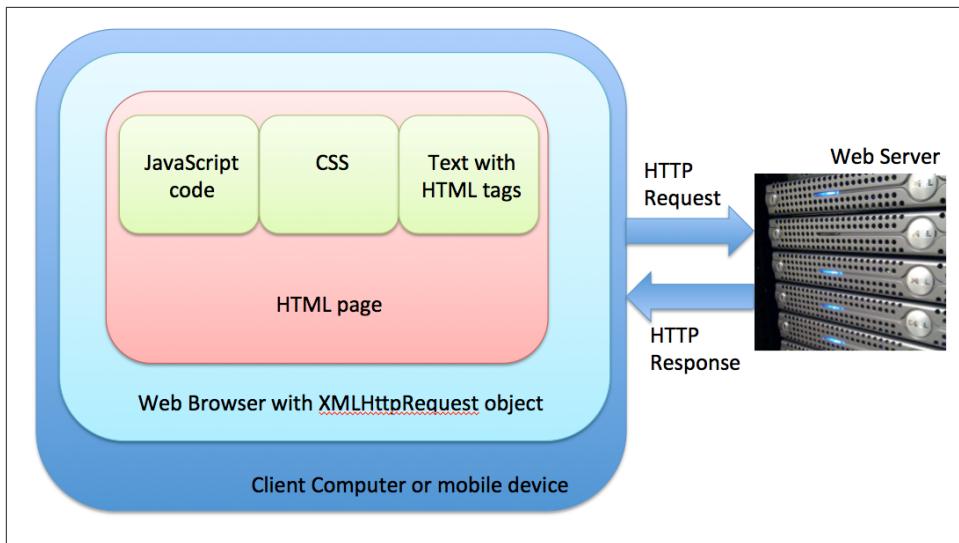
In early 2000th, other Web browsers implemented their own versions of `XMLHttpRequest`. Its Working Draft 6 is [published by W3C](#). Google created their famous email client GMail and Map Web applications. In 2005 Jesse James Garrett wrote an article titled “AJAX: A New Approach to Web Applications”. The Web developer community liked the term **AJAX**, which stand for Asynchronous JavaScript and XML, and this gave a birth to the new breed of Web applications that could update the content of just the portion of a Web page without re-retrieving the entire page. Interestingly enough, the last letter in the **AJAX** acronym stands for XML, while realistically presenting the data in the XML form is not required, and currently is seldom used as a data format in the client-server exchanges. JSON is used a lot more often to represent the data, but apparently AJAJ didn't sound as good as **AJAX**.

Visit the [Google Finance](#) or [Yahoo! Finance](#) Web pages when the stock market is open, and you'll see how the price quotes or other financial indicators are changing while the most of the content remains the same. This gives an illusion of the server pushing the data to your Web client. But most likely, it not a data push but rather periodic *polling* of the server's data using **AJAX**. In the modern Web applications we expect to see more of the real server side data push using **HTML5 WebSockets API**, which is described in details in Chapter 8 of this book.

## What's **JSON**

**JSON** stands for **JavaScript Object Notation**. It's a more compact than XML way to represent the data. Besides, all modern Web Browsers understand and can parse **JSON** data. After learning the **JavaScript object literals** in Chapter 1, you'll see that the presenting the data in **JSON** notation is almost the same as writing **JavaScript object literals**.

**Figure 2-1** depicts a high level view of a typical Web application. All of the code samples from Chapter 1 (and in Appendixes) were where written in **HTML**, **JavaScript** and **CSS**. In this chapter will add to the mix the `XMLHttpRequest` object that will send and receive the **JSON** content wrapped into `HTTPRequest` and `HTTPResponse` objects.



*Figure 2-1. Anatomy of a Web application*

When a Web page is loaded the user doesn't know (and doesn't have to know) that the page content was brought from several servers that can be located thousands miles apart. More often than not, when the user enters the URL requesting an HTML document, the server side code can engage several servers to bring all the data requested by the user. Some of the data are retrieved by making calls to one or more Web Services. The legacy Web services were build using SOAP + XML, but majority of today's Web services are build using lighter *RESTful architecture*, and JSON has become a de facto standard data exchange format of the REST Web services.

## How AJAX Works

Imagine a single-page application that needs some data to refresh in real time. For example, our Save The Child application includes an online auction where people can bid and purchase some goods as a part of a charity event. If John from New York placed a bid on certain auction item, and some time later Mary from Chicago placed the higher bid on the same item, we want to make sure that John knows about it immediately, in real time. This means that the server-side software that received Mary's bid has to push this data to all users who expressed their interest in the same item.

But the server has to send and the browser has to modify only the new price while the rest of the content of the Web page should remain the same. You can implement behavior using AJAX. But first, the bad news: you can't implement the real-time server side push with AJAX. You can only emulate this behavior by using polling techniques, when the

`XMLHttpRequest` object sits inside your JavaScript code and periodically sends HTTP requests to the server asking if there were any changes in bids since the last request.

If, for instance, the last request was made at 10:20:00AM, the new bid was placed at 10:20:02AM, and the application makes a new request (and updates the browser's window) at 10:20:25AM, this means that the user will be notified about the price change with a three second delay. AJAX is still request-response based way of getting the server's data, and strictly speaking, doesn't offer a real real-time updates. Some people use the term *near real time* notifications.

Another bad news is that AJAX uses HTTP protocol for the data communication, which means that a substantial overhead in the form of `HTTPResponse` header will be added to the new price, and it can be as large as several hundred bytes. This is still better than sending the entire page to the Web browser, but HTTP adds a hefty overhead.



We'll implement such an auction in Chapter 8 using a lot more efficient protocol called WebSockets, which supports a real-time data push and adds only several extra bytes to the data load.

## Retrieving Data From the Server

Let's try to implement AJAX techniques by implementing a data retrieval. The process of making an AJAX request is well defined and consists of the following steps:

- Create an instance of `XMLHttpRequest` object.
- Initialize the request to your data source by invoking the method `open()`.
- Assign the a handler to the `onreadystatechange` attribute to process server's response.
- Make a non-blocking request to the data source by calling `send()`.
- In your handler function process the response when it arrives from the server. This is where *asynchronous* came from - the handler can be invoked at any time whenever the server prepares the response.
- Modify the DOM elements based on the received data, if need be.



Most likely you are going to be using one of the popular JavaScript frameworks, which will spare you from knowing all these details, but knowing how it works under the hood can be beneficial.

In most of the books on AJAX you'll see browser-specific ways of instantiating the XMLHttpRequest object (a.k.a. XHR). Most likely you'll be developing your application using some JavaScript library or a framework and all browser specifics in instantiating of the XMLHttpRequest will be hidden from you. Chapters 3 and 4 include such examples, but let's stick to the standard JavaScript way implemented by all modern browsers:

```
var xhr = new XMLHttpRequest();
```

The next step is to initialize a request by invoking the method `open()`. You need to provide the HTTP method (GET, POST et al.), the URL of the data source. Optionally, you can provide three more arguments: a Boolean variable indicating if you want this request to be processed asynchronously (which is the default), and the user id and password if the authentication is required. Keep in mind, that the following method does not request the data yet.

```
xhr.open('GET', dataUrl);
```



Always use HTTPS protocol if you need to send the user id and password. Using secure HTTP should be your preferred protocol in general (read more in Chapter 9).

XHR has an attribute `readyState`, and as soon as it changes the callback function assigned to the `onreadystatechange` will be invoked. This callback should contain your application specific code to analyze the response and process it accordingly. Assigning such a callback is pretty simple:

```
xhr.onreadystatechange = function(){...}
```

Inside such a callback function you'll be analyzing the value of the XHR's attribute `readyState`, which can have one of the following values:

*Table 2-1. States of the Request*

Value	State	Description
0	UNSENT	the XHR has been constructed
1	OPENED	open() was successfully invoked
2	HEADERS_RECEIVED	All HTTP headers has been received
3	LOADING	The response body is being received
4	DONE	the data transfer has been completed

Finally, send the AJAX request for data. The method `send()` can be called with or without parameters depending on if you need to send the data to the server or not. In its simplest form the method `send()` can be invoked as follows:

```
xhr.send();
```

The complete cycle of the `readyState` transitions is depicted in Figure 2-2

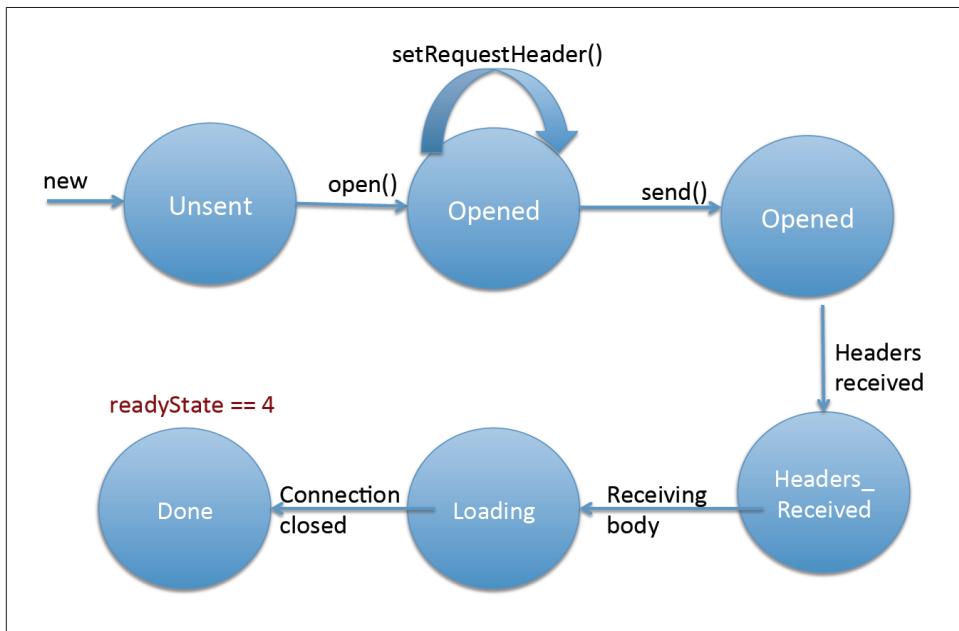


Figure 2-2. Transitions of the `readyState` attribute

Let's spend a bit more time discussing the completion of the this cycle when server's response is received and the XHR's `readyState` is equal to 4. This means that we've got something back, which can be either the data we've expected or the error message. We need to handle both scenarios in the function assigned to the `onreadystatechange` attribute. This is a common way to do it in JavaScript without using frameworks:

```
xhr.onreadystatechange = function(){
  if (xhr.readyState == 4) {
    if((xhr.status >=200 && xhr.status <300) || xhr.status==304) {
      // We got the data. Get the value from one of the response attributes
      // e.g. xhr.responseText and process the data accordingly.
    } else {
      // We got an error. Process the error code and
      // display the content of the statusText attribute.
    }
  }
}
```

```
};
```

One note about the third line of the code above. Here we're checking the **HTTP status code** received from server. W3C splits the HTTP codes into groups. The codes numbered as 1xx are informational, 2xx are successful codes, 3xx are about redirections, 4xx represent bad requests (like infamous 404 for Not Found), and 5xx for server errors. That's why the above code fragment checks for all 2xx codes and 304 - the data was not modified and taken from cache.



If your application needs to post the data to the server, you need to open the connection to the server with the `POST` parameter. You'll also need to set the HTTP header attribute `Content-type` to either `multipart/form-data` for large-size binary data or to `application/x-www-form-urlencoded` (for forms and small-size alphanumeric data). Then prepare the data object and invoke the method `send()`:

```
var data="This is some data";
xhr.open('POST', dataUrl, true);
xhr.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');

...
xhr.send(data);
```



`XMLHttpRequest Level 2` adds some new functionality like `FormData` object, timeouts, `ArrayBuffers` and more. It's supported **by most** of the Web browsers.

## AJAX: Good and Bad

AJAX techniques have their pros and cons. You saw how easy it was to create a Web page that didn't have to refresh itself, but provided the users with the means of communicating with the server. This certainly improves the user experience. The fact that AJAX allows you to lower the amount of data that goes over the wire is important too. Another important advantage of AJAX that it works in a standard HTML/JavaScript environment and is supported by all Web browsers. The JavaScript frameworks hides all the differences in instantiating `XMLHttpRequest` and simplify making HTTP requests and processing responses. Since the entire page is not reloaded, you can create "fat clients" that keep certain data preloaded once and reused in your JavaScript in different use cases. With AJAX you can lazy load some content as needed rather than loading everything at once. Taken for granted auto-completion feature would not be possible in HTML/JavaScript application without the AJAX.

On the bad side, with AJAX the user loses the functionality of the browser's Back button, which reloads the previous Web page while the user could expect to see the previous state of the same page.

Since the AJAX brings most of the content dynamically, the search engines might not rank your Web pages as high as it would do if the content was statically embedded in the HTML. If discoverability of your Web application is important, some extra steps should be taken to make it more SEO-friendly, e.g. using an [SEO Server](#).

Increasing the amount of AJAX interactions means that your application will have to send more of the JavaScript code to the Web browser, which increases the complexity of programming and decreases the scalability of your application.



Using HTML5 History API (see Chapter 1) will help you in teaching the old dog (the browser's Back button) new tricks.

AJAX applications are subject to [\*the same origin policy\*](#) (the same protocol, host name, and port) allowing XMLHttpRequest make HTTP requests only to the domains where the Web application was loaded from. It's a security measure to limit the ability of JavaScript code to interact with resources that arrive to the Web browser from a different Web server.



W3C has published a working draft of [\*Cross-Origin Resource Sharing\*](#) (CORS) - a mechanism to enable client-side cross-origin requests.

## Populating States and Countries from HTML Files

To see the first example where we are using AJAX in our Save The Child application run the project-01-donation-ajax-html, where we've removed the hard-coded data about countries and states from HTML and saved them in two separate files: data/us-states.html and data/countries.html. In this project the file index.html has two empty comboboxes (`<select>` elements):

```
<select name="state" id="state">
    <option value="" selected="selected"> - State - </option>
    <!-- AJAX will load the rest of content -->
</select>
<select name="country" id="countriesList">
    <option value="" selected="selected"> - Country - </option>
```

```
<!-- AJAX will load the rest of content -->
</select>
```

The resulting Save The Child page will look the same as the last sample from the previous chapter, but the Countries and States dropdowns are now populated by the data located in these files (later in this chapter in the section on JSON we'll replace this HTML file with its JSON version). These are the first three lines (out of 241) from the file countries.html:

```
<option value="United States">United States</option>
<option value="United Kingdom">United Kingdom</option>
<option value="Afghanistan">Afghanistan</option>
```

The JavaScript code that reads countries and states from file (text and HTML markup) and populates the dropdowns comes next. The content of these files is assigned to the innerHTML attribute of the given HTML <select> element.

```
function loadData(dataUrl, target) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', dataUrl, true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            if((xhr.status >= 200 && xhr.status < 300) ||
               xhr.status==304){
                target.innerHTML += xhr.responseText;
            } else {
                console.log(xhr.statusText);
            }
        }
    }
    xhr.send();
}

// Load the countries and states using XHR
loadData('data/us-states.html', statesList);
loadData('data/countries.html', countriesList);
```



The above code has an issue, which may not be so obvious, but can irritate users. The problem is that it doesn't handle errors. Yes, we print the error message on the developer's console, but the end user will never see them. If for some reason the data about countries or states won't arrive, the dropdowns will be empty, the donation form won't be valid and the users will become angry that they can't make a donation without knowing why. Proper error handling and reports are very important for any application so never ignore it. You should display a user-friendly error messages on the Web page. For example the above else statement can display the received message in the page footer

```

else {
    console.log(xhr.statusText);

    // Show the error message on the Web page
    footerContainer.innerHTML += '<p class="error">Error getting ' +
        target.name + ": " + xhr.statusText + ",code: " +
        xhr.status + "</p>';
}

```

This code uses the CSS selector `error` that will show the error message on the red background. you can find it in the file `styles.css` in the project-02-donation-error-ajax.html. It looks like this:

```

footer p.error {
    background:#d53630;
    text-align:left;
    padding: 0.9em;
    color: #fff;
}

```

The following code fragment shows how to add the received data to a certain area on the Web page. This code creates an HTML paragraph `<p>` with the text returned by the server and then adds this paragraph to the `<div>` with the ID `main`:

```

if (xhr.readyState == 4) {

    // All status codes between 200 and 300 mean success
    // and 304 means Not Modified
    if((xhr.status >=200 && xhr.status <300) || xhr.status==304){
        var p = document.createElement("p");

        p.appendChild(document.createTextNode(myRequest.responseText));

        document.getElementById("main").appendChild(p);
    }
}

```

## Using JSON

In any client-server application one of the important decisions to be made is about the format of the data that go over the network. We are talking about the application-specific data. Someone has to decide how to represent the data about an Auction Item, Customer, Donation et al. The easiest way to represent text data is Comma Separated Format (CSV), but it's not easily readable by humans, hard to validate, and recreation of JavaScript objects from CSV feed would require additional information about the headers of the data.

Sending the data in XML form addresses the readability and validation issues, but it's very verbose. Every data element has to be surrounded by an opening and closing tag describing the data. Converting the XML data to/from JavaScript object requires special

parsers, and you'd need to use one of the JavaScript libraries for cross-browser compatibility.

Douglas Crockford popularized new data format, which is known as JSON. In today's Web JSON became the most popular data format. It's not as verbose as XML, and JSON's notation is almost the same as JavaScript object literals. It's easily readable by humans, and every ECMAScript 5 compliant browser includes a native JSON object: `window.JSON`. Even though the JSON formatted data look like JavaScript object literals, JSON is language independent. Here's an example of the data in the JSON format:

```
{  
  "fname": "Alex",  
  "lname": "Smith",  
  "age": 30,  
  "address": {  
    "street": "123 Main St.",  
    "city": "New York"}  
}
```

Anyone who knows JavaScript understands that this is an object that represents a person, which has a nested object that represents an address. Note the difference with JavaScript literals: the names of the properties are always strings, and every string must be taken into quotes. Representing the same object in XML would need a lot more characters (e.g. `<fname>Alex</fname>` etc).

There are some other important differences between JSON and XML. The structure of the XML document can be defined using DTD or XML Schema, which simplifies the data validation, but requires additional programming and schema maintenance. On the other hand, JSON data have data types, for example the `age` attribute in the above example is not only a `Number`, but will be further evaluated by the JavaScript engine and will be stored as an integer. JSON also supports arrays while XML doesn't.

For parsing JSON in JavaScript you use the method `JSON.parse()`, which takes a string and returns JavaScript object, for example:

```
var customer=JSON.parse('{"fname": "Alex", "lname": "Smith"}');  
  
console.log("Your name is " + customer.fname + " " + customer.lname);
```

For a reverse operation - turning an object into JSON string - do `JSON.stringify(customer)`. The older browsers didn't have the `JSON` object, and there is an alternative way of parsing JSON is with the help of the script `json2.js`, which creates the `JSON` property on the global object. The `json2.js` is freely available on [Github](#). In Chapter 3 you've learned about feature detection with Modernizr, and you can automate the loading of this script if needed.

```
Modernizr.load({  
  test: window.JSON,  
  nope: 'json2.js',
```

```

complete: function () {
    var customer = JSON.parse('{"fname":"Alex","lname":"Smith"}');
}
});

```

Usually, JSON-related articles and blogs are quick to remind you about the evil nature of the JavaScript function `eval()` that can take an arbitrary JavaScript code and execute it. The `JSON.parse()` is pictured as a protection against the malicious JavaScript that can be injected into your application's code and then executed by `eval()` by the Web browser. The main argument is that `JSON.parse()` will not be processing the incoming code unless it contains valid JSON data.

Protecting your application code from being infected by means of `eval()` can be done outside of your application code. Replacing HTTP with secure HTTPS protocol helps a lot in this regard. Some Web applications eliminate the possibility of cross-origin scripting by routing all requests to third-party data sources via proxying such requests through your trusted servers. But proxying all requests through your server may present scalability issues - imagine if thousands of concurrent users will be routed through your server - so do some serious load testing before making this architectural decision.



There are several JSON tools useful for developers. To make sure that your JSON data is valid and properly formatted use [JSONLint](#). If you paste an ugly one-line JSON data JSONLint will reformat it into a readable form. There is also an add-on JSONView, available both for [Firefox](#) and for [Chrome](#) browsers. With JSONView the JSON objects are displayed in a pretty formatted collapsible format. If there are errors in the JSON document they will be reported. At the time of this writing Chrome's version of JSONView does a better job in reporting errors.

## Populating States and Countries from JSON Files

Earlier in this chapter you've seen an example of populating states and countries in the donate form from HTML files. Now you'll see how to retrieve the JSON data by making an AJAX call. Open in the Web browser the project-04-2-donation-ajax-json - it reads the countries and states from the files `countries.json` and `us_states.json` respectively. The beginning of the file `countries.json` is shown below:

```
{
  "countrieslist": [
    {
      "name": "Afghanistan",
      "code": "AF"
    },
    {
      "name": "Åland Islands",
      "code": "AX"
    },
    {

```

```

        "name": "Albania",
        "code": "AL"
    },

```

The JavaScript code that populates the countries and states comboboxes comes next. Note the difference in creating the `<option>` tags from JSON vs. HTML. In case of HTML, the received data were added to the `<select>` element as is: `target.innerHTML += xhr.responseText;` In JSON files the data were not wrapped in to the `<option>` tags, hence it's done programmatically.

```

function loadData(dataUrl, rootElement, target) {
    var xhr = new XMLHttpRequest();
    xhr.overrideMimeType("application/json");
    xhr.open('GET', dataUrl, true);

    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            if (xhr.status == 200) {

                //parse json data
                var jsonData = JSON.parse(xhr.responseText);

                var optionsHTML = ''
                for(var i= 0; i < jsonData[rootElement].length; i++){
                    optionsHTML+=""+jsonData[rootElement][i].name+""
                }

                var targetCurrentHtml = target.innerHTML;
                target.innerHTML = targetCurrentHtml + optionsHTML;

            } else {
                console.log(xhr.statusText);

                // Show the error on the Web page
                tempContainer.innerHTML += '<p class="error">Error getting ' +
                    target.name + ": " + xhr.statusText + ",code: " + xhr.status + "</p>";
            }
        }
    }
    xhr.send();
}

loadData('data/us-states.json', 'usstateslist', statesList);
loadData('data/countries.json', 'countrieslist', countriesList);

```

In the above example we called the method `XMLHttpRequest.overrideMimeType()` to ensure that the data will be treated by the browser as JSON even if the server won't report them as such.

## Arrays in JSON

JSON supports arrays, and the next example shows you how the information about a customer can be presented in JSON format. A customer can have more than one phone, which are stored in an array.

```
<script>
  var customerJson = '{"fname":"Alex",
                        "lname":"Smith",
                        "phones":[
                            "212-555-1212",
                            "565-493-0909"
                        ]
  }';

  var customer=JSON.parse(customerJson);

  console.log("Parsed customer data: fname=" + customer.fname +
              " lname=" + customer.lname +
              " home phone=" + customer.phones[0] +
              " cell phone=" + customer.phones[1]);
</script>
```

The code above creates an instance of the JavaScript object referenced by the variable `customer`. In this example the `phones` array just holds two strings. But you can store object in JSON array the same way as you'd do it in JavaScript object literal - just don't forget to put every property name in quotes.

```
var customerJson = '{"fname":"Alex",
                      "lname":"Smith",
                      "phones":[
                          {"type":"home", "number":"212-555-1212"},
                          {"type":"work", "number":"565-493-0909"}]
  }';
```

## Loading Charity Events using AJAX and JSON

The last example in Chapter 1 was about displaying various charity events on the Google map using multiple markers. But the data about these events were hard-coded in HTML file. After getting familiar with AJAX and JSON it should not be too difficult to create a separate file with the information about charities in JSON format and load them using `XMLHttpRequest` object.

The next version of Save The Child is a modified version of the application that displayed Google map with multiple markers from Chapter 1. But this time we'll load the information about the charity events from the file `campaigndata.json` shown next.

```
{
  "campaigns": {
    "header": "Nationwide Charity Events",
```

```

"timestamp": "12/15/2012",
"items": [
  {
    "title": "Lawyers for Children",
    "description": "Lawyers offering free services for The Children",
    "location": "New York,NY"
  },
  {
    "title": "Mothers of Asthmatics",
    "description": "Mothers of Asthmatics - nationwide Asthma network",
    "location": "Dallas,TX"
  },
  {
    "title": "Friends of Blind Kids",
    "description": "Semi-annual charity events for blind kids",
    "location": "Miami,FL"
  },
  {
    "title": "A Place Called Home",
    "description": "Adoption of The Children",
    "location": "Miami,FL"
  },
  {
    "title": "Marathon for Survivors",
    "description": "Annual marathon for cancer survivors",
    "location": "Fargo, ND"
  }
]
}

```

Run the project-03-maps-json-data and you'll see the map with the markers for each of the events loaded from the file campaigndata.json (see [Figure 2-3](#)). Click on the marker to see an overlay with the event details.

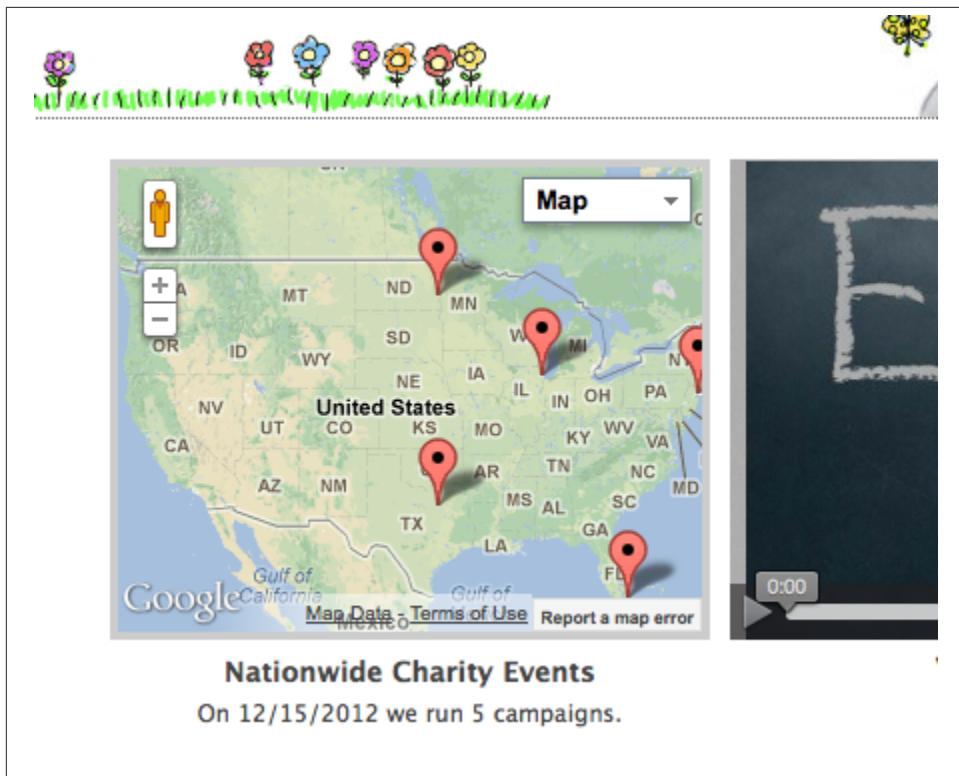


Figure 2-3. Markers built from JSON data

Note that this JSON file contains the object `campaigns`, which includes the array of objects `items` representing charity events. XMLHttpRequest object loads the data and the JSON parses it assigning the `campaigns` object to the variable `campaignsData` that is used in `showCampaignsInfo()` with Google Maps API (we've omitted the mapping part for brevity).

```
function showCampaignsInfo(campaigns) {  
    campaignsCount = campaigns.items.length;  
  
    var message = "<h3>" + campaigns.header + "</h3>" +  
        "On " + campaigns.timestamp +  
        " we run " + campaignsCount + " campaigns.";  
  
    locationUI.innerHTML = message + locationUI.innerHTML;  
    resizeMapLink.style.visibility = "visible";  
  
    createCampaignsMap(campaigns);  
}
```

```

function loadCampaignsData(dataUrl) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', dataUrl);

    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            if ((xhr.status >= 200 && xhr.status < 300) ||
                xhr.status === 304) {
                var jsonData = xhr.responseText;

                var campaignsData = JSON.parse(jsonData).campaigns;
                showCampaignsInfo(campaignsData);
            } else {
                console.log(xhr.statusText);

                tempContainer.innerHTML += '<p class="error">Error getting ' +
                    target.name + ": " + xhr.statusText +
                    ", code: " + xhr.status + "</p>";
            }
        }
    }
    xhr.send();
}

var dataUrl = 'data/campaignsdata.json';
loadCampaignsData(dataUrl);

```



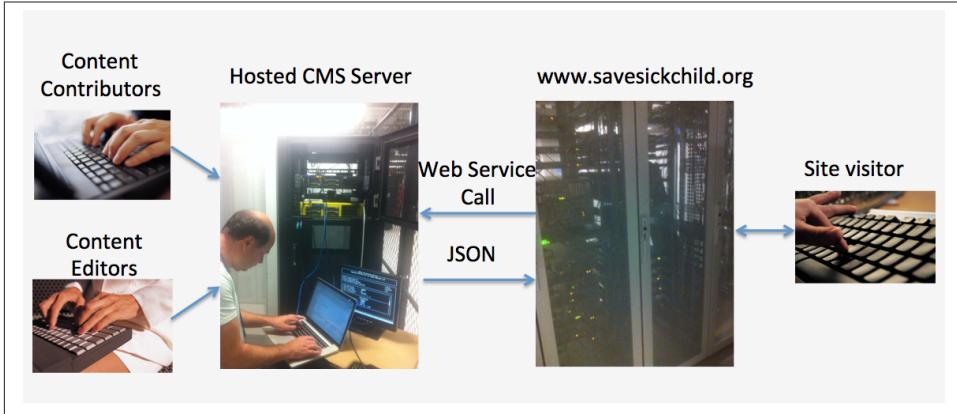
Some older Web browsers may bring up a File Download popup window when the content type of the server's response is set to "application/json". Try to use the MIME type "text/html" instead, if you ran into this issue.



For simplicity, in this section we've been loading JSON formatted data from files, but in the real-world applications the JSON data is created on the server dynamically. For example, a browser makes a RESTFull call to the Java-based server, which queries a database, generates JSON formated result and sends it back to the Web server.

## JSON in CMS

Large-scale Web applications could be integrated with some Content Management Systems (CMS), which can be supplying content such as charity events, sales promotions, et al. CMS servers can be introduced into the architecture of a Web application to separate the work on preparing the content from the application delivering it as shown in **Figure 2-4** depicting a diagram of a with a Web application integrated with the CMS server.



*Figure 2-4. CMS in the picture*

The content contributors and editors prepare the information on the charities and donation campaigns using a separate application, not the Save The Child page. The CMS server and the Web application server [www.savescickchild.org](http://www.savescickchild.org) may be located in the same or separate data centers. The server-side code of the Save The Child is making a call to a CMS server whenever the site visitor is requesting the information about charity events. If you get to pick a CMS for your future Web application make sure it offers data feed in JSON format.

Some time ago one of the authors of this book were helping Mercedes Benz USA in development of their consumer facing Web application where people could search, review and configure their next car. [Figure 2-5](#) shows a snapshot taken from the [mbusa.com](http://mbusa.com). Three rectangular areas at the bottom were created by the Web designers to display today's deals and promotions. The up-to-date content for these areas (in a JSON format) was retrieved from a CMS server when the user visited [mbusa.com](http://mbusa.com).

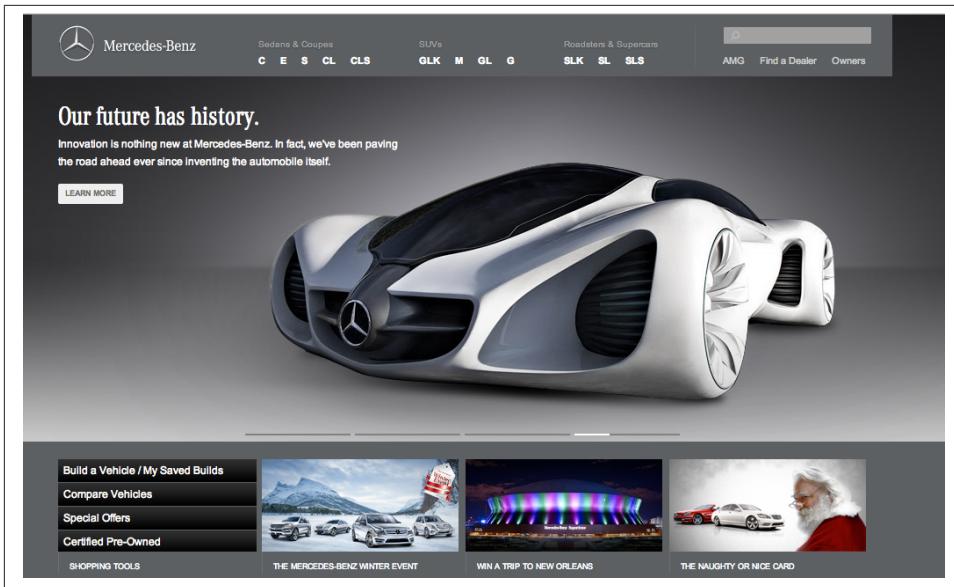


Figure 2-5. Current Mercedes deals from CMS



There's a side benefit of learning JSON - it's used as data format in NoSQL databases like [MongoDB](#).

## JSON in Java

If a Web browser receives JSON stream from the server the application needs to turn it into JavaScript objects. If a Web client needs to send the JavaScript objects to the server they can be converted into JSON string. Similar tasks have to be performed on the server side. Our Save The Child application uses Java application server. There is a number of third-party Java libraries that can consume and generate JSON content.

There are several Java libraries to convert Java objects into their JSON representation and back, for example [Google's Gson](#), [Jackson](#), [json-simple](#).

Google's Gson is probably the simplest one for use. It provides methods `toJson()` and `fromJson()` to convert Java objects to JSON and back. Gson allows pre-existing unmodifiable objects to be converted to and from JSON and Supports Java Generics. Gson works well with complex objects with deep inheritance hierarchies.

Let's say JavaScript sends to Java the following JSON string:

```
{"fname": "Alex", "lname": "Smith", "skillLevel": 11}
```

The Java code can turn it into an instance of the Customer object by calling the method `Gson.fromJson()`. Similarly, Java code can create a JSON string from an object instance. Both of these operations are illustrated below.

```
public Customer createCustomerFromJson(String jsonString){  
  
    Gson myGson = new Gson();  
    Customer cust = myGson.fromJson(jsonString, Customer.class);  
    return cust;  
}  
  
public String createJsonFromCustomer(Customer cust){  
  
    Gson gson = new Gson();  
  
    return gson.toJson(cust, Customer.class);  
}
```

Of course, the declaration of the Java class `Customer` must exist in the in the classpath and don't forget to include `gson.jar` to your Java project.

JSON data format is often used in non-JavaScript applications. For example, a Java server can exchange the JSON-formatted data with a .Net server.



The Java EE 7 specification includes JSR 353, which defines a standarized way for parsing and generating JSON. JSR 353 defines the Java API from JSON Processing (JSON-P) that shouldn't be confused with another acronym **JSONP or JSON-P**, which is JSON with Padding (we'll discuss it at the end of this chapter).

## Compressing JSON

JSON format is more compact than XML and is readable by the human beings. But when you are ready to deploy your application in production, you still want to compress the data so less bytes will travel over the wire to the user's browser. The server-side libraries that generate JSON will make the data sent to the client compact by removing the tab and the new line characters.

If you want to turn the pretty-print JSON into a more compact one-line format just use such Web sites as **JavaScript Compressor** or **JSON Formatter**. For example, after running the 12Kb file `countries.json` through this compressor, its size was decreased to 9Kb. JSONLint can also compress JSON if you provide this URL: <http://jsonlint.com?reformat=compress>.

Similarly to most of the content that is being sent to browsers by the Web servers, the JSON data should be compressed. **GZip** and **Deflate** are the two main compression

methods used in today's Web. Both use the same compression algorithm *deflate*, but while with Deflate the compressed data are being streamed to the client, the GZip first compresses the entire file, calculates the size and adds some additional headers to the compressed data. So GZip may need some extra time and memory, but you are more protected from getting incomplete JSON, JavaScript or other content. Both Gzip and Deflate are easily configurable by major Web servers, but it's hard to say which one is better for your application - set up some tests with each of them and decide which one works faster or take less system resources, but don't compromise on reliability of the compressed content.

We prefer using GZip, which stands for GNU zip compression. On the server side you'd need to configure the gzip filters on your Web server. You need to refer to your Web server's documentation to find out how to configure gzipping, which is done by the MIME type. For example, you can request to gzip everything except images (you might want to do this if you're not sure if all browsers can properly uncompress certain MIME types).

For example, applying the GZip filter to the 9Kb countries.json will reduce its size to 3Kb, which means serious bandwidth savings especially in the Web applications with lots of concurrent users. This is even more important for the mobile Web clients, which may be operating in the areas with slower connections. Web clients usually set the HTTP Request attribute `Accept-Encoding: gzip` inviting the server to return gzipped content, and the Web server may compress the response if it does support it or unzipped content otherwise. If the server supports gzip, the HTTP response will have the attribute `Content-Encoding: gzip`, and the browser will know to unzip the response data before use.

Gzip is being used for compressing all types of content: HTML, CSS, JavaScript and more. If your server sends JSON content to the client setting the content type to `application/json` don't forget to include this MIME type in your server configuration for Gzip.

Web browsers support the gzipping too, and your application can set `Content-Encoding: gzip` in HTTP request while sending the data from the Web client to the server. But Web clients usually don't send massive amounts of data to the server so the benefits of the compression on the client side may not be as big.

## Adding Charts to Save The Child

Let's consider yet another use case for JSON in Save The Child. We want to display charts with statistics about the donations. By now, our application look not exactly as the original mockup from [Figure 1-2](#), but it's pretty close. There is an empty space in the left to the maps, and the charts showing donation statistics can fit right in. Now we need to decide how to draw the charts using nothing, but HTML5 elements. Note that we are

not talking about displaying static images using the `<img>` element - the goal is to draw the images dynamically in the client's code. You can accomplish this goal using HTML5 elements `<canvas>` or `<svg>`.

The `<canvas>` element provides a bitmap canvas, where your scripts which can draw graphs, game graphics, or other visual images on the fly without using any plugins like Flash Player or Silverlight. To put it simple, the `<canvas>` defines a rectangular area that consists of pixels, where you can draw. Keep in mind that the DOM object can't peek inside the canvas and access specific pixels. So if you are planning to create an area with dynamically changed graphics you might want to consider using `<svg>`.

The `<svg>` element supports **Scalable Vector Graphics (SVG)**, which is the XML-based language for describing two-dimensional graphics. Your code has to provide commands to draw the lines, text, images et al.

## Adding Chart With Canvas Element

Let's review some code fragments from project-04-canvas-pie-chart-json. The HTML section defines `<canvas>` of 260x240 pixels. If the user's browser doesn't support `<canvas>`, the user won't see the chart, but will see the text "Your browser does not support HTML5 Canvas" instead. You need to give an ID to your canvas element so your JavaScript code can access it.

```
<div id="charts-container">
    <canvas id="canvas" width="260" height="240">
        Your browser does not support HTML5 Canvas
    </canvas>
    <h3>Donation Stats</h3>
    <p> Lorem ipsum dolor sit amet, consectetur</p>
</div>
```

Run the project-04-canvas-pie-chart-json, and you'll see the chart with donation statistics by city as in [Figure 2-6](#). We haven't style our `<canvas>` element, but we could've added a background color, the border and other bells and whistles if required.

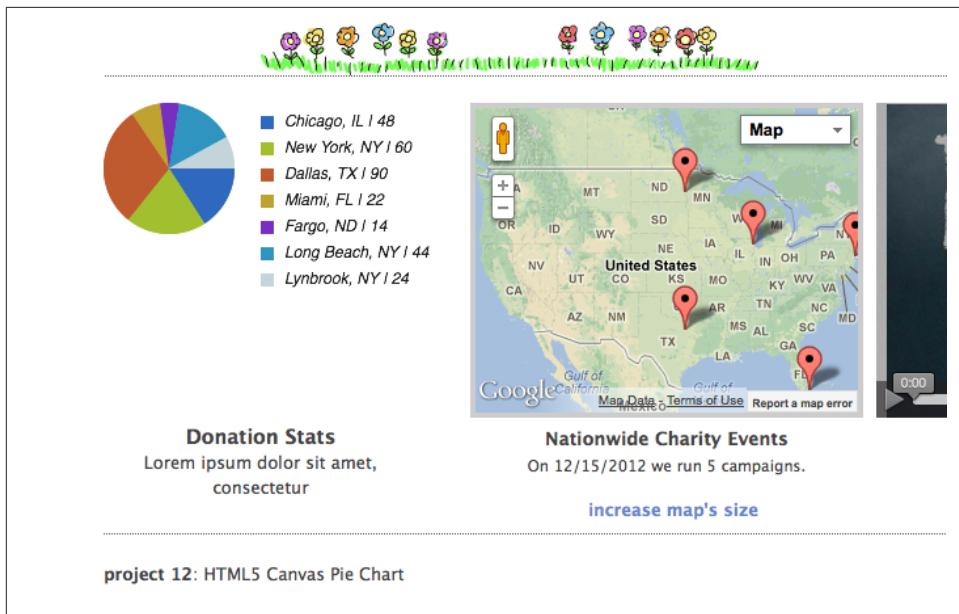


Figure 2-6. Adding a chart

The data to be used for drawing a pie chart in our canvas are stored in the file data/chartdata.json, but in a real-world the server side code can generate it based on the up-to-the-second donation data and send it to the client. For example, you could do it as was explained in the section Json in Java above. This is the content of our file chartdata.json:

```
{
  "ChartData": {
    "items": [
      {
        "donors": 48,
        "location": "Chicago, IL"
      },
      {
        "donors": 60,
        "location": "New York, NY"
      },
      {
        "donors": 90,
        "location": "Dallas, TX"
      },
      {
        "donors": 22,
        "location": "Miami, FL"
      }
    ]
  }
}
```

```

        "donors": 14,
        "location":"Fargo, ND"
    },
{
    "donors": 44,
    "location":"Long Beach, NY"
},
{
    "donors": 24,
    "location":"Lynbrook, NY"
}
]
}
}

```

Loading of the the chardata.json is done using AJAX techniques as explained earlier. Although in our example we're loading the chart immediately when the Save The Child loads, the following code could be invoked only when the user requests to see the charts by clicking on some menu item on the page.

```

function getChartData(dataUrl, canvas) {
    var xhr = new XMLHttpRequest();
    xhr.open('GET', dataUrl, true);

    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            if ((xhr.status >= 200 && xhr.status < 300) ||
                xhr.status === 304) {
                var jsonData = xhr.responseText;

                var chartData = JSON.parse(jsonData).ChartData;      // ①

                drawPieChart(canvas, chartData, 50, 50, 49);      // ②

            } else {
                console.log(xhr.statusText);
                tempContainer.innerHTML += '<p class="error">Error getting ' +
                    target.name + ": " + xhr.statusText +
                    ',code:' + xhr.status + "</p>";
            }
        }
        xhr.send();
    }

    loadData('data/chardata.json', document.getElementById("canvas"));
}

```

- ① Parse JSON and create the ChartData Javascript object.

- ② Pass the data to the `drawPieChart()` function that will draw the pie in the `canvas` element with the center coordinates  $x=50$  and  $y=50$  pixels. The top left corner of the canvas has coordinates  $(0,0)$ . The radius of the pie will be 49 pixels. The code of the function that draws the pie on the canvas goes next.

```
function drawPieChart (canvas, chartData, centerX, centerY, pieRadius) {
    var ctx; // The context of canvas
    var previousStop = 0; // The end position of the slice
    var totalDonors = 0;

    var totalCities = chartData.items.length;

    // Count total donors
    for (var i = 0; i < totalCities; i++) {
        totalDonors += chartData.items[i].donors; // ❶
    }

    ctx = canvas.getContext("2d"); // ❷
    ctx.clearRect(0, 0, canvas.width, canvas.height);

    var colorScheme = ["#2F69BF", "#A2BF2F", "#BF5A2F", // ❸
                      "#BFA22F", "#772FBF", "#2F94BF", "#c3d4db"];
    for (var i = 0; i < totalCities; i++) { // ❹

        //draw the sector
        ctx.fillStyle = colorScheme[i];
        ctx.beginPath();
        ctx.moveTo(centerX, centerY);
        ctx.arc(centerX, centerY, pieRadius, previousStop, previousStop +
               (Math.PI * 2 * (chartData.items[i].donors/totalDonors)),false);
        ctx.lineTo(centerX, centerY);
        ctx.fill();

        // label's bullet
        var labelY = 20 * i + 10;
        var labelX = pieRadius*2 + 20;

        ctx.rect(labelX, labelY, 10, 10);
        ctx.fillStyle = colorScheme[i];
        ctx.fill();

        // label's text
        ctx.font = "italic 12px sans-serif";
        ctx.fillStyle = "#222";
        var txt = chartData.items[i].location + " | " +
                  chartData.items[i].donors;
        ctx.fillText (txt, labelX + 18, labelY + 8);

        previousStop += Math.PI * 2 * (chartData.items[i].donors/totalDonors);
    }
}
```

```
        }  
    }
```

- ➊ Count the total number of donors.
- ➋ Get the 2D context of the <canvas> element. This is the most crucial element to know for drawing on a canvas.
- ➌ The color scheme is just a set of colors to be used for painting each slice (sector) of the pie.
- ➍ The for-loop paints one sector on each iteration. This code draws lines, arcs, rectangles, and adds text to the canvas. Describing the details of each method of the context object is out of scope of this book, but you can find the details of the context API in the [W3C documentation](#) available online.



To minimize the amount of manual coding, consider using one of the JavaScript libraries that helps with visualization (e.g. [D3.js](#)).

## Adding Chart With SVG

What if we want to make this chart dynamic and reflect the changes in donations every 5 minutes? If you're using <canvas>, you'll need to redraw each and every pixel of our canvas with the pie. With SVG, each element of the drawing would be the DOM element so we could have redraw only those elements that have changed. If with canvas your script draws using pixels, the SVG drawings are done with vectors.

To implement the same donation statistics pie with the <svg> element, you'd need to replace the <canvas> element with the following markup:

```
<div id="charts-container">  
    <svg id="svg-container" xmlns="http://www.w3.org/2000/svg">  
  
        </svg>  
        <h3>Donation Stats</h3>  
        <p>  
            Lorem ipsum dolor sit amet, consectetur  
        </p>  
    </div>
```

Running the project-05-svg-pie-chart-json would show you pretty much the same pie as it uses the file chartdata.json with the same content, but the pie was produced differently. The code of the new version of the `drawPieChart()` is shown below. We won't discuss all the details of the drawing with SVG, but will highlight a couple of important lines of code that illustrate the difference between drawing on <canvas> vs. <svg>.

```

function drawPieChart(chartContainer, chartData, centerX, centerY,
                      pieRadius, chartLegendX, chartLegendY) {
    // the XML namespace for svg elements
    var namespace = "http://www.w3.org/2000/svg";
    var colorScheme = ["#2F69BF", "#A2BF2F", "#BF5A2F", "#BFA22F",
                      "#772FBF", "#2F94BF", "#c3d4db"];

    var totalCities = chartData.items.length;
    var totalDonors = 0;

    // Count total donors
    for (var i = 0; i < totalCities; i++) {
        totalDonors += chartData.items[i].donors;
    }

    // Draw pie sectors
    startAngle = 0;
    for (var i = 0; i < totalCities; i++) {
        // End of the sector = starting angle + sector size
        var endAngle = startAngle + chartData.items[i].donors / totalDonors * Math.PI * 2;
        var x1 = centerX + pieRadius * Math.sin(startAngle);
        var y1 = centerY - pieRadius * Math.cos(startAngle);
        var x2 = centerX + pieRadius * Math.sin(endAngle);
        var y2 = centerY - pieRadius * Math.cos(endAngle);

        // This is a flag for angles larger than a half circle
        // It is required by the SVG arc drawing component
        var big = 0;
        if (endAngle - startAngle > Math.PI) {
            big = 1;
        }

        //Create the <svg:path> element
        var path = document.createElementNS(namespace, "path"); // ①

        // Start at circle center
        var pathDetails = "M " + centerX + "," + centerY + // ②
                        " L " + x1 + "," + y1 +
                        " A " + pieRadius + "," + pieRadius +
                        " 0 " + big + " 1 " +
                        " Z";
        // Close the path at (centerX, centerY)

        // Attributes for the <svg:path> element
        path.setAttribute("d", pathDetails);
        // Sector fill color
        path.setAttribute("fill", colorScheme[i]);
    }
}

```

```

chartContainer.appendChild(path); // ③

// The next sector begins where this one ends
startAngle = endAngle;

// label's bullet
var labelBullet = document.createElementNS(namespace, "rect");
// Bullet's position
labelBullet.setAttribute("x", chartLegendX);
labelBullet.setAttribute("y", chartLegendY + 20 * i);

// Bullet's size
labelBullet.setAttribute("width", 10);
labelBullet.setAttribute("height", 10);
labelBullet.setAttribute("fill", colorScheme[i]);

chartContainer.appendChild(labelBullet); // ④

// Add the label text
var labelText = document.createElementNS(namespace, "text");

// label position = bullet's width(10px) + padding(8px)
labelText.setAttribute("x", chartLegendX + 18);
labelText.setAttribute("y", chartLegendY + 20 * i + 10);
var txt = document.createTextNode(chartData.items[i].location +
" | " + chartData.items[i].donors);

labelText.appendChild(txt);
chartContainer.appendChild(labelText); // ⑤
}

}

```

- ➊ Create the `<svg:path>` HTML element, which is the most important SVG element for drawing basic shapes.. It includes a series of commands that produce the required drawing. For example, `M 10 10` means *move to the coordinate 10,10* and `L 20 30` means *draw the line to the coordinate 20,30*.
- ➋ Fill the details of the `<svg:path>` element to draw the pie sector. Run the project-05-svg-pie-chart-json to see the Save The Child page, then right-click on the pie chart and select Inspect Element (this is the name of the menu item in Firefox). **Figure 2-7** shows the resulting content of our `<svg>` element. As you can see, it's not pixel based but a set of XML-like commands that drew the content of the chart. If you'll run the previous version of our application (project-04-canvas-pie-chart-json) and right-click on the chart, you will be able to save it as an image, but won't see the internals of the `<canvas>` element.

- ③ ④ Adding the internal elements of the chart container to the DOM - path, bullets  
 ⑤ and text. These elements can be modified if needed without redrawing the entire content of the container.



in our code example we have written the path commands manually to process the data dynamically. But Web designers often use tools ([Adobe Illustrator](#), [Incscape](#) et al.) to draw and then export images into an SVG format. In this case all paths will be encoded as <svg:path> automatically.

**Firebug – Save Sick Child**

Console    HTML    CSS    Script    DOM    Net    Cookies    Illuminations    cssUpdater

Edit    Sync now    path < svg#svg-container < div#charts-container < section#...-section < div#main <

```
<div id="charts-container">
  <svg id="svg-container" xmlns="http://www.w3.org/2000/svg">
    <path d="M 50,52 L 50,3 A 49,49 0 0 1 91.19635194393138,25.469628979003286 Z" fill="#2F69BF">
    <rect x="115" y="10" width="10" height="10" fill="#2F69BF">
    <text x="133" y="20">Chicago, IL | 48</text>
    <path d="M 50,52 L 91.19635194393138,25.469628979003286 A 49,49 0 0 1 88.21877053797101,82.66472857479614 Z" fill="#A2BF2F">
    <rect x="115" y="30" width="10" height="10" fill="#A2BF2F">
    <text x="133" y="40">New York, NY | 60</text>
    <path d="M 50,52 L 88.21877053797101,82.66472857479614 A 49,49 0 0 1 9.36449499269846,79.3816678235927 Z" fill="#BF5A2F">
    <rect x="115" y="50" width="10" height="10" fill="#BF5A2F">
    <text x="133" y="60">Dallas, TX | 90</text>
    <path d="M 50,52 L 9.36449499269846,79.3816678235927 A 49,49 0 0 1 1.447380111154402,58.60629600943014 Z" fill="#BFA22F">
    <rect x="115" y="70" width="10" height="10" fill="#BFA22F">
    <text x="133" y="80">Miami, FL | 22</text>
    <path d="M 50,52 L 1.447380111154402,58.60629600943014 A 49,49 0 0 1 1.5953235862052395,44.385060651861934 Z" fill="#772FBF">
    <rect x="115" y="90" width="10" height="10" fill="#772FBF">
    <text x="133" y="100">Fargo, ND | 14</text>
    <path d="M 50,52 L 1.5953235862052395,44.385060651861934 A 49,49 0 0 1 26.53713764295629,8.982630368484621 Z" fill="#2F94BF">
    <rect x="115" y="110" width="10" height="10" fill="#2F94BF">
    <text x="133" y="120">Long Beach, NY | 44</text>
    <path d="M 50,52 L 26.53713764295629,8.982630368484621 A 49,49 0 0 1 50.0000000000003,3 Z" fill="#c3d4db">
    <rect x="115" y="130" width="10" height="10" fill="#c3d4db">
    <text x="133" y="140">Lynbrook, NY | 24</text>
  </svg>
  <h3>Donation Stats</h3>
  <p>Lorem ipsum dolor sit amet, consectetur</p>
</div>
</section>
</div>
```

Figure 2-7. The chart content in SVG

Since the SVG is XML-based, it's very easy to generate the code shown in [Figure 2-7](#) on the server, and lots of Web applications send ready to display SVG graphics to the users' Web browsers. But in our example we are generating the SVG output in the JavaScript from JSON received from the server, which provides a cleaner separation between the client and the server-side code. The final decision on what to send to the Web browser (ready to render SVG or raw JSON) has to be made after considering various factors such as available bandwidth, the size of data, the number of users, the existing load on the server resources.



SVG supports animations and transformation effects, while canvas doesn't.

## Loading Data From Other Servers With JSONP

Imagine that a Web page was loaded from the domain abc.com, and it needs JSON-formatted data from another domain (xyz.com). As mentioned earlier, AJAX has cross-origin restrictions, which prevent this. JSONP is a technique used to relax the cross-origin restrictions. With JSONP, instead of sending plain JSON data, the server wraps them up into a JavaScript function and then sends it to the Web browser for execution as a callback. The Web page that was originated from abc.com may send the request `http://xyz.com?callback=myDataHandler` technically requesting the server xyz.com to invoke the JavaScript callback named `myDataHandler`. This URL is a regular HTTP GET request, which may have other parameters too so you can send some data to the server too.

The server will send to the browser the JavaScript function that may look as follows:

```
function myDataHandler({"fname": "Alex", "lname": "Smith", "skillLevel": 11});
```

The Web browser will invoke the callback `myDataHandler()`, which must exist in the Web page. The Web browser will pass the received JSON object as an argument to this callback:

```
function myDataHandler(data){  
    // process the content of the argument data - the JSON object  
    // received from xyz.com  
}
```

If all you need is just to retrieve the data from a different domain on page just add the following tag to your HTML page:

```
<script src="http://xyz.com?callback=myDataHandler">
```

But what if you need to dynamically make such requests periodically (e.g. get all twits with a hashtag #savisickchild by sending an HTTP GET using Twitter API at `http://search.twitter.com/search.json?q=savesickchild&rpp=5&include_entities=true&with_twitter_user_id=true&result_type=mixed`)? You add a change handler to the option that is called and passes or grabs the value needed.

You can dynamically add a `<script>` tag to the DOM object from your JavaScript code. Whenever the browser sees the new `<script>` element it executes it. Such script injection can be done like this:

```
var myScriptTag = document.createElement("script");
myScriptTag.src = "http://xyz.com?callback=myDataHandler";
document.getElementsByTagName("body").appendChild(myScriptTag);
```

Your JavaScript can build the URL for the `myScriptTag.src` dynamically and pass parameters to the server based on some user's actions.

Of course, this technique presents a danger if there is a chance that the JavaScript code sent by xyz.com is intercepted and replaced by a malicious code (similarly to the JavaScript `eval()` danger). But it's not more dangerous than receiving any JavaScript from non-trusted server. Besides, your handler function could always make sure that the received data is a valid object with expected properties, and only after that handle the data.

If you decide to use JSONP don't forget about error handling. Most likely you'll be using one of the JavaScript frameworks and they usually offer a standard mechanism for JSONP error handling, dealing with poorly formatted JSON responses, and recovery in cases of network failure. One of such libraries is called **jQuery-JSONP**.

## Beer and JSONP

In this section you'll see a small code example illustrating the data retrieval from publicly available **Open Beer DataBase**, which exists to help software developers test their code that makes REST Web service calls and works with JSON and JSONP data. Our Save The Child page won't display beer bottles, but we want to show that in addition to the retrieval of the donations and charts data from one domain we can get the data from a third-party domain openbeerdatabase.com.

First, enter the URL `http://api.openbeerdatabase.com/v1/breweries.json` in the address bar of your Web browser, and it'll return the following JSON data (only 2 out of 7 breweries are shown for brevity):

```
{
  "page": 1,
  "pages": 1,
  "total": 7,
  "breweries": [
    {
```

```

        "id": 1,
        "name": "(512) Brewing Company",
        "url": "http://512brewing.com",
        "created_at": "2010-12-07T02:53:38Z",
        "updated_at": "2010-12-07T02:53:38Z"
    },
    {
        "id": 2,
        "name": "21st Amendment Brewing",
        "url": "http://21st-amendment.com",
        "created_at": "2010-12-07T02:53:38Z",
        "updated_at": "2010-12-07T02:53:38Z"
    }
]
}

```

Now let's request the same data, but in a JSONP format by adding to the URL a parameter with a callback name `myDataHandler`. Entering in the browser `http://api.openbeerdatabase.com/v1/breweries.json?callback=processBeer` will return the following (it's a short version):

```

processBeer({"page":1,"pages":1,"total":7,"breweries":[{"id":1,"name":"(512) Brewing Company",
"url":"http://512brewing.com","created_at":"2010-12-07T02:53:38Z",
"updated_at":"2010-12-07T02:53:38Z"}, {"id":2,"name":"21st Amendment Brewing",
"url":"http://21st-amendment.com","created_at":"2010-12-07T02:53:38Z",
"updated_at":"2010-12-07T02:53:38Z"}]})

```

Since we haven't declared the function `processBeer()` yet, it won't be invoked. Let's fix it now. The function will check first if the received data contains the information about the breweries. If it does, the name of the very first brewery will be printed on the JavaScript console. Otherwise the console output will read "Retrieved data has no breweries info".

```

var processBeer=function (data){

    // Uncomment the next line to emulate malicious data
    // data="function evilFunction(){alert(' Bad function');}";

    if (data.breweries == undefined){
        console.log("Retrieved data has no breweries info.");
    } else{
        console.log("In the processBeer callback. The first brewery is "
            + data.breweries[0].name);
    }
}

var myScriptTag = document.createElement("script");
myScriptTag.src =
    "http://api.openbeerdatabase.com/v1/breweries.json?callback=processBeer";

var bd = document.getElementsByTagName('body')[0];
bd.appendChild(myScriptTag);

```

Figure 2-8 is a screen snapshot taken in the Firebug when it reached the breakpoint placed inside the processBeer callback on the `console.log`(in the `processBeer` call back"). You can see the content of the `data` argument - the beer has arrived.



The screenshot shows the Firebug JavaScript console with the 'data' object expanded. The 'data' object contains an array of brewery objects ('breweries') and some metadata ('page', 'pages', 'total'). Each brewery object has properties like 'name', 'url', 'id', 'created\_at', and 'updated\_at'. The 'page', 'pages', and 'total' properties are set to 1, 1, and 7 respectively. The 'arguments' and 'Closure Scope' sections are also visible at the bottom.

```
▶ this
  ▶ data
    ▶ breweries
      [ Object { page=1, pages=1, total=7, more... }
        [ Object { id=1, name="(512) Brewing Company", url="http://512brewing.com", more... }, Object { id=2, name="21st Amendment Brewing", url="http://21st-amendment.com", more... }, Object { id=3, name="Abbaye de Leffe", url="http://www.abbaye-de-leffe.be", more... }, 4 more... ]
      ▶ 0
        Object { id=1, name="(512) Brewing Company", url="http://512brewing.com", more... }
          created_at "2010-12-07T02:53:38Z"
          id 1
          name "(512) Brewing Company"
          updated_at "2010-12-07T02:53:38Z"
          url "http://512brewing.com"
      ▶ 1
        Object { id=2, name="21st Amendment Brewing", url="http://21st-amendment.com", more... }
      ▶ 2
        Object { id=3, name="Abbaye de Leffe", url="http://www.abbaye-de-leffe.be", more... }
      ▶ 3
        Object { id=4, name="Abita Brewing Company", url="http://abita.com", more... }
      ▶ 4
        Object { id=5, name="Alaskan Brewing Company", url="http://alaskanbeer.com", more... }
      ▶ 5
      ▶ 6
        Object { id=6, name="Ale Asylum", url="http://aleasylum.com", more... }
        Object { id=7, name="AleSmith", url="http://alesmith.com", more... }
      page 1
      pages 1
      total 7
    ▶ arguments
      [ Object { page=1, pages=1, total=7, more... } ]
    ▶ Closure Scope
      Closure Scope { toString=function() }
    ▶ Window
      Window index_beer.html
```

Figure 2-8. The beer has arrived

As a training exercise, try to replace the data retrieval from the beer Web service with adding the data feed from Twitter based on certain hash tags. See if you can find the place in the Save The Child Web page to display (and periodically update) this Twitter stream.



[json-generator.com](http://json-generator.com) is a handy Web site that can generate a file with a JSON or JSONP content based on your template. You can use this service to test AJAX queries - the generated JSON can be saved on this server to help in testing of your Web application.

## Summary

In this chapter you've learned about using AJAX as a means of communications of your Web browser with the servers. AJAX also deserves a credit for making the JavaScript language popular again by showing a practical way of creating single-page Web applications. Over the years JSON became the standard way of exchanging the data on the Web. The current version of the Save The Child application cleanly separates the code from the data, and you know how to update the content of the Web page without the need to re-retrieve the entire page from the server. In the next chapter you'll get familiar with the more productive way of developing Web applications with the library called jQuery.

## CHAPTER 3

# Introducing jQuery Library

Up till now we've been using HTML, plain JavaScript, and CSS to create the Save The Child Web application. In the real world, developers try to be more productive by using JavaScript libraries.

There are libraries like **jQuery Core** that substantially minimizes the amount of manual coding while programming core functionality of a Web application. The **jQuery UI** library offers widgets, animations, advanced effects. The **RequireJS** library is a module loader that allows to modularize HTML5 applications. There are also hundreds of micro libraries that can do just one thing and can be used ala cart (visit [microjs.com](http://microjs.com) for details).

Libraries are different from frameworks, which we discuss in Chapter 4. While frameworks force you to organize your code a certain way, a library simply offers your components that allow you to write less code.

This chapter is about the JavaScript library **jQuery** or to be more precise, JQuery Core. About a half of top Web sites use jQuery (visit <http://trends.builtwith.com/javascript/top> for the current statistics). jQuery is simple to use and doesn't require you to dramatically change the way you program for the Web. jQuery offers you a helping hand with the tasks that most of the Web developers need to deal with, for example, finding and manipulating DOM elements, processing browser's events, and dealing with browser incompatibility, which makes your code more maintainable. Since jQuery is an extensible library, lots and lots of plugins were created by developers from around the world, and all of them are available for free. If you can't find the plugin that fits your need you can create one yourself.



jQuery UI library is a “close relative” of jQuery Core. It’s a set of user interface interactions, effects, widgets, and themes built on top of the jQuery. You can find in jQuery UI such widgets as Datepicker, Accordion, Slider, Tabs, Autocomplete and more. jQuery UI will also help you with adding to your Web page various interactions (e.g. drag and drop) and effects, e.g. adding CSS classes or animations (the jQuery Core also has some effects). jQuery UI is built on top of jQuery Core library and can’t be used independently. jQuery UI is covered in the [O’Reilly book “jQuery UI”](#).



jQuery Mobile is yet another library built on top of jQuery Core. But this one is geared to creating mobile applications. We’ll cover it in detail in Chapter 11 of this book.

This chapter is not a detailed jQuery Core primer - there are jQuery books and [the online API Documentation](#) that provide a comprehensive explanation of jQuery. But we’ll give you just enough of the information to understand how jQuery can be used. In the section Working on Save The Child we’ll review the code of several versions of this application highlighting the benefits of using jQuery library.

## Getting Started With jQuery

At the time of this writing you can download either jQuery version 1.9 or jQuery 2.0 (the latter doesn’t support Internet Explorer 6, 7, and 8). You can download one of two distributions of jQuery library. The gzipped minified version of jQuery weighs 33Kb (it’s 93Kb if unzipped), and this is all you need unless you are planning to develop jQuery plugins, in which case get a larger development version - it’s about 270Kb. We’ve downloaded jQuery from [jquery.com](http://jquery.com) and included it in the `<script>` tag in our HTML code samples so you can run them even if the Internet connection is not available.

But instead of deploying jQuery framework on your servers as a part of your application, you should use a common Content Delivery Network (CDN) URL in your HTML as shown below. Since jQuery is an extremely popular library, many Web sites use it. If more than one Web page would get it from the same CDN, the Web browser would cache it locally and reuse it rather than downloading a separate copy from different servers for every Web application that uses jQuery. The [download page of jquery.com](#) offers three CDN: Google, Microsoft, and MediaTemple. For example, if you don’t need to use HTTPS protocol with your application the MediaTemple’s CDN suffice:

```
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
```



Using CDN may have another advantage - the content (jQuery in this case) is distributed around the globe and may be served to the user from the servers located in the same city/country thus reducing the latency.

You can provide a fallback URL by adding one extra line that will load jQuery from an alternative location if your CDN fails:

```
<script> window.jQuery || document.write('<script  
    src="http://code.jquery.com/jquery-1.9.1.min.js"></script>')  
</script>
```



You may find code samples that use the URL <http://code.jquery.com/jquery-latest.min.js> to download the latest version of jQuery. But keep in mind that by doing this you may run into a situation when some of the API of jQuery has been changed or deprecated. For example, jQuery 2.0 **stopped support** Internet Explorer 6, 7, and 8 and automatically switching to the latest version may result in malfunctioning of your application. We recommend using the specific version that has been tested with your application.

After covering the basics of jQuery Core, we are going to continue reviewing the code of a series of projects representing same Save The Child application, but this time using jQuery. Other than adding the validation to the Donate form and using an image slider this application remains the same in the previous chapter - we just want to show that the developers can be more productive in achieving the same result.

Some people will say that anyone who knows HTML can easily learn jQuery, but this is not so. Understanding of JavaScript is required (see Appendix A for reference). Programming with jQuery components starts with invoking the jQuery constructor named `jQuery()`. But people are using the shorter version of this constructor that's represented by a \$ sign: `$()`. This \$ property is the only object that jQuery will create in the global namespace. Everything else will be encapsulated inside the \$ object.



While it's easier to write `$()` than `jQuery()`, keep in mind that if you decide to use in your application another library in addition to jQuery, with the chances are higher to run into a conflict of having another `$` than another `jQuery` in the global name space. To make sure you won't find yourself in the "Another day, another `$`" position, put your code inside a closure, passing it `jQuery`. The following code allows you to safely use the `$` object:

```
(function($){  
    // Your code goes here  
})(jQuery);
```

As you remember, JavaScript functions do not require you to invoke them with exactly the same number of parameters as they were declared with. Hence, when you invoke `jQuery` constructor you can pass different things to it. You can pass a String as an argument or another function, and `jQuery` constructor will invoke different code based on the argument type. For example, if you'll pass it a String, `jQuery` will assume that it's a CSS selector, and the caller wants to find element(s) in the DOM that match this selector. Basically, you may think of it this way - whenever you want `jQuery` do something for you, invoke `$()` passing it your request.

You'll need to get used to yet another feature of `jQuery` - method chaining. Each function returns an object, and you don't have to declare a variable to hold this object. You can just write something like `funcA().funcB()`; This means that the method `funcB()` will be called on the object, returned by the `funcA()`.



While method chaining is often presented as a great feature allowing to do more with less typing, it may complicate debugging of your code. Imagine that `funcA()` returned null for whatever reason. The entire chain (the `funcB()` in our example) that was attached to `funcA()` won't be working properly, and you may need to unchain these methods to find the problem.

Also, if you need to get an access to a DOM object more than once, save the reference in a variable and reuse it rather than invoking the same selector method in several chains - it can improve the performance of your Web page.

## Hello World

The "Hello World" program is always a good start to learn any new software, and we'll go this route too. The following code sample uses `jQuery` to display a Web page that reads "Hello World!". Note the functions that start with the `$` sign - they are all from `jQuery` library.

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Hello jQuery</title>
  </head>
  <body>
    <script src="js/libs/jquery-1.9.1.min.js"></script>
    <script>
      $(function(){ // ①
        $("body").append("<h1>Hello World!</h1>"); // ②
      });
    </script>
  </body>
</html>

```

- ❶ If the script passes a function as an argument to jQuery, such a function is called when the DOM object is ready - the jQuery's `ready()` function gets invoked . Keep in mind that it's not the same as invoking a function handler `window.on load`, which is called after all windows resources (not just the DOM object) are completely loaded (read more in the [jQuery Events](#) section).
- ❷ If the script passes a String to jQuery, such String is being treated as a CSS selector, and jQuery tries to find the matching collection of HTML elements (it'll return the reference to just one `<body>` in the Hello World script). This line also demonstrates the method chaining - the `append()` method is called on the object returned by `$( "body" )`.

## Selectors and Filters

Probably the most frequently used routine in a JavaScript code that's part of the HTML page is finding DOM elements and making some manipulations with them, and this is where the jQuery's power is. Finding HTML elements based on the [CSS selectors](#) is very easy and concise. You can specify one or more selectors in the same query. Below is a code snippet with a number of random samples of selectors. Going through this code and reading comments will help you to understand how to use [jQuery selectors](#) (note that with jQuery you can write one selector for multiple ID's, which is not allowed in the pure JavaScript's `getElementById()`).

```

$(".donate-button"); // find the elements with the class donate-button

$("#login-link") // find the elements with id=login-link

// find elements with id=map-container or id=video-container
$("#map-container, #video-container");

```

```
// Find an HTML input element that has a value attribute of 200
$('input[value="200"]');

// Find all <p> elements that are nested somewhere inside <div>
$('div p');

// Find all <p> elements that are direct children (located directly inside) <div>
$('div>p');

// Find all <label> elements that are styled with the class donation-heading
$('label.donation-heading');

// Find an HTML input element that has a value attribute of 200
// and change the text of its next sibling to "two hundred"
$('input[value="200"]').next().text("two hundred");
```



If jQuery returns a set of elements that match the selector's expression, you can access its elements using array notation: `var thesecondDiv = $('div')[1]`. If you want to iterate through the entire set use jQuery method `$(selector).each()`. For example, if you want to perform some function on each paragraph of an HTML document, you can do it as follows: `'$("p").each(function(){...})'`.

## Testing jQuery Code with JSFiddle

There is a handy online site **JSFiddle** for performing quick testing of the code fragments of HTML, CSS, JavaScript, and popular frameworks. This Web page has a sidebar of the left and four large panels on the right. Three of these panels are for entering or copy/pasting: HTML, CSS, and JavaScript, and the fourth panel is for showing the results of applying this code (see [Figure 3-1](#)).

```


<div>
        <label class="donation-heading">Please select or enter<br/>
        <input type="radio" name="amount" id="d10" value="10"/>
        <input type="radio" name="amount" id="d20" value="20"/>
        <input type="radio" name="amount" id="d50" checked="" checked="" value="50"/>
        <input type="radio" name="amount" id="d100" value="100"/>
        <input type="radio" name="amount" id="d200" value="200"/>
        <input id="customamount" name="amount" value="" type="text" autocomplete="off"/>
    </div>
    <button class="donate-button donate-button-submit">
        <span>Donate Now</span>
        <br/>
        <span class="donate-2nd-line">Children can't wait</span>
    </button>


```

```

.donate-button {
    background: #ff6060;
    background: -webkit-linear-gradient(Linear, left top, left bottom, #ff6060, #ff6060);
    background: -moz-linear-gradient(Cusp, #fec000, #ff6060);
    background: -ms-linear-gradient(Cusp, #fec000, #ff6060);
    background: -o-linear-gradient(Cusp, #fec000, #ff6060);
    width:180px;
    height:40px;
    padding: 10px 0px;
    border: 1px solid black;
    border-radius: 12px;
    vertical-align: middle;
    cursor: pointer;
}
.donate-button-submit {
    margin:0;
}

```

Please select or enter  
donation amount  
 10  
 20  
 50  
 100  
 two hundred  
Other amount

Donate Now  
Children can't wait

Figure 3-1. Testing jQuery using JSFiddle

Copy/paste the fragments from the HTML and CSS written for the Donate section of the Save The Child page into the top panels, and press the button Run on JSFiddle’s toolbar, you’ll see our donate form where each radiobutton has a label in the form of digits (10, 20, 50, 100, 200). Now select jQuery 1.9.0 from the dropdown at the top left and copy paste the jQuery code fragment you’d like to test into the JavaScript panel locate under the HTML one. As you see on [Figure 3-1](#), we’ve pasted `$(input[value="200"]).next().text("two hundred")`. After pressing the button Run the jQuery script was executed and the label of the last radiobutton has been replaced from “200” to “two hundred” (test this fiddle [here](#)). JSFiddle’s tutorial is located at <http://doc.jsfiddle.net/tutorial.html>.



If you chained a method, e.g. an event handler, to the HTML element returned by a selector, your can use `$(this)` from inside such a handler to get a reference to this HTML element.

## Filtering Elements

If jQuery selector returns a number of HTML elements, you can further narrow down this collection by applying so-called filters. jQuery has such filters as `eq()`, `has()`, `first()` and more.

For example, applying the selector `$('label')`;` to the Donate section HTML fragment shown in <>FIG5-1>> would return a set of HTML elements `<label>. Say we want to change the background of the label “20” to be red. This is the third label in the HTML from [Figure 3-1](#), and the `eq(n)` filter selects the element at the zero-based index `n` within the matched set.

You can apply this filter using the following syntax: `$('label: eq(2)')`; But jQuery documentation suggests using the syntax `$('label').eq(2)`; [for better performance](#).

Using method chaining we’ll apply the filter `eq(2)` to the set of lables returned by the selector `$('label')` and then and then change the styling of the remaining HTML element(s) using the `css()` method that can do all CSS manipulations. This is how the entire expression will look like:

```
$('label').eq(2).css('background-color', 'red');
```

Test this script in JSFiddle or in the code of one of the Save The Child projects from this chapter. The background of the label “20” will become red. If you wanted to change the CSS of the first label in this set, the filter expressions could look as `$('label: first')` or, for the better performance, you should do it like this:

```
$('label').filter(':first').css('background-color', 'red');
```

If you display data in HTML table, you may want to change the background color of every even or odd row `<tr>`, and jQuery offers you the filters `even()` and `odd()`, for example:

```
$('tr').filter(':even').css('background-color', 'grey');
```

Usually, you’d be doing this to interactively change the background colors. You can also alternate background colors by using straight CSS selectors `p:nth-child(odd)` and `p:nth-child(even)`.

Visit [jQuery API documentation](#) for the complete list of [selectors](#) and [traversing filters](#).



If you need to display data in a grid-like form, consider using a Java-Script grid called [SlickGrid](#).

## Handling Events

Adding events processing with jQuery is simple. Your code will follow the same pattern: find the element in DOM using selector or filter, and then attach the appropriate function that handles the event. We’ll show you a handful of code samples of how to do it,

but you can find the description of all methods that deal with events in the [jQuery API documentation](#).

There are a couple of ways of passing the handler function to be executed as callback when a particular event is dispatched. For example, Our Hello World code used passes a handler function to the `ready` event:

```
$(function());
```

This is the same as using the following syntax:

```
$(document).ready(function());
```

For the Hello World example this was all that mattered - we just needed to have the DOM object to be able to append the `<h1>` element to it. But this would not be the right solution if the code needs to be executed only after all page resources have been loaded. In such case the code could have been re-written to utilize the DOM's `window.load` event, which in jQuery looks as follows:

```
$(window).load(function(){
    $("body").append("<h1>Hello World!</h1>");
});
```

If the user interacts with your Web page using the mouse , the events handlers can be added using a similar procedure. For example, if you want the header in our Hello World example to process click events, find the reference to this header and attach the `click()` handler to it. Adding the following to the `<script>` section of Hello World will append the text each time the user clicks on the header.

```
 $("h1").click(function(event){
    $("body").append("Hey, you clicked on the header!");
})
```

If you'd like to process double-clicks - replace the `click()` invocation with `dblclick()`. jQuery has handlers for about a dozen mouse events, which are wrapper methods to the corresponding JavaScript events that are dispatched when mouse entering or leaving the area, the mouse pointer goes up/down, or the focus moves in or out of an input field. The shorthand methods `click()` and `dblclick()` (and several others) internally use the method `on()`, which you can and should use in your code too (it works during the bubbling phase of the event as described in the section DOM Events in Appendix A).

## Attaching Events Handlers and Elements With the Method `on()`

The event methods can be attached just by passing a handler function as it was done in the above examples, or to process the event or by using the `on()` method, which allows you to specify the native event name and the event handler as its arguments. In the section Working on Save The Child you'll see lots of examples that use the `on()` method.

The one liner below assigns the function handler named `showLoginForm` to the `click` event of the element with the id `login-link`. The following code snippets includes the commented out pure-JavaScript version of the code (see project-02-login in Chapter 1) that has the same functionality:

```
// var loginLink = document.getElementById("login-link");
// loginLink.addEventListener('click', showLoginForm, false);

$('#login-link').on('click', showLoginForm);
```

The `on()` method allows you to assign the same handler function to more than one event. For example, to invoke the `showLoginForm` function when the user clicks or moves the mouse over the HTML element you could written `on('click mouseover', showLoginForm)`.

The method `off()` is used for removing the event handler and the event won't be processed anymore. For example, if you want to turn off the login link's ability to process `click` event, simply write this:

```
$('#login-link').off('click', showLoginForm);
```

## Delegating Events

The method `on()` can be called with passing an optional selector as an argument. Since we haven't used selectors in the example from the previous section, the event was triggered only when reached the element with an id `login-link`. Now imagine an HTML container that has child elements, e.g. a calculator implemented as a `<div id="calculator">` containing buttons. The following code would assign a click handler **to each** button styled with a class `.digitButton`:

```
$("#calculator .digitButton").on("click", function(){...});
```

But instead of assigning an event handler to each button, you can assign an event handler to the container and specify additional selector that child elements may be found by. The following code assigns the event handler function **to only one** object - the `div#calculator` instructing this container to invoke the event handler when any of its children matching `.digitButton` is clicked.

```
$("#calculator").on("click", ".digitButton", function(){...});
```

When the button is clicked, the event bubbles up and reaches the container's level, whose click handler will do the processing (jQuery doesn't support capturing phase of events). The work on processing clicks for digit buttons is delegated to the container.

Another good use case for delegating event processing to a container is a financial application that displays the data in an HTML table containing hundreds of rows. Instead of assigning event hundreds event handlers (one per table row), assign one to the table. There is one extra benefit to using delegation in this case - if the application can dy-

namically add new rows to this table (say, the order execution data), there is no need to explicitly assign event handlers to them - the container will do the processing for both old and new rows.



Starting from jQuery 1.7, the method `on()` is a recommended replacement of the methods `bind()`, `unbind()`, `delegate()`, and `undelegate()` that are still being used in earlier versions of jQuery. If you decide to develop your application with jQuery and its mobile version with jQuery Mobile, you need to be aware that the latter may not implement the latest code of the core jQuery. Using `on()` is safe though, because at the time of this writing jQuery Mobile 1.2 supports all the features of jQuery 1.8.2. In Chapter 10, you'll see how using the responsive design principles can help you to reuse the same code on both desktop and mobile devices.

The method `on()` allows passing the data to the function handler (see [jQuery documentation](#) for details).

You are also allowed to assign different handlers to different events in `on` invocation of `on()`. The following code snippet from project-11-jQuery-canvas-pie-chart-json assigns handlers to `focus` and `blur` events:

```
$('#customAmount').on({
    focus : onCustomAmountFocus,
    blur : onCustomAmountBlur
});
```

## AJAX with jQuery

Making AJAX requests to the server is also easier with jQuery than with pure JavaScript. All the complexity of dealing with various flavors of `XMLHttpRequest` is hidden from the developers. The method `$.ajax()` spares JavaScript developers from writing the code with multiple browser-specific ways of instantiating the `XMLHttpRequest` object. By invoking `ajax()` you can exchange the data with the server and load the JavaScript code. In its simplest form, this method takes just the URL of the remote resource to which the request is sent. Such invocation will use global defaults that should have been set in advance by invoking the method `ajaxSetup()`.

But you can combine specifying parameters of the AJAX call and making the `ajax()` call. Just provide as an argument a configuration object that defines the URL, the function handlers for success and failures, and some other parameters like a function to call right before the AJAX request (`beforeSend`) or caching instructions for the browser (`cache`).

Spend some time getting familiar with all different configuration parameters that you can use with the jQuery method `ajax()`. Here's a sample template for calling jQuery `ajax()`:

```
$.ajax({
    url: 'myData.json',
    type: 'GET',
    dataType: 'json'
}).done(function (data) {...})
.fail(function (jqXHR, textStatus) {...});
```

This example takes a JavaScript object that defines three properties: the URL, the type of the request, and the expected data type. Using chaining, you can attach the methods `done()` and `fail()`, which have to specify the function handlers to be invoked in case of success and failure respectively. The `jqXHR` is a jQuery wrapper for the browser's `XMLHttpRequest` object.

Don't forget about the asynchronous nature of AJAX calls, which means that the `ajax()` method will be finished before the `done()` or `fail()` callbacks will be invoked. You may attach another *promised callback* method `always()` that will be invoked regardless of if the `ajax()` call succeeds or fails.



An alternative to having a `fail()` handler for each ajax request is setting the global error handling routine. Consider doing this for some serious HTTP failures like 403 (access forbidden) or errors with codes 5xx. For example:

```
$(function() {
    $.ajaxSetup({
        error: function(jqXHR, exception) {
            if (jqXHR.status == 404) {
                alert('Requested resource not found. [404]');
            } else if (jqXHR.status == 500) {
                alert('Internal Server Error [500].');
            } else if (exception === 'parsererror') {
                alert('JSON parsing failed.');
            } else {
                alert('Got This Error:\n' + jqXHR.responseText);
            }
        });
    });
});
```

If you need to chain asynchronous callbacks (`done()`, `fail()`, `always()`) that don't need to be called right away (they wait for the result) the method `ajax()` returns **Deferred** object. It places these callbacks in a queue to be called later. As a matter of fact, the callback `fail()` may never be called if no errors occurred.

If you specify JSON as a value of the `dataType` property, the result will be parsed automatically by jQuery - there is no need to call `JSON.parse()` as it was done in Chapter 2. Even though the jQuery object has a utility method `parseJSON()`, you don't have to invoke it to process return of the `ajax()` call.

In the above example the type of the AJAX request was GET. But you can use POST too. In this case you'll need to prepare valid JSON data to be sent to the server. In this case the configuration object that you provide as an argument to the method `ajax()` has to include the property `data` containing valid JSON.

## Handy Shorthand Methods

jQuery has several shorthand methods that allow making AJAX calls with the simpler syntax, which we'll consider next.

The method `load()` makes an AJAX call from an HTML element(s) to the specified URL (the first argument) and populates the HTML element with the returned data. You can pass optional second and third arguments: HTTP request parameters and the callback function to process the results. If the second argument is an object, the `load()` method will make a POST request, otherwise - GET. You'll see the code that uses `load()` to populate states and countries from remote HTML files later in this chapter in the section on bringing the states and countries from remote HTML files. But the next line shows an example of calling `load()` with two parameters: the URL and the callback:

```
$('#countriesList').load('data/countries.html', function(response, status, xhr){...});
```

The global method `get()` allows you to specifically issue an HTTP GET request. Similarly to the `ajax()` invocation, you can chain the `done()`, `fail()`, and `always()` methods to `get()`, for example:

```
$.get('ssc/getDonors?city=Miami', function(){alert("Got the donors");})
  .done(function(){alert("I'm called after the donors retrieved");}
  .fail(function(){alert("Request for donors failed");});
;
```

The global method `post()` makes an HTTP POST request to the server. You must specify at least one argument - the URL on the server, and, optionally, the data to be passed, the callback to be invoked on the request completion, and the type of data expected from the server. Similarly to the `ajax()` invocation, you can chain the `done()`, `fail()`, and `always()` methods to `post()`. The following example makes a POST request to the server passing an object with the new donor information.

```
$.post('ssc/addDonor', {id:123, name:"John Smith"});
;
```

The global method `getJSON()` retrieves and parses the JSON data from the specified URL and passes the JavaScript object to the specified callback. If need be, you can send

the data to the server with the request. Callinf `getJSON()` is like calling `ajax()` with parameter `dataType: "json"`.

```
$.getJSON('data/us-states-list.json', function (data) {
    // code to populate states combo goes here})
    .fail(function(){alert("Request for us states failed");});
```

The method `serialize()` is used when you need to submit to the server a filled out HTML `<form>`. This method presents the form data as a text sting in a standard URL-encoded notation. Typically, the code finds a required form using jQuery selector and then calls `serialize()` on this object. But you can invoke `serialize()` not only on the entire form, but on selected form elements too. Belows is a sample code that finds the form and serializes it.

```
$('form').submit(function() {
    alert($(this).serialize());
    return false;
});
```



Returning `false` from a jQuery event handler is the same as calling either `preventDefault()` and `stopPropagation()` on the `jQuery.Event` object. In pure JavaScript returning `false` doesn't stop propagation (try to run [this fiddle](#)).

Later in this chapter in the section Submitting Donate Form you'll see a code that uses `serialize()` method.

## Save The Child With jQuery

In this section we'll review code samples from several small projects (see Appendix C for running instructions) that are jQuery re-writes of the corresponding pure-JavaScript projects from Chapters 1 and 2. We are not going to add any new functionality - the goal is to demonstrate how jQuery allows you to achieve the same results while writing less code. You'll also see how it can save you time by handling browser incompatibility for common uses (like AJAX).

### Login and Donate

For example, the file `main.js` from project-02-jQuery-Login is 33% less in size than `project-02-login` written in pure JavaScript. jQuery allows your programs to be brief. For example, the next code shows how six lines of code in JavaScript can be replaced with one - the jQuery function `toggle()` will toggle the visibility of `login-link`, `login-form`, and `login-submit`.

Note. The total size of your jQuery application is not necessarily smaller comparing the pure JavaScript one because it includes the code of jQuery library.

```
function showLoginForm() {  
  
    // The JavaScript way  
    // var loginLink = document.getElementById("login-link");  
    // var loginForm = document.getElementById("login-form");  
    // var loginSubmit = document.getElementById('login-submit');  
    // loginLink.style.display = "none";  
    // loginForm.style.display = "block";  
    // loginSubmit.style.display = "block";  
  
    // The jQuery way  
    $('#login-link, #login-form, #login-submit').toggle();  
}
```

The code of the Donation section also becomes slimmer with jQuery. For example, the following section from the JavaScript version of the application is removed:

```
var donateButton = document.getElementById('donate-button');  
var donationAddress = document.getElementById('donation-address');  
var donateFormContainer = document.getElementById('donate-form-container');  
var customAmount = document.getElementById('customAmount');  
var donateForm = document.forms['_xclick'];  
var donateLaterLink = document.getElementById('donate-later-link');
```

The jQuery method chaining allows combining (in one line) finding DOM objects and acting upon them. The following is the entire code of the main.js from project-01-jQuery-make-donation, which includes the initial version of the code of Login and Donate sections of Save The Child.

```
/* ----- login section ----- */  
  
$(function() {  
  
    function showLoginForm() {  
        $('#login-link, #login-form, #login-submit').toggle();  
    }  
  
    $('#login-link').on('click', showLoginForm);  
  
    function showAuthorizedSection() {  
        $('#authorized, #login-form, #login-submit').toggle();  
    }  
  
    function logIn() {  
        var userNameValue = $('#username').val();  
        var userNameValueLength = userNameValue.length;  
        var userPasswordValue = $('#password').val();  
        var userPasswordLength = userPasswordValue.length;
```

```

//check credentials
if (userNameValueLength == 0 || userPasswordLength == 0) {
    if (userNameValueLength == 0) {
        console.log('username is empty');
    }
    if (userPasswordLength == 0) {
        console.log('password is empty');
    }
} else if (userNameValue != 'admin' || userPasswordValue != '1234') {
    console.log('username or password is invalid');
} else if (userNameValue == 'admin' && userPasswordValue == '1234') {
    showAuthorizedSection();
}
}

$('#login-submit').on('click', logIn);

function logOut() {
    $('#username, #password').val('')
    $('#authorized, #login-link').toggle();
}

$('#logout-link').on('click', logOut);

$('#profile-link').on('click', function() {
    console.log('Profile link was clicked');
});
});

/* ----- make donation module start ----- */
$(function() {
    var checkedInd = 2; // initially checked radiobutton

    // Show/hide the donation form if the user clicks
    // on the button Donate or the Donate Later
    function showHideDonationForm() {
        $('#donation-address, #donate-form-container').toggle();
    }
    $('#donate-button').on('click', showHideDonationForm);
    $('#donate-later-link').on('click', showHideDonationForm);
    // End of show/hide section

    $('#donate-form-container').on('click', resetOtherAmount);

    function resetOtherAmount(event) {
        if (event.target.type == "radio") {
            $('#otherAmount').val('');
        }
    }
}

//uncheck selected radio buttons if other amount was chosen

```

```

function onOtherAmountFocus() {
    var radioButtons = $('form[name="_xclick"] input:radio');
    if ($('#otherAmount').val() == '') {
        checkedInd = radioButtons.index(radioButtons.filter(':checked'));
    }
    $('form[name="_xclick"] input:radio').prop('checked', false); // ❶
}

function onOtherAmountBlur() {
    if ($('#otherAmount').val() == '') {
        $('form[name="_xclick"] input:radio:eq(' + checkedInd + ')')
            .prop("checked", true); // ❷
    }
}
$('#otherAmount')
    .on({focus:onOtherAmountFocus, blur:onOtherAmountBlur}); // ❸

});

```

- ❶ This one liner finds all elements of the form named `_xclick`, and immediately applies the jQuery filter to remove from this collection any elements except radiobuttons. Then it unchecks all of them by setting the property `checked` to `false`. This has to be done if the user places the focus inside the “Other amount” field.
- ❷ If the user leaves the “Other amount” return the check the previously selected radiobutton again. The `eq` filter picks the radiobutton whose number is equal to the value of the variable `checkedInd`.
- ❸ A single invocation of the `on()` method registers two event handlers: one for the `focus` and one for the `blur` event.

jQuery includes a **number of effects** that make the user experience more engaging. Let’s use one of them called `fadeToggle()`. In the code above there is a section that toggles visibility of the Donate form. If the user clicks on the Donate button, the form becomes visible (see [Figure 1-11](#)). If the user clicks on the link “I’ll donate later”, the form becomes hidden as in [Figure 1-10](#). The jQuery method `toggle()` does its job, but the change happens abruptly. The effect `fadeToggle()` allows to introduce slower fading which improves the user experience, at least to our taste.

If the code would hide/show just one component, the code change would be trivial - replacing `toggle()` with `fadeToggle('slow')` would do the trick. But in our case, the `toggle` changes visibility of two `<div>`:`s`: `donation-address` and `donation-form-container`, which should happen in a certain order. The code below is a replacement of the `show/hide` section in the `main.js` to introduce the fading effect.

```

function showHideDonationForm(first, next) {
    first.fadeToggle('slow', function() {
        next.fadeToggle('slow');
    });
}

```

```

        });
    }

    var donAddress = $('#donation-address');
    var donForm = $('#donate-form-container');

    $('#donate-button').on('click', function() {
        showHideDonationForm(donAddress, donForm));
    });

    $('#donate-later-link').on('click', function() {
        showHideDonationForm(donForm, donAddress));
    });

```

If you want to see the difference, first run the project-01-jQuery-make-donation and click on the Donate button (no effects), and then run project-04-jQuery-donation-ajax-json, which has the fading effect.

## HTML States and Countries With jQuery AJAX

The project project-03-jQuery-donation-ajax-html illustrates retrieving the HTML data about the states and countries using jQuery method `load()`. Here's the fragment from `main.js` that makes two `load()` calls. The second call purposely misspells the name of the file to generate error.

```

function loadData(dataUrl, target, selectionPrompt) {
    target.load(dataUrl,
        function(response, status, xhr) { // ①
            if (status != "error") {
                target.prepend(selectionPrompt); // ②
            } else {
                console.log('Status: ' + status + ' ' + xhr.statusText);

                // Show the error message on the Web page
                var tempContainerHTML = '<p class="error">Error getting ' + dataUrl +
                    ': ' + xhr.statusText + ", code: " + xhr.status + "</p>";

                $('#temp-project-name-container').append(tempContainerHTML); // ③
            }
        });
}

var statePrompt =
    '<option value="" selected="selected"> - State - </option>';
loadData('data/us-states.html', $('#state'), statePrompt);

var countryPrompt =
    '<option value="" selected="selected"> - Country - </option>';

// Pass the wrong data URL on purpose
loadData('da----ta/countries.html', $('#countriesList'), countryPrompt); // ④

```

- ① The callback to be invoked right after the `load()` completes the request.

- ② Using jQuery method `prepend()` insert the very first element to HTML `<select>` to prompt the user to select a state or a country.
- ③ Display an error message at the bottom of the Web page in the `<div>` with ID `temp-project-name-container`.
- ④ Pass the misspelled data URL to generate error message.

## JSON States and Countries With jQuery AJAX

The project named project-04-jQuery-donation-ajax.json demonstrates how to make a jQuery `ajax()` call to retrieve the JSON data about countries and states and populate the respective comboboxes in the donation form. The function `loadData()` in the following code fragment takes three arguments: the data URL, the name of the root element in the JSON file and the target HTML element to be populated with the data retrieved from the AJAX call.

```

function loadData(dataUrl, rootElement, target) {
    $.ajax({
        url: dataUrl,
        type: 'GET',
        cache: false,
        timeout: 5000, // ①
        dataType: 'json'
    }).done(function (data) { // ②
        var optionsHTML = '';
        $.each(data[rootElement], function(index) {
            optionsHTML+='option value="'+data[rootElement][index].code+'"' + data[rootElement][index].name+''
        });

        var targetCurrentHTML = target.html(); // ③
        var targetNewHTML = targetCurrentHTML + optionsHTML;
        target.html(targetNewHTML);
    }).fail(function (jqXHR, textStatus, error) { // ④
        console.log('AJAX request failed: ' + error +
                    ". Code: " + jqXHR.status);

        // The code to display the error in the
        // browser's window goes here
    });
}

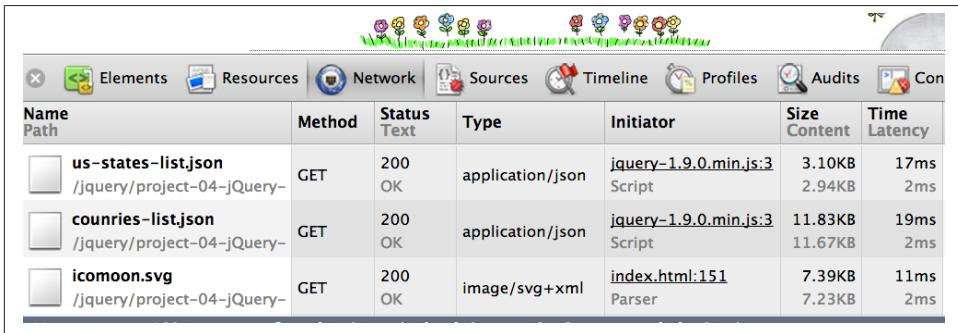
// Load the State and Country comboboxes
loadData('data/us-states-list.json', // ⑤
          'usstateslist', $('#state'));
loadData('data/countries-list.json', // ⑥
          'countrieslist', $('#countriesList'));

```

- ❶ Set the timeout. If the result of the `ajax()` call won't return within 5 second, the method `fail()` will be invoked.
- ❷ The handler function to process the successfully retrieved data
- ❸ Get the content of the HTML `<select>` element to populate with states or countries. The jQuery method `html()` uses the browser's `innerHTML` property.
- ❹ The handler function to process errors, if any
- ❺ Calling `loadData()` to retrieve states and populate the `#state` combobox. The `usstatelist` is the name of the root element in the json file `us-states-list.json`.
- ❻ Calling `loadData()` to retrieve countries and populate the `#countriesList` combobox

Compare this code with the pure JavaScript version from Chapter 2 that populates states and countries. If the jQuery code doesn't seem to be shorter, keep in mind that to writing a cross-browser version in pure JavaScript would require more than a dozen of additional lines of code that deal with instantiation of `XMLHttpRequest`.

Run the project-04-jQuery-donation-ajax-json and open Google Developer Tools and click on the Network tab. From [Figure 3-2](#) you can see that jQuery made two successful calls retrieving two JSON files with the data on states and countries.



The screenshot shows the Network tab in Google Developer Tools. It lists three requests:

Name	Method	Status	Type	Initiator	Size	Time
Path		Text			Content	Latency
<code>us-states-list.json</code> <code>/jquery/project-04-jQuery-</code>	GET	200 OK	application/json	<code>jquery-1.9.0.min.js:3</code> Script	3.10KB 2.94KB	17ms 2ms
<code>countries-list.json</code> <code>/jquery/project-04-jQuery-</code>	GET	200 OK	application/json	<code>jquery-1.9.0.min.js:3</code> Script	11.83KB 11.67KB	19ms 2ms
<code>icomoon.svg</code> <code>/jquery/project-04-jQuery-</code>	GET	200 OK	image/svg+xml	<code>index.html:151</code> Parser	7.39KB 7.23KB	11ms 2ms

*Figure 3-2. Calling `ajax()` to retrieve states and countries*

Click on the the countries-list on the left (see [Figure 3-3](#)) and you'll see the JSON data in the response object.

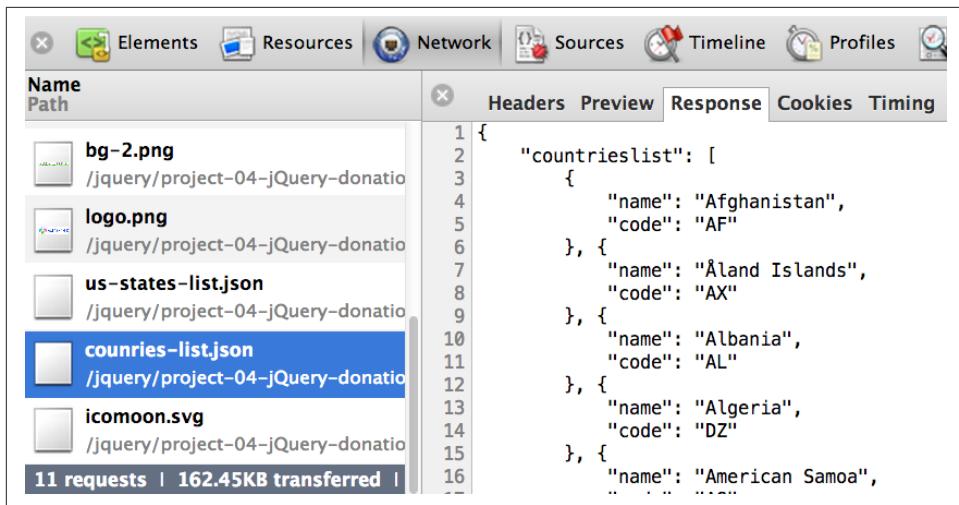


Figure 3-3. The JSON with countries is successfully retrieved

Now let's create an error situation to test the `$.ajax().fail()` chain. Just change the name of the first parameter to be `data/countries.json` in the `loadData()` invocation. There is no such file and the AJAX call will return the error 404 - see the Watch expressions in [Figure 3-4](#) that depicts the moment when the script execution stopped at the breakpoint in the `fail()` method.

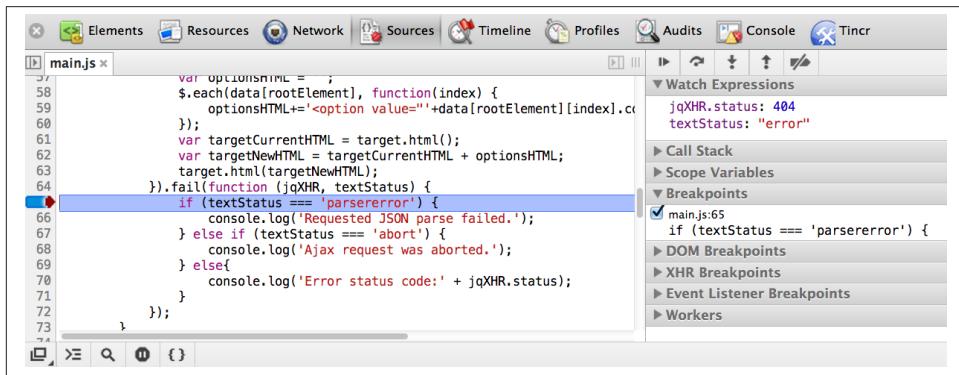


Figure 3-4. The file countries.json is not found: 404

## Submitting Donate Form

Our Save The Child application should be able to submit the donation form to [Paypal.com](https://www.paypal.com). The file `index.html` from project `project-04-jQuery-donation-ajax-json` contains the form with `id="donate-form"`. The fragment of this form is shown below.

```

<form id="donate-form" name="_xclick" action="https://www.paypal.com/cgi-bin/webscr" method="post">
    <input type="hidden" name="cmd" value="_xclick">
    <input type="hidden" name="paypal_email"
           value="email-registered-in-paypal@site-url.com">
    <input type="hidden" name="item_name" value="Donation">
    <input type="hidden" name="currency_code" value="USD">
    <div class="donation-form-section">
        <label class="donation-heading">Please select or enter
            <br/>
            donation amount</label>
        <input type="radio" name = "amount" id="d10" value = "10"/>
        <label for = "d10">10</label>
        ...
    </div>
    <div class="donation-form-section">
        <label class="donation-heading">Donor information</label>
        <input type="text" id="full_name" name="full_name"
               placeholder="full name *" required>
        <input type="email" id="email_addr" name="email_addr"
               placeholder="email *" required>
        ...
    </div>
    <div class="donation-form-section make-payment">
        <h4>We accept Paypal payments</h4>
        <p>
            Your payment will be processed securely by <b>PayPal</b>.
        </p>
        ...
        <button class="donate-button donate-button-submit"></button>
        ...
    </div>
</form>

```

## Manual Form Serialization

If you simply want to submit this form to the URL listed in its `action` property when the user clicks on the button submit, there is nothing else to be done. This already works and Paypal's login page opens up in the browser. But if you wanted to seamlessly integrate your page with Paypal or any other third-party service, a preferred way is not to send the user to the third-party Web site, but do it without leaving your Web application. We won't be implementing such integration with Paypal here, but technically it would be possible to pass the user's credentials and bank information to charge the donor of Save The Child without even opening the Paypal Web page in the browser. To do this, you'd need to submit the form using AJAX and Paypal API with processing the results of this transaction using standard AJAX techniques.

To post the form to a specified URL using jQuery AJAX we'll serialize the data from the form on `submit` event. The code fragment from `main.js` finds the form with ID `donate-form` and chains to it the `submit()` method passing to it a callback that will prepare the

data and make an AJAX call. You may use the method `submit()` instead of attaching an event handler to process clicks on the button `donate` - the method `submit()` will be invoked not only on the Submit button click event, but when the user presses the Enter key while the cursor is in one of the form's input fields.

```
$( '#donate-form' ).submit( function() {
  var formData = $( this ).serialize();
  console.log("The Donation form is serialized:" + formData);
  // Make an AJAX call here and pass the data to the server

  return false; // stop event propagation and default action
});
```

Run project-04-jQuery-donation-ajax-json and open Chrome Developer Tools of Firebug. Then fill out the donation form as shown in **Figure 3-5:**

The screenshot shows a web browser window with the URL `localhost:63342/chapter3/project-04-jQuery-donation-ajax-json/index.html`. The page features a logo with two stylized children and the text "SAVE THE CHILD". Below the logo, there is a section titled "Make a donation today" with a sub-instruction "Please select or enter donation amount". A radio button group allows selecting amounts of 10, 20, 50, 100, or 200. An "Other amount" input field is also present. To the right, there is a "Donor information" section containing fields for name ("Alex Smith"), email ("asmith@gmail.com"), address ("123 Broarway"), city ("New York"), zip ("10013"), state ("NEW YORK"), and country ("United States"). On the right side of the form, there is a "Who We Are" button, a "We accept Paypal payments" section with a note about secure processing, and a large orange "DONATE NOW Children can't wait" button. At the bottom, there is a link "I'll donate later" with a crossed-out icon. The page is decorated with small flower icons at the bottom.

Figure 3-5. Donation Form

Now press the Enter key and you'll see the output in the console with the serialized form data that will look like this:

*"The Donation form is serialized: cmd=\_xclick&business=email-registered-in-paypal%40site-url.com&item\_name=Donation&currency\_code=USD&amount=50&amount=&full\_name=Alex+Smith&email\_addr=asmith%40gmail.com&street\_address=123+Broadway&scty>New+York&zip=10013&state=NY&country=US"*

Manual form serialization has other advantages too - you don't have to pass the entire form to the server, but select only some of the input fields to be submitted. The following code snippet shows several ways of sending the partial form content.

```
var queryString;

queryString = $('form[name="_xclick"]') // 1
    .find(':input[name=full_name],:input[name=email_addr]')
    .serialize();

queryString = $('form[name="_xclick"]') // 2
    .find(':input[type=text]')
    .serialize();

queryString = $('form[name="_xclick"]') // 3
    .find(':input[type=hidden]')
    .serialize();
```

- ① Find the form named `_xclick`, apply the filter to select only the full name and the email address and serialize only these two fields.
- ② Find the form named `_xclick`, apply the filter to select only the input fields of type `text` and serialize them
- ③ Find the form named `_xclick`, apply the filter to select only the hidden input fields and serialize them

We've prepared for you one more project illustrating manual serialization of the Donation form - project-15-jQuery-serialize-form. The main.js in this project suppresses the default processing of the form submit event and sends the form to a server side PHP script .



We decided to show you a PHP example, because Java is not the only language for developing server-side code in enterprise applications. Running JavaScript on the server with Node.JS or using one of the JavaScript engines like Google's V8 or Oracle's Nashorn can be considered too.

For the purposes of our example, we will use a common technique of creating a server-side echo script that simply returns the data received from the server. Typically, in the enterprise IT shops the server-side development is done by a separate team, and having a dummy server will allow front-end developers lower dependency on the readiness of the server with the real data feed. The file demo.php is shown next. It's located in the same directory where the index.html is.

```
<?php
if (isset($_POST['paypal_email'])) {
    $paypal_email = $_POST['paypal_email'];
```

```

$item_name = $_POST['item_name'];
$currency_code = $_POST['currency_code'];
$amount = $_POST['amount'];
$full_name = $_POST['full_name'];
$email_addr = $_POST['email_addr'];

echo('Got from the client and will send to PayPal: ' .
    '$paypal_email . ' Payment type: ' . $item_name .
    ' amount: ' . $amount . ' ' . $currency_code .
    ' Thank you ' . $full_name
. ' The confirmation will be sent to ' . $email_addr);

} else {
    echo('Error getting data');
}
exit();
?>

```

The process of integration with the payment system using [Paypal API](#) is out of this book's scope, but at least we can identify the place to do it - it's typically done on the server-side. In this chapter's example it's a server-side PHP script, but it can be a Java, .Net, Python or any other server. You'd need to replace the echo statement with the code making requests to Paypal or any other payment system. The fragment from the main.js that shows how to make a request to the demo.php comes next.

```

$('.donate-button-submit').on('click', submitSerializedData);

function submitSerializedData(event) {

    // disable the button to prevent more than one click
    onOffButton($('.donate-button-submit'), true, 'submitDisabled');

    event.preventDefault(); // ①

    var queryString;

    queryString = $('form[name="_xclick"]') // ②
        .find(':input[type=hidden][name!=cmd], :input[name=amount][value!=""], '
        ':input[name=full_name], :input[name=email_addr]')
        .serialize();

    console.log('----- get the form inputs data -----');
    console.log("Submitting to the server: " + queryString);

    $.ajax({
        type : 'POST',
        url : 'demo.php', // ③
        data : queryString
    }).done(function(response) {
        console.log('----- response from demo.php -----');
        console.log("Got the response from the ajax() call to demo.php: " +
            response);
    });
}

```

```

        // enable the donate button again
        onOffButton($('.donate-button-submit'), false, 'submitDisabled');
    }).fail(function (jqXHR, textStatus, error) {
        console.log('AJAX request failed: ' + error + ". Code: "
            + jqXHR.status);

        // The code to display the error in the
        // browser's window goes here
    });
}

```

- ➊ Prevent the default processing of the submit event - we don't want to simply the form to the URL listed in the form's action property.
- ➋ Serializing the form fields excluding the empty amounts and the hidden field with the name cmd.
- ➌ The serialized data from queryString will be submitted to the server-side script demo.php

## Installing the XAMPP server with PHP support

The above example uses a server-side PHP script to echo data sent to it. If you'd like to see this script in action so you can test that the client and server can communicate, deployed this script in any Web server that supports PHP. For example, you can install on your computer the XAMPP package from the [Apache Friends web site](#), which includes Apache Web Server that supports PHP, FTP, preconfigured MYSQL database server(we are not going to use it). The installation process is very simple - just go through the short instructions on the Apache Friends website that are applicable for your OS. Start the XAMPP Control application and click on the button Start next to the label Apache. By default, Apache server starts on the port 80, so entering **http://localhost** will open the XAMPP welcome page.



If you use MAC OS X, you may need to kill the pre-installed Apache server by using the **sudo apachectl stop** command.

The directory xampp/htdocs is the document root of the Apache Web Server, hence you can place the index.html of your project there or in one of its subdirectories. To test that a PHP is supported, just save the following code in the helloworld.php in the htdocs directory:

```
<?php  
    echo('Hello World!');  
?>
```

After entering the URL <http://localhost/helloworld.php> in your Web browser, you should see a greeting from this simple PHP program. The home Web page of XAMPP server contains the link `phpinfo()` on the left panel that shows the current configuration of your PHP server.

The easiest way to test the project-15-jQuery-serialize-form that uses `demo.php` is to copy this folder into the `htdocs` directory of your XAMPP install. Then enter the URL <http://localhost/project-15-jquery-serialize-form/> in your Web browser and you'll see the Save The Child application. Then fill out the form and click on the Donate Now button. The form will be serialized and submitted to the `demo.php` as explained above. If you'll open Google Developers Tools in the Network tab you'll see that the `demo.php` has received the AJAX request and the console will show the output similar to the following (for Alex Smith, [alex@gmail.com](mailto:alex@gmail.com)):

```
----- get the form inputs data ----- main.js:138  
Submitting to the server: paypal_email=email-registered-in-paypal%40  
site-url.com&item_name=Donation+to+the+Save+Sick+Child&currency_code  
=USD&amount=50&full_name=Alex+Smith&email_addr=alex%40gmail.com main.js:139  
  
----- response from demo.php ----- main.js:146  
Got the response from the ajax() call to demo.php: Got from the client  
and will send to PayPal: email-registered-in-paypal@site-url.com  
Payment type: Donation to the Save The Child amount: 50 USD  
Thank you Alex Smith  
The confirmation will be sent to alex@gmail.com main.js:147
```

## jQuery Plugins

jQuery plugins are reusable components that know how to do a certain thing, for example validate a form or display images as a slide show. There are thousands of third-party jQuery plugins available in the [jQuery Plugin Registry](#). Below are some of the useful plugins:

- [jTable](#) - AJAX-based tables (grids) for CRUD applications
- [jQuery Form](#) - an HTML form that supports AJAX
- [HorizontalNav](#) - a navigational bar with tabs that uses the full width of its container
- [EGrappler](#) - a stylish Akordeon (collapsible panel)
- <http://paweldecowski.github.com/jQuery-CreditCardValidator/> [Credit Card Validator] - detects and validates credit card numbers
- [Responsive Carousel](#) - a slider to display images in a carousel fashion

- [morris.js](#) - a plugin for charting
- [Map Marker](#) - puts multiple markers on maps using Google MAP API V3.
- The [Lazy Load plugin](#) delays loading of images, which are outside of viewports.

The chances are that you will be able to find a plugin written by someone that fits your needs. jQuery plugins are usually freely available and their source code is plain JavaScript, so you can tweak it a little if need be.

## Validating the Donate Form With Plugin

The project-14-jQuery-validate illustrates the use of the jQuery [validate](#) plugin, which allows you to specify the rules to be checked when the user tries to submit the form. If the value is not valid, your custom message is displayed. We've included this plugin in index.html of project-14-jQuery-validate:

```
<script src="js/plugins/jquery.validate.min.js"></script>
```

To validate a form with this plugin, you need to invoke a jQuery selector finding the form and then call the method `validate()` on this object - this is a simplest way of using this plugin. But to have more control over the validation process you need to pass the object with validation options:

```
$("#myform").validate({// validation options go here});
```

The file main.js includes the code to validate the Donation form. The validation options can include many options described in the plugin documentation. Our code sample uses the following options:

- the `highlight` and `unhighlight` callbacks
- the HTML element to be used for displaying errors
- the name of the CSS class to style the error messages
- the validation rules



Validating data only on the client side is not sufficient. It's a good idea to warn the user about data issues without sending the data to the server. But ensure that the data was not corrupted/modified while traveling to the server re-validate them on the server side too. Besides, a malicious user can access your server without using your Web application. Doing server-side validation is a must.

The code fragment below displays error messages in the HTML element `<div id="validationSummary"></div>` that's placed above the form in index.html. The Validator

plugin provides the number of invalid form entries by invoking `validator.numberOfInvalids()`, and our code displays this number unless it's equal to zero.

```
var validator = $('form[name=_xclick"]').validate({  
  
    highlight : function(target, errorClass) { // ❶  
        $(target).addClass("invalidElement");  
        $('#validationSummary').text(validator.numberOfInvalids() +  
            " field(s) are invalid");  
        $('#validationSummary').show();  
    },  
  
    unhighlight : function(target, errorClass) { // ❷  
        $(target).removeClass("invalidElement");  
  
        var errors = validator.numberOfInvalids();  
        $('#validationSummary').text( errors + " field(s) are invalid");  
  
        if(errors == 0) {  
            $('#validationSummary').hide();  
        }  
    },  
  
    rules : { // ❸  
        full_name : {  
            required : true,  
            minlength : 2  
        },  
        email_addr : {  
            required : true,  
            email : true  
        },  
        zip : {  
            digits:true  
        }  
    },  
  
    messages : { // ❹  
        full_name: {  
            required: "Name is required",  
            minlength: "Name should have at least 2 letters"  
        },  
        email_addr : {  
            required : "Email is required",  
        }  
    }  
});
```

- ❶ When the invalid field will be highlighted, this function will be invoked. It changes the styling of the input field and updates the error count to display in the validation summary <div> on top of the form.

- ② When the error is fixed, the corrected field will be unhighlighted, and this function will be invoked. It revokes the error styling of the input field and updates the error count. If the error count is zero, the validation summary <div> becomes hidden.
- ③ Set the custom validation rules for selected form fields
- ④ Set the custom error messages to be displayed if the user enters invalid data.

Figure 3-6 shows the above code in action. After entering a one-character name and not proper email the user will see the corresponding error messages. Each message will be shown when the user leaves the corresponding field. But as soon as the user will fix any of them (e.g. enter one more letter in the name) the form will be immediately re-validated and the error messages will be removed as soon as the user fix the error.

The screenshot shows a web browser window with the URL `localhost:63342/chapter3/project-14-jQuery-validate/index.html`. The page features a logo of three interlocking shapes (blue, green, pink) and the text "SAVE THE CHILD". A navigation bar includes "Who We Are" and a sun icon. On the left, there's a section for "Make a donation today" with a dropdown menu for "donation amount" containing options 10, 20, 50, 100, and 200, where 50 is selected. Below it is an "Other amount" input field. In the center, a "Donor information" section contains two input fields: "name" (containing "A") and "email" (containing "fff"). Both fields have red borders, indicating they are invalid. A tooltip above the "name" field says "Name should have at least 2 letters" and a tooltip above the "email" field says "Please enter a valid email address.". To the right, there's a sidebar with the text "We accept Paypal payments" and a yellow button labeled "DONATE NOW Children can't wait". At the bottom right is a link "I'll donate later".

Figure 3-6. Validator's Error Messages



Before including a jQuery plugin to your application spend some time testing it - check its size and compare its performance with competing plugins.

## Adding Image Slider

Pretty often you need to add a rotation of the images feature to a Web page. The Save The Child page, for example, could rotate the images of the kids saved by the donors.

To give you yet another illustration of using jQuery plugin, we've created the project called project-16-jQuery-slider, where we integrated the jQuery plugin called **Responsive Carousel**. The file index.html of this project includes the CSS styles and the JavaScript code plugin as follows:

```
<link rel="stylesheet" href="assets/css/responsive-carousel.css" />
<link rel="stylesheet" href="assets/css/responsive-carousel.slide.css" />
<link rel="stylesheet" href="assets/css/responsive-carousel.fade.css" />
<link rel="stylesheet" href="assets/css/responsive-carousel.flip.css" />
...
<script src="js/plugins/responsive-carousel/responsive-carousel.min.js"></script>
<script src="js/plugins/responsive-carousel/responsive-carousel.flip.js"></script>
```

Run the project-16-jQuery-slider and you'll see how three plain slides rotate as shown on **Figure 3-7**. The HTML part of the container includes the three slides as follows.

```
<div id="image-carousel" class="carousel carousel-flip"
      data-transition="flip">
    <div>
      
    </div>
    <div>
      
    </div>
    <div>
      
    </div>
  </div>
```



Figure 3-7. Using Responsive Carousel plugin

With this plugin, the JavaScript code that the application developer has to write to implement several types of rotation is minimal. When the user clicks on the one of the radio buttons (Fade, Slide, or Flip transitions) the code below just changes the CSS class name to be used with the carousel.

```
$(function() {
    $("input:radio[name=transitions]").click(function() {
        var transition = $(this).val();
        var newClassName = 'carousel carousel-' + transition;
        $('#image-carousel').attr('class', '');
        $('#image-carousel').addClass(newClassName);
        $('#image-carousel').attr('data-transition', transition);
    });
});
```



To see code samples of using Responsive Carousel (including popular auto-playing slide shows) visit the Web page [Responsive Carousel variations](#).

The Validator and Responsive Carousel clearly demonstrate that jQuery plugins can save you some serious time of writing code to implement some commonly required features. It's great that the members of the jQuery community from around the world share their creations with other developers. If you can't find a plugin that fits your needs or have specific custom logic that needs to be used or reused in your application. Should you decide to write a plugin on your own, refer to the [Plugins/Authoring](#) document.

## Summary

In this chapter you became familiar with the jQuery Core library, which became the de-facto standard library in millions Web applications. Its simplicity and extensibility via the mechanism of plugins made it a must have in almost every Web page. Even if your organization standardizes decides on a more complex and feature-rich JavaScript framework, the chances are that you may find a handy jQuery plugin that will complement “the main” framework and made it into the code of your application. There is nothing wrong with this and you shouldn't be in the position of “either jQuery or XYZ” - most likely they can coexist.

We can recommend one of the small frameworks that will complement your jQuery code is [Twitter's Bootstrap](#). Bootstrap can quickly make the UI of your desktop or mobile application look stylish. Bootstrap is [the most popular framework](#) on GitHub.

Chapter 7 on test-driven development will show you how to test jQuery applications. In this chapter we've re-written the pure JavaScript application for the illustration purposes. But if this would be a real-world project to convert the Save The Child application

from JavaScript to jQuery, having tests even for the JavaScript version of the application would have helped to verify that everything transitioned to jQuery successfully.

In Chapter 11 you'll learn how to use jQuery Mobile library - an API on top of jQuery code that allows building UI for mobile devices.

Now that we've covered JavaScript, HTML5 APIs, AJAX, JSON, and jQuery library, we're going to the meat of the book: frameworks, productivity tools, and strategies for making your application enterprise-ready.



## PART II

# Enterprise Considerations

The content of this part justifies having the word Enterprise on this book's cover.

In Chapter 4 you'll learn how to use rich and feature complete frameworks - Ext JS from Sencha. While using this framework may be an overkill for a small Web site, it's pretty popular in the enterprise world, where rich-looking UI is required. Besides learning how to work with this framework, you'll build a new version of the Save The Child application in Ext JS. In [this version](#) we'll introduce an interactive chart (a popular feature for enterprise dashboards) and a data grid (any enterprise app uses grids).

Chapter 5 is a review of productivity tools used by enterprise developers (NPM, Grunt, Bower, Yeoman, CDB). It's about build tools, code generators, and managing dependencies (a typical enterprise application uses various software that need to work in harmony).

Chapter 6 is dedicated to dealing with issues that any mid-to-large enterprise Web application is facing: how to modularize the application to reduce the load time and make it more responsive. Our sample application Save The Child will be divided into modules with the help of RequireJS framework.

Chapter 7 Test-Driven Development (TDD) is a way of writing less buggy applications. TDD was originated in large projects written in such languages as Java, C++, or C#, and now it's adopted by HTML5 community. After reviewing how to do TDD in JavaScript, we'll show how to introduce testing into the Save The Child application.

Chapter 8 is about WebSockets - a new HTML5 API that can be a game changer for enterprise Web applications that need to have as fast communication with the servers as possible (think financial trading applications or online auctions). We'll show how to add an auction to our sample charity application.

Chapter 9 is a brief overview of various Web application security issues. While small Web sites often forget dealing with security vulnerabilities, this subject can't be ignored in the enterprise World.

---

# Developing Web Applications in Ext JS Framework

In the previous chapter you've got familiar with the JavaScript library `jQuery`. Now we'll introduce you to a more complex product - a JavaScript framework `Ext JS` from [Sencha](#). This is one of the most feature-complete frameworks available on the market, and you should give it a serious consideration while deciding on the tooling for your next enterprise HTML5 application.

## JavaScript Frameworks

The word *framework* implies that there is some pre-created “software frame”, and application developers need to fit their business-specific code inside such a frame. Why someone would want to do this as opposed to having a full freedom in developing your application code the way you want? The reason being that most enterprise projects are developed by teams of software engineers, and having an agreed-upon structure of the application with clear separation of software layers can make the entire process of development more productive.

There are JavaScript frameworks that are mainly forcing developers to organize application code in layers implementing the Model-View-Controller design pattern. There are more than a dozen MVC JavaScript frameworks that are being used by professional developers: [Backbone.js](#), [ExtJS](#), [AngularJS](#), [Ember.js](#), [Knockout](#) just to name a few.



Ext JS also supports MVC, and you can read about it later in this chapter in section titled MVC in Ext JS.



There is an excellent Web site called [TodoMVC](#), which shows the examples of implementing the same application (a Todo list) using various popular frameworks. Studying the source code of this application implemented in several frameworks can help you in selecting the one for your project.



To keep the size of this book manageable, we were not able to review more of JavaScript frameworks. But if you'd ask us to name one more great JavaScript framework that didn't make it into this book, we would recommend you to learn AngularJS from Google. There are lots of free online resources on AngularJS that Jeff Cunningham has collected all in one place on [GitHub](#).

Besides splitting the code into tiers, frameworks may offer a number of pre-fabricated UI components and build tools, and Ext JS is one of such frameworks.



If you decide to develop your application with Ext JS, you don't need to use jQuery library.

## What This Chapter Covers

The title clearly states that this chapter is about Ext JS framework. Providing detailed coverage of Ext JS in one chapter is almost mission impossible because of the vast variety of features this framework offers. Consider this chapter a hands-on overview of Ext JS. The material in this chapter is divided into three parts:

1. You'll get a high level overview of the Ext JS framework.
2. We'll do a code review of a new version of the Save The Child application developed with Ext JS. This is where we want you to spend most of the time in this chapter. Learn while studying commented code. We've also provided multiple links to the relevant product documentation.

## Why Ext JS?

After learning how jQuery library can simplify development of the HTML5 applications you might be wondering, what's so good about Ext JS framework that makes it worthwhile for studying. First of all, Core jQuery is just a library of utilities that simplify working with DOM and - you still need to write the Web application using HTML and

JavaScript. In addition to it, there are lots and lots of jQuery plugins that include handy widgets to add to your manually created Web site. We just mentioned the frameworks that help better organizing or modularizing your project, but enterprise application may need more. So here comes the Ext JS sales pitch.

1. Ext JS is an HTML5 framework that doesn't require you to write HTML. Your single HTML file (index.html) will just include three files in the head section: one with Ext JS framework, one CSS file, and one *app.js*, but the <body> section will be empty.
2. Ext JS includes a comprehensive library of JavaScript-based classes that can help you with pretty much everything you need to develop a Web application (UI components, UI layouts, collections, networking, collections, CSS compiler, packaging tool, and more).
3. Ext JS offers a way to write object-oriented code (for those who like it), define classes and inheritance in a way that's closer to classical inheritance and doesn't require the *prototype* property.
4. Ext JS can jump start your application development by generating the initial code layered according to the Model-View-Controller (MVC) design pattern.
5. Ext JS is a cross-browser framework that promises to automatically take care of all differences in major Web browsers.

If you just finished reading the jQuery chapter, you'll need to switch to a different state of mind. Core jQuery library was light, it didn't drastically change the way of developing pure HTML/JavaScript applications. But working with Ext JS framework is a completely different ball game. It's not about improving an existing Web page, it's about re-writing it from scratch without using HTML. Ext JS includes a rich library of UI components, a flexible class system, custom layouts, code generators. But Web browsers understand only HTML, DOM, CSS, and JavaScript. This means that the framework will have to do some extra work in converting the code written using the home-made Ext JS class system into same old HTML objects. Such extra work requires additional processing time, and we'll discuss this in the section titled "The Components Lifecycle".

## Downloading and Installing Ext JS

First, you need to know that *Ext JS* framework can be used for free only for non-commercial projects. To use Ext JS for enterprise Web development you or your firm has to purchase one of the [Ext JS licenses](#). But for studying, you can download the complete commercial version of Ext JS for free for the 45-days evaluation period.

The materials presented in this chapter were tested only with the current version of Ext JS framework, which at the time of this writing was 4.2.

After downloading the Ext JS framework, unzip it to any directory of your choice - later on the framework will be copied either in in your project directory (see the Sencha CMD section below) or in the document root of your Web server.

After unzipping the Ext JS distribution, you'll find a number of files and folders there. There are several JavaScript files containing differently-packaged Ext JS framework. You'll need to pick just one of these files. The files that include the word *all* in their names contain the entire framework and if you'll include one of the following files, all the classes will be loaded to the user's browser even though your application may never use most of them.

- ext-all.js - minimized version of the source code of Ext JS, which literally looks like one line of 1.4 million characters (it's still JavaScript, of course). Most likely you won't deploy this file on your production server.
- ext-all-debug.js - human-readable source code of Ext JS with no comments. If you like to read comments, use ext-all-debug-w-comments.js.
- ext-all-dev.js - human-readable source code of Ext JS that includes console.log() statements that generates and outputs debugging information in the browser's console.

Similarly, there are files that don't include *all* in their names: ext.js, ext-debug.js, and ext-dev.js. These are much smaller files that do not include the entire framework, but rather a minimum set of classes required to start the application. Later on, the additional classes may be lazy-loaded on the as-needed basis.



Typically, you shouldn't be using the *all* files. We recommend you to use the file ext.js and Sencha CMD tool to create a customized version of Ext JS library to be included with your application. You can find more details in the section Sencha CMD later in this chapter.

The *docs* folder contains extensive documentation - just open the file index.html in your browser and start reading and studying.

The *builds* folder includes sandboxed versions of Ext JS in case you need to to use say Ext JS 4.2 along with older versions of this framework. Browsing the *builds* folder reveals that the Ext JS framework consists of three parts:

- Ext Core - it's a free to use **JavaScript library** for enhancing Web sites. It supports DOM manipulation with CSS selectors, events and AJAX requests. It also offers a syntax to define and create classes that can extend from each other. The functionality of Ext Core is comparable to Core jQuery.
- Ext JS - a UI framework that includes a rich library of UI components.

- The Foundation - a set of useful utilities.

Such code separation allowed creators of Ext JS reuse a large portion of the framework's code in the mobile library Sencha Touch, which we'll cover in Chapter 12.



Ext JS framework is large so be prepared that your application will weigh at least 1Mb. This is not an issue for enterprise applications that run on fast networks. But if you need to create a small consumer-oriented Web site, you may be better off by just using a lightweight, easy to learn and free jQuery library or one of a dozen of other JavaScript frameworks that either improve organizational structure of your project or offer a set of a la carte components to prettify your HTML5 application. On the other hand, if you had a chance to develop or use rich Internet applications developed with such frameworks as Microsoft Silverlight or Apache Flex, then you'll quickly realize that Ext JS is the closest by functionality, rich set of components and tools.

## Getting Familiar with Ext JS and Tooling

This section is not going to be an Ext JS tutorial that gradually explains each and every feature and API of Ext JS. For this we'd need to write a fat Ext JS book. Sencha publishes detailed documentation, multiple online [examples](#) and [videos](#). In this chapter you'll get an overview of the framework.

### The First Version of Hello World

Before we'll explain how things work in Ext JS, we'll develop a Hello World application. But the section where we'll review the code of the Save The Child application will serve as a hands-on way of learning the framework. You'll read the code fragments followed by brief explanations. You'll be able to run and debug this application on your own computer seeing how various components and program layers work in practice. But first things first - let's create a couple of versions of Hello World.

Create a new directory (e.g. *hello1*). Inside *hello1* create a subdirectory named *ext* and copy there the entire content of your Ext JS installation directory. Create yet another subdirectory *app* inside *hello1* - this is where your application JavaScript files will go.

At the very minimum, every Ext JS application will contain one HTML and one JavaScript file - usually index.html and app.js. The file index.html will include the references to the CSS and JavaScript code of Ext JS and will include your app.js containing the code of the Hello World application:

```
<!DOCTYPE HTML>
<html>
```

```

<head>
    <meta charset="UTF-8">
    <title>HelloWorld</title>
    <link rel="stylesheet" href="ext/resources/ext-all-gray.css">
    <script src="ext/ext.js"></script>
    <script src="app/app.js"></script>
</head>
<body></body>
</html>

```

The next comes the content of the app.js that you should place in the *app* directory of your project. This is how the app.js may look like:

```

Ext.application({
    launch: function(){
        alert("Hello World");
    }
});

```

This Ext.application() method gets a *configuration object* - JavaScript literal - with configured launch method that's called automatically when the Web page has completely loaded. In our case it mandates to launch the anonymous function that displays the "Hello World" message. In Ext JS you'll be using such configuration objects a lot.

Open the file index.html in your Web browser and you'll see this greeting. But this was a plain vanilla Hello World. In the next section we'll automate the process of creation of a fancier Hello World (or the initial version of any other application) by using the Sencha Cmd tool.

## Configuration Options

In the pre 4.0 versions of Ext JS you'd be invoking the Ext.onReady() method instead of passing the configuration object with the launch *config option*.

Providing a function argument as a configuration object overrides configurable properties of the current instance of the class. This is different from the class properties, which are defined at the prototype level and changing a value of a property would apply to all instances of the class. When you'll be reading Ext JS online documentation for any class, you'll see three categories of class elements: Configs, Properties, and Methods. For example, this is how you can create a panel passing configs:

```

Ext.create('Ext.panel.Panel', {
    title: 'Hello',
    width: 200,
    html: '<p>World!</p>',
});

```

In this example we are creating an instance of the panel using a configuration object with three config options: title, width, and html. The values of these properties will

be assigned to the corresponding properties of this instance only. For example, the documentation for `Ext.panel.Panel` lists 116 available configs that you can set on the panel instance.

Ext JS classes are organized into packages. For example, the class `Panel` in the above example is located in the package `Ext.panel`. You'll be using packaging in your applications too. For example, in the next chapter you'll see classes from Save The Child and Clear frameworks named as `SSC.view.DonateForm` or `Clear.override.ExtJSOverrider`. Such packages should be properly *namespaced* and `SSC` and `Clear` are top-level namespaces here. The next fragment shows how to give a name to your application, and such a given name will serve as a top-level namespace.

```
Ext.application({
    name: 'SSC',
    // more config options can go here
});
```

In the next section we'll automate the process of creating of Hello World application.

## Generating Applications With Sencha CMD Tool

Sencha CMD is a handy command line tool that automates your work starting from scaffolding your application to minimizing, packaging and deploying it.

Download Sencha CMD from <http://www.sencha.com/products/sencha-cmd/download>. Run the installer, and when it's complete, open the Terminal or Command window and enter the command `sencha` - you should see a prompt with all possible commands and options that CMD understands.

For example, to generate the initial project structure for `HelloWorld` application enter the following command, specifying the absolute path to your ExtJS SDK directory (we keep it in the `/Library` directory) and to the output folder, where the generated project should reside.

```
sencha -sdk /Library/ext-4.2 generate app HelloWorld /Users/yfain11/hello
```

After the code generation was complete, you'll see the folder `hello` of the structure shown on the figure [Figure 4-1](#).

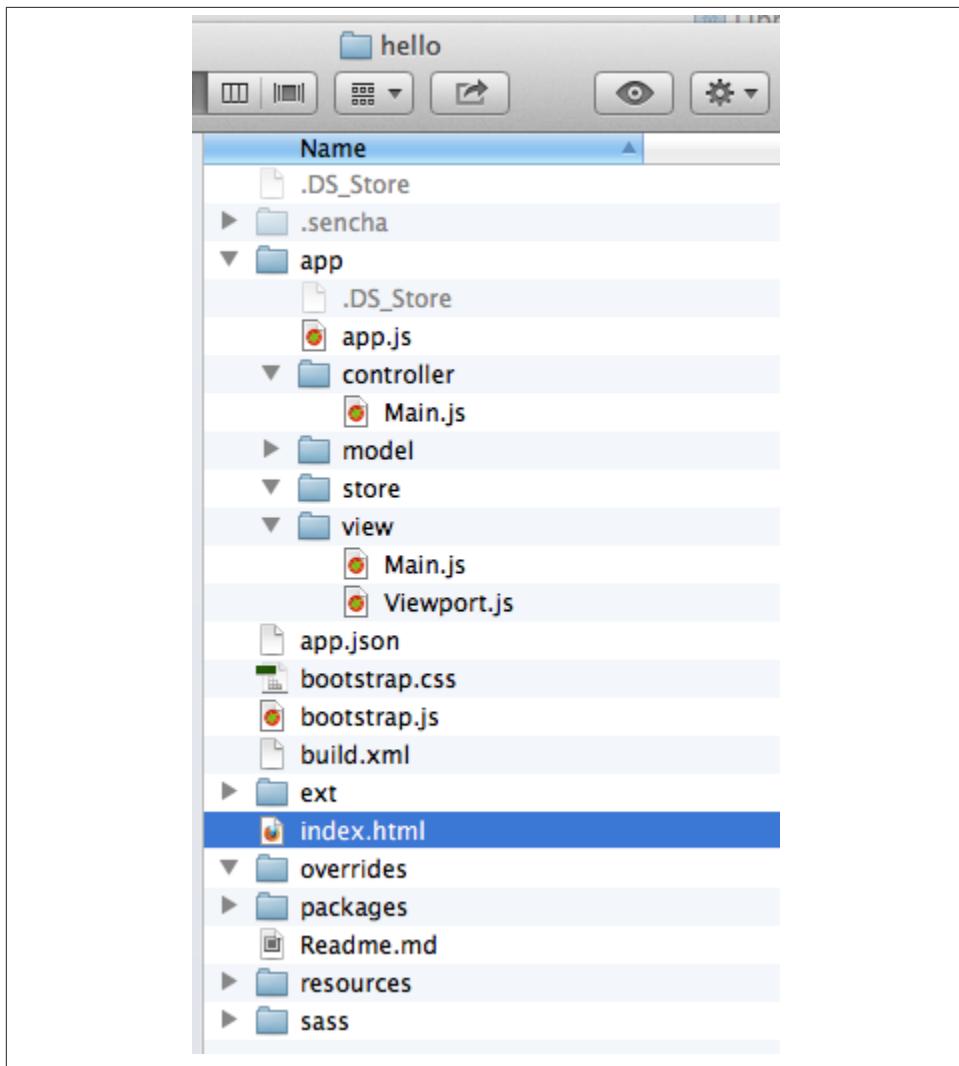


Figure 4-1. CMD-generated project

The generated project is created with the assumption that your application will be built using the MVC paradigm discussed in the section Best Practice:MVC. The JavaScript is located in the *app* folder, which includes the *view* subfolder with the visual portion of your application, the *controller* folder with controller classes, and the *model* is for data. The *ext* folder contains multiple distributions of the Ext JS framework. The *sass* folder is a place for your application's CSS files (see the sidebar titled SASS and CSS later in this chapter).

The entry point to your application is index.html, which contains the references to the main application file app.js, the Ext JS framework extdev.js, the CSS file bootstrap.css (imports the classic theme), and the supporting script bootstrap.js, which contains the mapping of the long names if the framework and application classes to their shorter names (*xtypes*). Here's how the generated index.html file looks:

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="UTF-8">
    <title>HelloWorld</title>
    <!-- <x-compile> -->
    <!-- <x-bootstrap> -->
        <link rel="stylesheet" href="bootstrap.css">
        <script src="ext/ext-dev.js"></script>
        <script src="bootstrap.js"></script>
    <!-- </x-bootstrap> -->
    <script src="app/app.js"></script>
<!-- </x-compile> -->
</head>
<body></body>
</html>
```

The content of the generated app.js is shown next. This script just calls the method `Ext.application()` passing as an argument configuration object that specifies the application name, and the names of the classes that play roles of views and controller. We'll go into details a bit later, but at this point let's concentrate on the big picture.

```
Ext.application({
    name: 'HelloWorld',

    views: [
        'Main',
        'Viewport'
    ],

    controllers: [
        'Main'
    ],

    autoCreateViewport: true
});
```

Finally, if you'll open index.html in your Web browser, you'll see our Hello World initial Web page that looks as in [Figure 4-2](#). This view uses so called border layout and shows a panel on the west and atabpanel in the central region of the view.

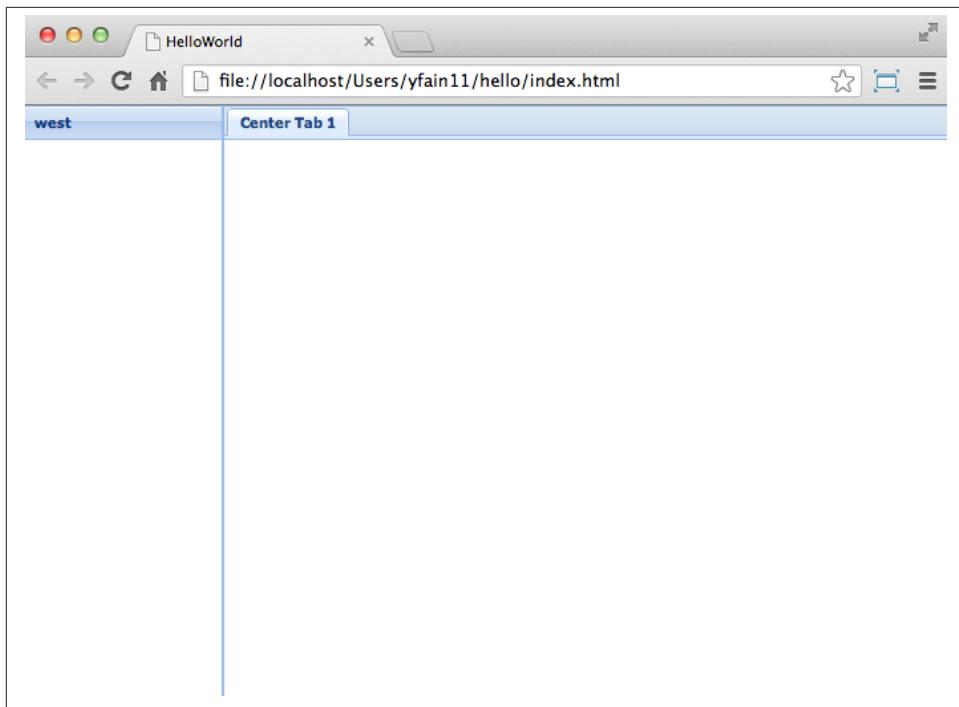


Figure 4-2. The UI of our Sencha CMD-generated Application

The total size of this version of the Hello World application is pretty large: 4Mb, and the browser makes 173 requests to the server by the time the user sees the application shown on [Figure 4-2](#). But Sencha Cmd knows how to build the production version of the Ext JS application. It minimizes and merges the application's and required framework's JavaScript code into one file. The application's css file is also minimized and the references to the image resources become relative hence shorter. Besides, the images may be automatically sliced - cut into smaller rectangular pieces that can be downloaded by the browser simultaneously.

To create optimized version of your application go to the Terminal or a command window and change to the root directory of your application (in our case it's `/Users/yfain11/hello`) and run the following command:

```
sencha app build
```

After the build is finished, you'll see newly generated version of the application in the directory `build/HelloWorld/production`. Open the file `index.html` while running Chrome Developers Tools, and you'll see that the total size of the application is substantially lower (about 900Kb) and the browser had to make only five requests to the server (see [Figure 4-3](#)). Using gZip will reduce the size of this application to 600Kb,

which is still a lot, but Ext JS framework is not the right choice for writing Hello World type of applications or light Web sites.

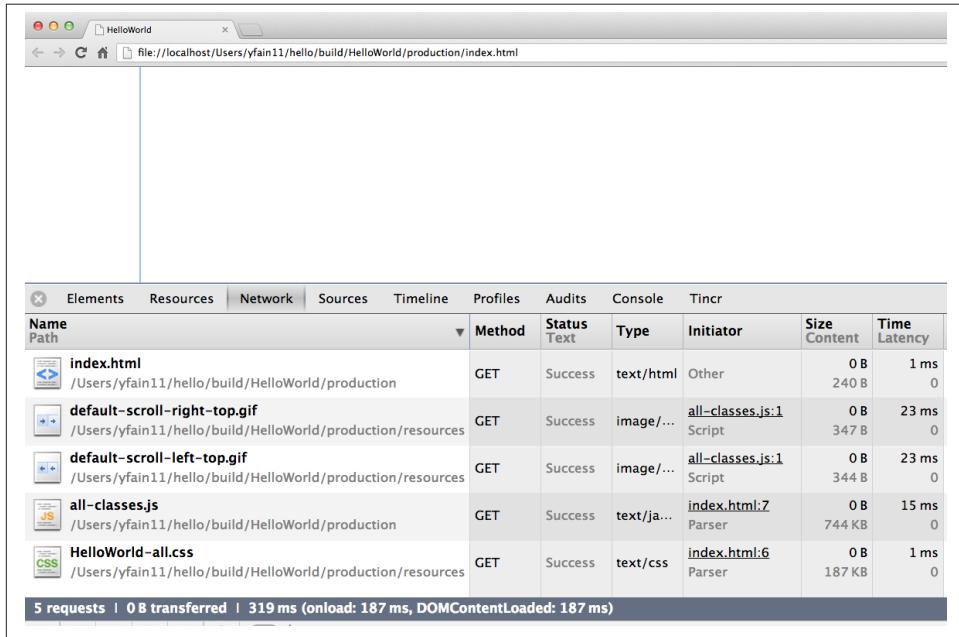


Figure 4-3. Running production version of Hello World

For more details about code generation refer to the section [Using Sencha Cmd with Ext JS](#) in the product documentation.



**Sencha Desktop Packager** allows you to take an existing Ext JS Web application (or any other HTML5 application) and package it as a native desktop application for Windows and MAC OS X. Your application can also integrate with native menus, file dialogs and access the file system.

Later in this chapter we'll use Sencha CMD tool again in the section “Building Production Version” to create an optimized version of the Save The Child application.



Sencha CMD comes with embedded **Web server**. To start the server on the default port 1841, open the Terminal or command window in your application directory and run the command `sencha web start`. To serve your Web application on another port, e.g. 8080 and from any directory, run it as follows : `sencha fs web -port8080 start -map /path/to/app/docrootdir`.

If your organization is developing Web applications with Ext JS without using Sencha CMD - it's a mistake. Sencha CMD is a very useful code generator and optimizer that also enforces the MVC principles of application design.

## Which Ext JS Distribution to Use?

First you need to select the packaging of the Ext JS framework that fits your needs. You may select its minimized version to be used in production or a larger and commented version with detailed comments and error messages. As we mentioned earlier in this chapter, you may select a version of Ext JS that include either all or only the core classes. The third option is to create a custom build of Ext JS that will include only the those framework classes that are used by your application.

The files with the minimized production version of Ext JS are called ext-all.js (all classes) and ext.js (just the core classes plus the loader of required classes). We usually pick ext-all.js for development, but for production use the distribution fine-tuned for our application as described in the section on Sencha CMD.

If this application will be used on the high-speed networks and size is not the object, simply add it to your index.html from your local servers or see if Sencha offers the CDN for the Ext JS version you need, which may look similar to the following:

```
<link rel="http://cdn.sencha.io/ext-4.2.0-gpl/resources/css/ext-all.css" />  
  
<script type="text/javascript" charset="utf-8"  
src="http://cdn.sencha.io/ext-4.2.0-gpl/ext.js"></script>
```

## Declaring, Loading and Instantiating Classes

Pure JavaScript doesn't have classes and constructor functions are the closest to classes language elements. Ext JS extends the JavaScript language and introduces classes and a special way to define and instantiate them with functions `Ext.define()` and `Ext.create()`. Ext JS also allows to extend one class from another using the property `extend` and define class constructors using the property `constructor`. With `Ext.define()` you declare a class declaration, and `Ext.create()` instantiates it. Basically, `define()` serves as a template for creation of one or more instances.

Usually the first argument you specify to `define()` is a fully qualified class name, the second argument is an object literal that contains the class definition. If you use `null` as the first argument Ext JS creates an anonymous class.

The next class `Header` has 200 pixel height, uses the  `hbox` layout, has a custom `config` property `logo`, extends `Ext.panel.Panel`:

```
Ext.define("SSC.view.Header", {
    extend: 'Ext.panel.Panel',
    title: 'Test',
    height: 200,
    renderTo: 'content',           // ①
    config: {
        logo: 'sony_main.png'   // ②
    },
    layout: {
        type: 'hbox',
        align: 'middle'
    }
});
```

- ① Render this panel to an HTML element with `id=content`.
- ② Defining a custom `config` property `logo`.

You can optionally include a third argument for `define()`, which is a function to be called when the class definition is created. Now you can create one or more instances of the class class, for example:

```
var myHeader = Ext.create("SSC.view.Header");
```

The values of custom config properties from the `config{}` section of the class can be reassigned during the class instantiation. For example, the next code snippet will print `sony.png` for the first instance of the header, and `sony_small.png` for the second one. Please note that Ext JS automatically generated getters and setters for all `config` properties, which allowed us to use the method `getLogo()`.

```
Ext.onReady(function () {
    var myHeader1 = Ext.create("SSC.view.Header");
    //
    var myHeader2 = Ext.create("SSC.view.Header",
        { logo: 'sony_small.png' });

    console.log(myHeader1.getLogo());
    console.log(myHeader2.getLogo());
});
```



Don't forget about an online tool JSFiddle that allows you to test and share JavaScript code snippets. JSFiddle knows about Ext JS 4.2 already. For example you can run the code snippet above by following this [JSFiddle link](#). If it doesn't render the styles properly, check the URL of the ext-all.css in the section External Resources.

If a class has dependencies on other classes, which must be preloaded, use the `requires` parameter. For example, the next code snippet shows that the class `SSC.view.Viewport` requires the `Panel` and the `Column` classes. So the Ext JS loader will check if `Panel` and/or `Column` was not loaded yet, it'll dynamically lazy-load them first.

```
Ext.define('SSC.view.Viewport', {
    extend: 'Ext.container.Viewport',
    requires: [
        'Ext.tab.Panel',
        'Ext.layout.container.Column'
    ]
    // the rest of the class definition is omitted
});
```

`Ext.create()` is a preferred way of instantiation as it does more than the `new` operator that is also allowed in Ext JS. But `Ext.create()` may perform some additional functionality, for example if `Ext.Loader` is enabled, `create()` will attempt to synchronously load dependencies (if you haven't used the option `require`). But with `requires` your preloads all dependencies asynchronously in parallel and is a preferred way of specifying dependencies. Besides, the `async` mode allows loading from different domains, while `sync` loading doesn't.



Ed Spencer published a useful list of recommendations on improving performance of Ext JS applications in his blog titled [SenchaCon 2013: Ext JS Performance Tips](#).

## Dynamic Class Loading

The singleton `Ext.Loader` offers a powerful mechanism of dynamic loading of any classes on demand. You have to explicitly enable the loader first thing after including the Ext JS framework in your HTML file it providing the paths where the loaded should look for files, for example:

```
<script type="text/javascript">
    Ext.Loader.setConfig({
        enabled: true,
        disableCaching: false,
```

```
    paths: {
        'SSC': 'my_app_path'
    });
</script>
```

Then the manual loading of a class can be done using Ext.require('SSC.SomeClass') or Ext.syncRe

For each class Ext JS creates one instance of special class Ext.Class, which will be shared by all objects instantiated from this class.



The instance of any object has access to its class via a special variable **self**.

Prior to creating a class, Ext JS will run some pre-processors and some post-processors based on the class definition. For example, the class 'SSC.view.Viewport' from the code sample above uses `extend: 'Ext.container.Viewport'`, which will engage the *extend* pre-processor that will do some background work to properly build a subclass of `viewport`. If your class includes the `config` section, the *config* preprocessor will be engaged.

### Xtype: An Efficient Way to Create Class Instances

One of the interesting pre-processors is *xtype*, which is an alternative to the invocation of the `create()` method for creating the instance of the class. Every Ext JS component has assigned and documented `xtype`. For example, `Ext.panel.Panel` has an `xtype` of `panel`. Online documentation displays the name of the corresponding `xtype` in the header of each component as in [Figure 4-4](#).

A screenshot of the ExtJS API documentation for the 'Ext.panel.Panel' class. The page title is 'Ext.panel.Panel' with the 'xtype: panel' suffix. Below the title, there are four navigation tabs: 'Configs 116', 'Properties 17', 'Methods 177', and 'Events 45'. Each tab has a small blue icon next to it: a gear for 'Configs', a document for 'Properties', a cube for 'Methods', and a lightning bolt for 'Events'.

Figure 4-4. Each component has an xtype

Using `xtype` instead of `create()` leads to more efficient memory management. If the object is declared with the `xtype` attribute, it won't be instantiated until some container uses it. You are encouraged to assign `xtype` to your custom classes, and Ext JS will instantiate it for you without the need to call `create()`. You can find many examples of using the `xtype` property in the section "Developing Save The Child with Ext JS" later in this chapter. For example, the following class definition includes many components with the `xtype` property.

```
Ext.define("SSC.view.LoginBox", {
    extend: 'Ext.Container',
    xtype: 'loginbox',

    layout: 'hbox',

    items: [
        {
            xtype: 'container',
            flex: 1
        }, {
            xtype: 'textfield',
            emptyText: 'username',
            name: 'username',
            hidden: true
        }, {
            xtype: 'textfield',
            emptyText: 'password',
            inputType: 'password',
            name: 'password',
            hidden: true
        }, {
            xtype: 'button',
            text: 'Login',
            action: 'login'
        }
    ]
});
```

Most of the above components use the standard Ext JS `xtype` values, so the fact that you have included them into the class `SSC.view.LoginBox` is a command for Ext JS to instantiate all these buttons and text fields. But the class `SSC.view.LoginBox` also includes `xtype: 'loginbox'` - we decided to assign the value `loginbox` to serve as the `xtype` of our class. Now, you can use the statement `xtype: 'loginbox'` in any other container, it'll know how to instantiate it. For example, later in this chapter you'll see the complete code of the main window `SSC.view.ViewPort`, which includes (and instantiates) our login box as follows:

```
items: [
    xtype: 'loginbox',
    margin: '10 0 0 0'
],
```

```
// more items go here  
]
```

## Multiple Inheritance With Mixins

Object-oriented languages Java and C# can be considered as simpler version of C++. One of the C++ features that didn't make it into Java and C# was support of multiple inheritance: in these languages a class can extend only one other class. This was done for a good reason - debugging of the C++ programs that were written with multiple inheritance was difficult.

Ext JS supports multiple inheritance via JavaScript mixins. A class constructor can get any object as an argument, and Ext JS will use its property values to initialize the corresponding properties defined in the class, if they exist, and the rest of the properties will be created on the fly. The following code snippet shows how to define a `classB` that will have features defined in classes `classA`, `classC`, and `classD`.

```
Ext.define("MyApp.classB", {  
    extend: "MyApp.classA",  
    mixins: {classC: "MyApp.ClassC"  
             classD, "MyApp.classD"}  
  
}  
...  
});
```



If more than one mixin has a method with the same name, the first method that was applied to the resulting class wins. To avoid collisions Ext JS allows you to provide fully qualified name of the method, for example `this.mixins.classC.conflictingName();`  
`this.mixins.classD.conflictingName();`.

## MVC in Ext JS

While Ext JS doesn't force you to architect your application based on the MVC paradigm, it's a really good idea to do so. Earlier in the section on Sencha CMD you've seen how this tool generates a project, which separates model, views, controllers and stores into separate directories as in [Figure 4-1](#) that depicted the structure of the Hello World project. But later in this chapter we'll build our Save The Child application the same way. [Figure 4-5](#) presents a diagram illustration how the Ext JS application that contains all Model-View-Controller tiers.

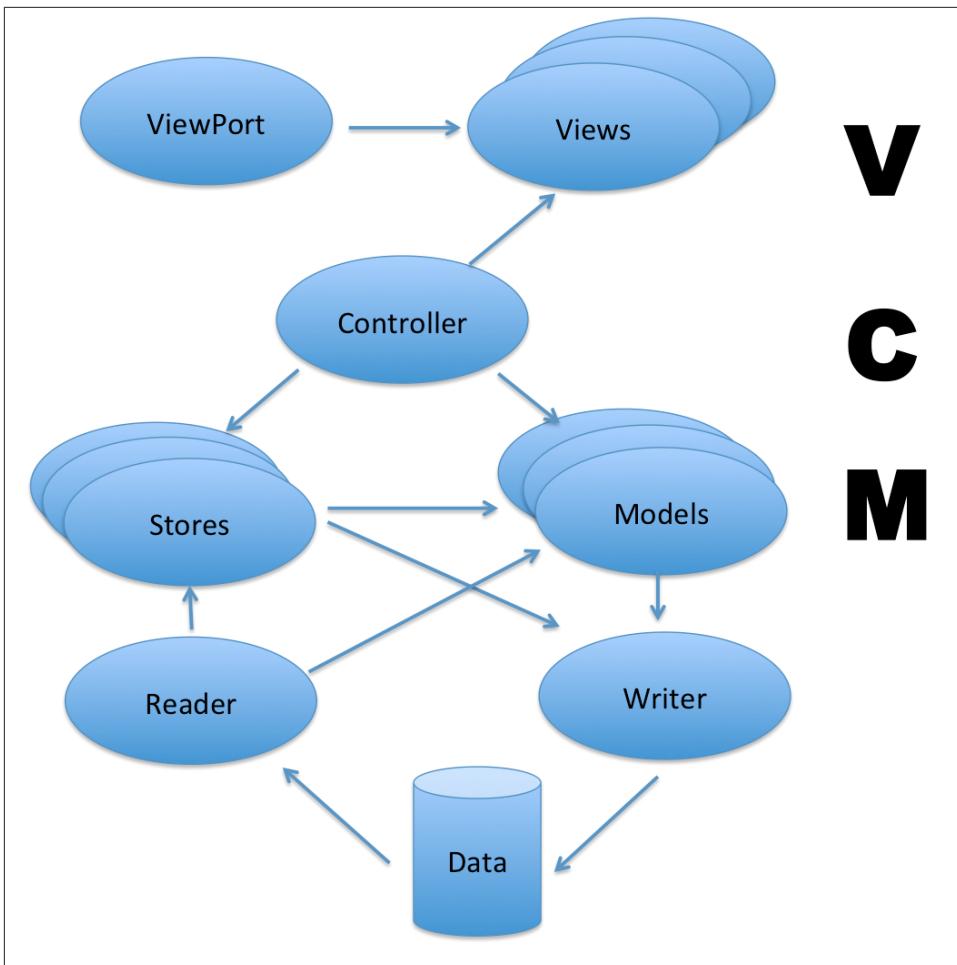


Figure 4-5. Model-View-Controller in Ext JS

- Controller is an object that serves as an intermediary between the data and the views. The data has arrived to your application, and controller has to notify the appropriate view. The user changed the data on the view - the controller should pass the changes to the model (or stores in the Ext JS world). Controller is the place to write event listeners reaction to some important events of your application (e.g. a user clicked on the button). In other words, Controller maps the events to actions to be performed on the data or the view.
- View is a certain portion of the UI that the user sees. The view is populated with the data from the model (or stores).

- Model represents some business entity, e.g. Donor, Campaign, Customer, Order etc. In Ext JS models are access via stores.
- Store contains one or more model instances. Typically, a Model is a separate class that is instantiated by the store object, but in simple cases a store can have the model data embedded in its own class. A store may use more than one model if need be. Both stores and model can communicate with the data feed that in a Web application is usually provided by some server-side data feed.

The application object defines its controllers, views, models, and stores. When the Save The Child will be ready, the code of its app.js will look as follows:

```
Ext.application({
    name: 'SSC',

    views: [
        'CampaignsMap',
        'DonateForm',
        'DonorsPanel',
        'Header',
        'LoginBox',
        'VideoPanel',
        'Viewport'
    ],

    stores: [
        'Campaigns',
        'Donors'
    ],

    controllers: [
        'Donate'
    ]
});
```

The above code is a clean and simple to read/write code helps Ext JS framework in generating additional code required for wiring views, models, controllers and stores together. There is no explicit `models` section, because in our implementation the models were defined inside the `stores`. For better understanding of the rest of this chapter you should read the [MVC Architecture](#) section from Ext JS documentation. We don't want to repeat the content of Sencha product documentation, but rather will be giving you brief descriptions while doing code review of the Save The Child application.

## Models and Stores

When you create a class to be served as a model, it must be a subclass of `Ext.data.Model`. A Model has the `fields` property. For example, this is how the you can represent a Donor entity using just two fields: name and location:

```

Ext.define('HR.model.Donor',{
    extend: 'Ext.data.Model',
    requires: [
        'Ext.data.Types'
    ],
    fields: [
        { name: 'donors', type: Ext.data.Types.INT },
        { name: 'location', type: Ext.data.Types.STRING}
    ]
});

```

Think of an instance of a model is a of one record representing some business entity, e.g. Donor. Ext JS generate getters and setters for models, so if an instance of the model is represented by a variable `sscDonor`, you can set or get its value as follows:

```

sscDonor.set('name', 'Farata Systems');
var donorName= sscDonor.get('name');

```

A store in Ext JS holds a collection of instances of some model. For example, if you the application has retrieved the information about ten donors, it'll be represented in Ext JS as a collection of ten instances of the class `Donor`. A custom store in your application has to extend from the class `Ext.data.Store`.

If you need to quickly create a mock store for testing purposes, you can declare a store with inline data that you can specify using the config option `data`. The next code sample shows a declaration of the store for providing the information about the donors as inline data:

```

Ext.define('SSC.store.Donors', {
    extend: 'Ext.data.Store',
    fields: [
        { name: 'donors', type: 'int' },
        { name: 'location', type: 'string' }
    ],
    data: [
        { donors: 48, location: 'Chicago, IL' },
        { donors: 60, location: 'New York, NY' },
        { donors: 90, location: 'Dallas, TX' }
    ]
});

```

It's a good idea to have a mock store with the test data located right on your computer. This way you won't depend on the readiness and availability of the server-side data. But usually, a store makes some AJAX call to a server and retrieves the data via the object `Ext.data.reader.Reader` or one of its descendants, for example:

```

Ext.define('SSC.store.Donors', {
    extend: 'Ext.data.Store',

```

```

model: 'SSC.model.Donor',           // ①
proxy: {                           // ②
    type: 'ajax',
    url: 'donors.json',           // ③
    reader: {                     // ④
        type: 'json'
    }
}
});

```

- ① The model `SSC.model.Donor` has to be described in your application as a separate class and contain only the fields defined, no data.
- ② Unless you need to load some raw data from a third-party server provider, wrap your reader into a [Proxy object](#). Server proxies are used for implementing CRUD operations and include the corresponding methods - `create()`, `read()`, `update()`, `destroy()`.
- ③ For the mockup mode, we use json-formatted file that contains an array of object literals (each object represents one donor). The `donors.json` should look like the content of the `data` attribute in the code of `SSC.store.Donors` above.
- ④ The Reader object will consume JSON. Read the [Ext JS documentation](#) to decide how to properly configure your JSON reader. The reader knows how to convert the data into the model.

Populating of a store with the external data is usually done via a `Proxy` object, and Ext JS offers several server side proxies: `Ajax`, `JsonP`, `Rest`, and `Direct`. To retrieve the data from the server you'd be calling the method `load()` on your `Store` object. To send the data to the server - call the method `sync()`.

The most frequently used proxy is `Ajax`, which uses `XMLHttpRequest` to communicate with the server. The code fragment below shows another way of defining the store `Donors`. It specifies via the config `api` the server sides URIs responsible for the four CRUD operations. We've omitted the `reader` section here because the default data type is JSON anyway. Since we've specified the URIs for the CRUD operations, there is no need to specify the `url` attribute as in the above code sample.

```

Ext.define('SSC.store.Donors', {
    extend: 'Ext.data.Store',

    model: 'SSC.model.Donor',
    proxy: {
        type: 'ajax',
        api: {
            create: '/create_donors',
            read: '/read_donors',
            update: '/update_donors',

```

```

        destroy: '/destroy_donors'
    }
});

```

When you create an instance of the data store you can specify the `autoload` parameter. If it's `true`, the store will be populated the store automatically. Otherwise, explicitly call the method `load()` whenever the data retrieval is needed. For example, you can call the method `myStore.load({callback:someCallback})` passing it some callback to be executed.



In Appendix B we discuss HTML5 local storage API. Ext JS has a class `Ext.data.proxy.LocalStorage` that saves the model data locally if the Web browser supports it.

## Controllers and Views

Your application controller is a liaison between the data and the views. This class has to extend `Ext.app.Controller`, and will include references to the views and, possibly stores. Controller will automatically load every class mentioned in its code, create an instance of each store and register each instance with the class [`Ext.StoreManager`].

A controller class has config properties `stores`, `models`, and `views`, where you can list stores, models, and views that controller should know about. For example, the next code listing shows the controller `SSC.controller.Donate` includes the names of two stores - `SSC.store.Campaigns` and `SSC.store.Donors`.

```

Ext.define('SSC.controller.Donate', {
    extend: 'Ext.app.Controller',
    stores: ['SSC.store.Campaigns', 'SSC.store.Donors'] // ❶

    refs: [{                                         // ❷
        ref: 'donatePanel',
        selector: '[cls=donate-panel]'
    }
    // more views can go here
    ],

    init: function () {                           // ❸

        this.control({
            'button[action=showform]': {
                click: this.showDonateForm
            }
            // more event listeners go here
        });
    },
});

```

```

        showDonateForm: function () {
            // ④
            this.getDonatePanel().getLayout().setActiveItem(1);
        }
    );
}

```

- ❶ Listing stores in your controller. Actually, in most cases you'd list stores in the Ext.application singleton as we did earlier. But if you need to dynamically create controllers, you don't have a choice but declare stores in such controllers.
- ❷ Listing one or more views of your application in the refs property, which simplifies the search of the component globally or within some container. Controller generates getters and setters for each object listed in the refs.
- ❸ Registering event listeners in the function init(). In this case we're registering the event handler function showDonateForm that will process clicks on the button, which has an attribute action=showform.
- ❹ The getter getDonatePanel() will be auto-generated by Ext JS because donatePanel was included in the refs section.

Ext.StoreManager provides a convenience method to look up the store by store ID. If stores were automatically injected into Ext.StoreManager by the controller, the default store ID is its name, e.g. SSC.store.Donors:

```

var donorsStore = Ext.data.StoreManager.lookup('SSC.store.Donors');

// An alternative syntax to use StoreManager lookup
var donorsStore = Ext.getStore('SSC.store.Donors');

```

The above SSC.controller.Donate doesn't use the config properties views, but if it did, Ext JS would generate getters and setters for every view (the same is true for stores and models). It uses refs instead to reference components, and getters and setters will be generated for each components listed in refs, e.g. getDonatePanel(). Lookup of such components is done based on the value in selector using the syntax compatible with ComponentQuery. The difference between refs and the config property views is that the former generates references to instances of specific components from views, while the latter generates getters and setters only to the "class" (not the instance) of the entire view for further instance creation.



You can view and test Ext JS components against bundled themes browsing the Theme Viewer at the [Ext JS 4.2 Examples](#) page.

## A Component's Lifecycle

In previous versions of our Save The Child application CSS was responsible for all layouts of the UI components. In Chapter 10 you'll be learning about the *responsive design* techniques and CSS media queries, which allow to create fluid layouts that automatically adjust to the size of the viewport. But this section is about Ext JS proprietary way of creating and adding UI components to Web pages. Before the user will see a component, Ext JS framework will go through the following phases for each component:

- Load - load the required (or all) Ext JS classes and their dependencies
- Initialization of components when the DOM is ready
- Layout - measuring and assigning sizes
- Rendering - convert components to HTML elements
- Destruction - removing the reference from DOM, removing event listeners and unregistering from the component manager.

Rendering and layout are the most time consuming phases. The rendering does a lot of preparations to give the browser's rendering engine HTML elements and not Ext JS classes. The layout phase is slow because the calculation of sizes and positions (unless they are in absolute coordinates) and applying of cascading stylesheets takes time.

There's also the issue of *refflows*, which happen when the code reads-measures-writes to the DOM and makes dynamic style modifications. Fortunately, Ext JS 4.1 was redesigned to minimize the number of reflows; now a large portion of recalculations is done in a batch before modifying the DOM.

### Components as Containers

If a component can contain other components, it's a container (e.g. `Ext.panel.Panel`) and will have `Ext.container.Container` as one of its ancestors. In Ext JS class hierarchy, `Container` is a subclass of `Component`, so all methods and properties defined for a component are available for a container too. Each Web page consists of one or more containers, which include some children - components (in Ext JS they are subclasses of `Ext.Component`), for example, `Ext.button.Button`.

You'll be defining your container class with as a subclass of a container by including `extend: Ext.container.Container`. The child elements of a container are accessible via its property `items`. In the `Ext.define()` statement of the container you may specify the code that will loop through this `items` array and, say style the components, but actual instances of the children will be provided during the `Ext.create()` call via configuration object.

The process of adding a component to a container will typically consist of invoking `Ext.create()` and specifying in a configuration object where to render the component to, for example `renderTo: Ext.getBody()`.

But under the hood Ext JS will do a lot more work. The framework will auto-generate a unique ID for the component, assign some event listeners, instantiate component plugins if specified, invoke the `initComponent()`, and add the component to `Ext.ComponentManager`.



Even though you can manually assign an ID to the component via configuration object, it's not recommended because it could result in duplicate IDs.

## Events

Events in Ext JS are defined in the mixin `Ext.util.Observable`. Components interested in receiving events can subscribe to them using one of the following methods:

- By calling the method `addListener()`
- By using the method `on()`.
- Declaratively

The next code snippet shows two different ways of how a combobox can subscribe to the event `change`. The handler function is a callback that will be invoked if the event `change` will be dispatched on this combobox:

```
combobox.addListener('change', myEventHandlerFunction);

combobox.on('change', myEventHandlerFunction);
```

To unsubscribe from the event call the method `removeListener()` or its shorter version `un()`:

```
combobox.removeListener('change', myEventHandlerFunction);
combobox.un('change', myEventHandlerFunction);
```

You can also declaratively subscribe to events using the `listeners` config property of the component:

```
Ext.create('Ext.button.Button', {
    listeners: {
        click: function() { // handle event here }
    }
})
```

JavaScript support event bubbling (see Appendix A). In Ext JS event bubbling mechanism enables events dispatched by components that include `Ext.util.Observable` bubble up through all enclosing containers. For components it means that you can handle component's event on container level. It can be handy to subscribe and handle multiple similar events in one place. To enable bubbling for selected events use the `enableBubble()` method, for example:

```
this.enableBubble(['textchange', 'validitychange']);
```

To define custom events use the method `addEvents()`, where you can provide one or more of the custom event names:

```
this.addEvents('messagesent', 'updatecompleted');
```

For components you have to define custom events inside the `initComponent()` method. For controllers - inside `init()`, and for any other class – inside its constructor.

## Layouts

The container's property `layout` controls how its children are laid out. It does it by referring to the container's property `items`, which lists all of the child components. If you won't explicitly set the `layout` property, its default value is `Auto`, which is just placing components inside the container top to bottom regardless of the component size.

Usually you'll be explicitly specifying the layout. For example, the  `hbox` layout would arrange all components inside the container horizontally next to each other, but `vbox` layout would arrange them vertically. The `card` layout places the components one under another, but only the top component is visible (think of a tab folder, where the content of only one tab is visible at any given time).

The `border` layout is often used to arrange the components in the main viewport (a.k.a. home page) of your application. This layout allows you to split the container's real estate into five imaginary regions: `north`, `east`, `west`, `south`, and `center`. If you need to allocate the top menu items, place them to the region `north`. The footer of the page is in the `south` as shown in the code sample below.

```
Ext.define('MyApp.view.Viewport', {
    extend: 'Ext.container.Viewport',
    layout: 'border',
    items: [
        {
            width: 980,
            height: 200,
            title: "Top Menu",
            region: "north",
            xtype: "panel"
        }
    ]
});
```

```

        width: 980,
        height: 600,
        title: "Page Content",
        region: "center",
        xtype: "panel"},
    ],
{
    width: 980,
    height: 100,
    title: "The footer",
    region: "south",
    xtype: "panel"}],
});

```

## Setting Proportional Layouts With The `flex` Property

Ext JS has a property `flex` that allows make your layout more flexible. Instead of specifying the width or height of a child component in absolute values you can split the available space proportionally. For example if the space has to be divided between two components having the `flex` values 2 and 1, this means that the 2/3 of the container's space will be allocated to the first component, and 1/3 to the second one as illustrated in the following code snippet.

```

layout: 'vbox',
items: [{{
    xtype: 'component',
    html: 'Lorem ipsum dolor',
    flex: 2
},
{
    xtype: 'button',
    action: 'showform',
    text: 'DONATE NOW',
    flex: 1
}]

```



The format of this book doesn't allow us to include detailed description of major Ext JS component. If you are planning to use Ext JS for development of enterprise Web applications, allocate some extra time to learn the data grid `Ext.grid.Panel` that's used for rendering of tabular data. You should also master working with forms with `Ext.form.Panel`.

In the next section you'll see Ext JS layouts in action while working on the Save The Child application.

# Developing Save The Child With Ext JS

In this section we'll do a code walk-through of the Ext JS version of our Save The Child application. Ext JS framework is often used in the enterprise applications that communicate with Java-based server-side. The most popular IDE among Java enterprise developers is called Eclipse. That's why we decided to switch from WebStorm to Eclipse IDE for some time. Apache Tomcat is one of the most popular servers among Java developers.

We've prepared two separate Eclipse projects:

- SSC\_Top\_ExJS contains the code required to render the top portion of the UI
- SSC\_Complete\_ExtJS contains the complete version.

To test these applications in Eclipse, you need to install its version titled “Eclipse IDE for Java EE developers” and configure it with Apache Tomcat as described below.

Note: If you are not planning to work with Java servers, you can continue using WebStorm IDE. Just open in WebStorm the WebContent directory from the above project (as you did in the previous chapters) and open the index.html file in the browser. WebStorm will run these Web application using its internal Web server.



To make WebStorm work faster, exclude directories *ext*, *packages*, *build*, and *WEB-INF* from the project (hit the icon with a wrench image on the toolbar and select the Directories and the Excluded). This way WebStorm won't be indexing these directories.

## Setting Up Eclipse IDE and Apache Tomcat

Eclipse is the most popular IDE among Java developers. You can use it for developing JavaScript too although this would not be the best choice. But we'll need it to demonstrate the HTML/Java application generation in the next chapter hence let's set it up.



Sencha offers Eclipse plugin (not covered in the book) for those who purchased a license of Sencha Complete.

We'll use the version “Eclipse IDE for Java EE developers”. It's available free of charge at [Eclipse Downloads site](#). The installation comes down to unzipping of the downloaded archive. Then double-click on the Eclipse executable to start this IDE.

## Apache Tomcat

In Chapter 3 we used a XAMPP server that was running PHP scripts. Since this chapter will include server-side code written in Java, we'll use **Apache Tomcat**, which is one of the **popular** servers used by Java developers for deploying Web applications. Besides being a Web Server, Tomcat also contains Java Servlet container that will be used later in the next chapter in the section "Generating CRUD applications". But for most examples we'll use Tomcat as a Web server where Ext JS code will be deployed.

Get the latest version of Apache Tomcat from the Download section at <http://tomcat.apache.org>. At the time of this writing Tomcat 7 is the latest production-quality build, so download the zip file with the Tomcat's Binary Distributions (Core). Unzip the file in the directory of your choice.

Even though you can start Tomcat from a separate command window, the more productive way is to configure Tomcat right in the Eclipse IDE. This will allow to deploy your applications, and start/stop Tomcat without the need to leave Eclipse. To add a server to Eclipse, open Eclipse Java EE perspective (menu Window | Open Perspective), select the menu File | New | Other | Server | Server | Apache | Tomcat v7.0 Server, select your Tomcat installation directory and press Finish. If you don't see Tomcat 7 in the list of Apache servers, click on "Download additional server adapters".

You'll see the Tomcat entry in the Eclipse Project Explorer. Go to Eclipse menu Windows | Show View and open the Servers view. Start Tomcat using the right-click menu.



By default, Eclipse IDE keeps all required server configuration and deployment files in its own hidden directory. To see where exactly they are located in your computer, just double-click on the name of Tomcat in the Server view. The server path field contains the path. Keep in mind that while Tomcat documentation defines *webapps* as a default deployment directory, Eclipse uses *wtpwebapps* directory instead. If you prefer to deploy your Eclipse projects under your original Tomcat installation path, select the option Use Tomcat Installation.

In the next section you'll learn how to create Dynamic Web Projects in Eclipse, where you'll need to specify the Target Runtime for deployment of your Web applications. This newly installed and configured Tomcat server will serve as a deployment target for our sample projects.

## Dynamic Web Projects and Ext JS

Eclipse for Java EE developers comes with **Web Tools Platform** that simplifies development of Web applications by allowing you to create so-called Dynamic Web Project. This is an Eclipse pre-configured project that already knows where its Java server located

and deployment to the server is greatly simplified. Sample projects from this chapter will be specifically created for deployment under Apache Tomcat server.

To create such a project select Eclipse menu File | New | Other | Web | Dynamic Web Project. It'll pop up a window similar to [Figure 4-6](#). Note that the Target Runtime is Apache Tomcat v7.0 that we've configured in the previous section.

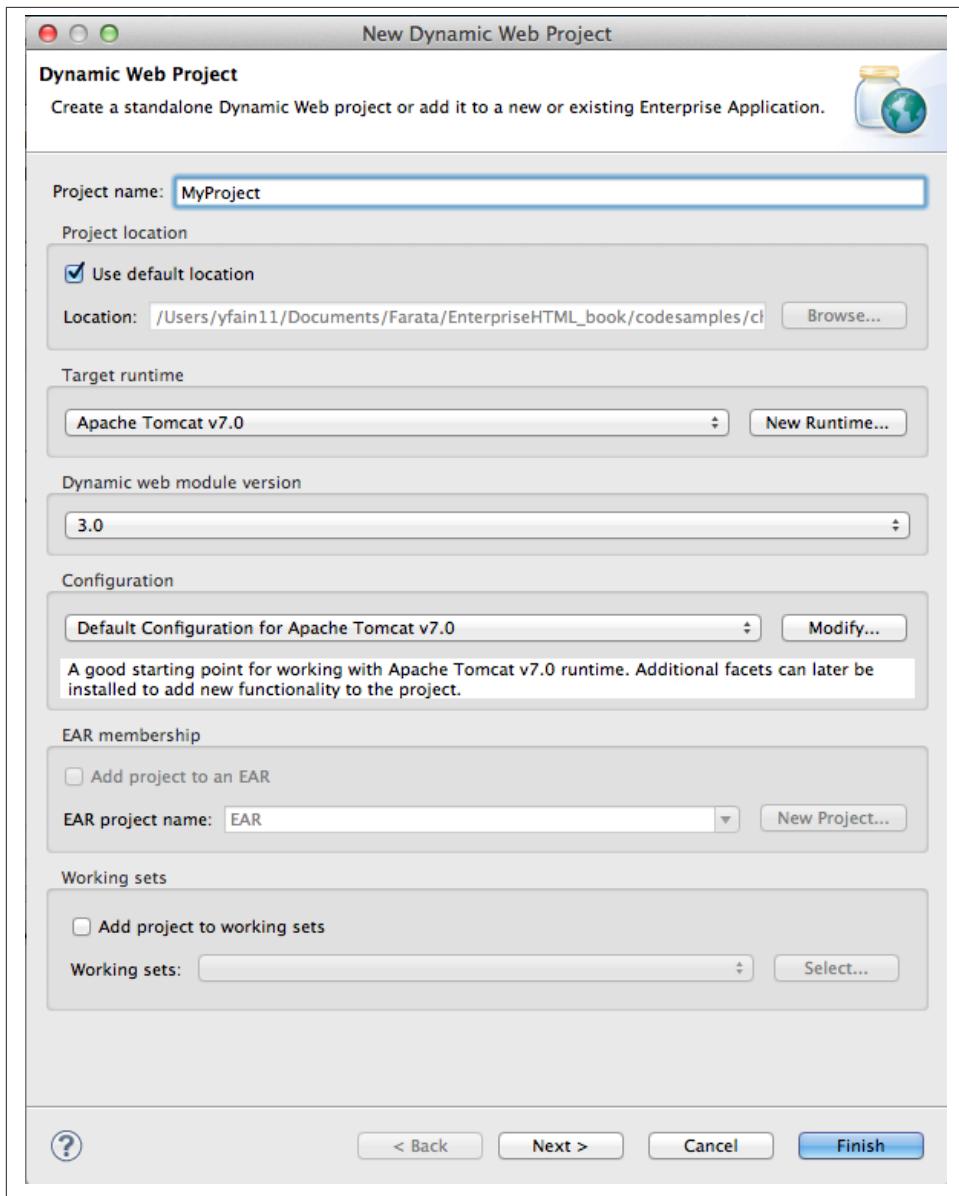


Figure 4-6. Creating Dynamic Web Project in Eclipse

Upon creation, this project will include several directories, and one of them will be called *WebContent*. This directory it serves as a document root of the Web server in Eclipse Dymamic Web Projects . This is the place to put your index.html and one of possible places to keep the Ext JS framework.Create a subdirectory *ext* under *WebContent* and

copy there all files from the Ext JS distribution. The *app* directory should also go under *WebContent*.

Unfortunately, Eclipse IDE is infamous for slow indexing of JavaScript files, and given the fact that Ext JS has hundreds of JavaScript files, your work may be interrupted by Eclipse trying to unnecessary re-validate these files. Developers of Sencha Eclipse plugin decided to solve this problem by creating a special type library file (*ext.ser*) supporting code assistance in Eclipse. This solution will work until some of the Ext JS API changes, after that Sencha should update the type library file.

If you don't have Sencha Eclipse plugin, there is a couple of solutions to this problem (we'll use the first one).

1. Exclude from Eclipse build the following Ext JS directories: ext, build, and packages.
2. Don't copy the Ext JS framework into your Eclipse project. Keep it in the place known for Tomcat, and configure as a loadable module.

To implement the first solution, right click on the properties of your project and select **JavaScript | Include Path**. Then switch to the **Source** tab, expand the project's **Web content** and press the buttons **Edit** and then **Add**. One by one add the ext, build, and packages as exclusion patterns (add the slash at the end) as shown in **Figure 4-7**

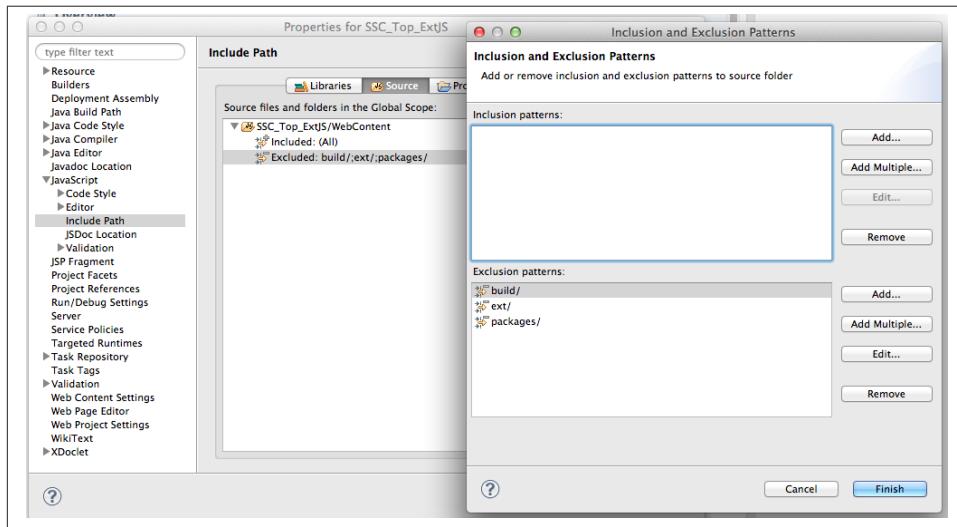
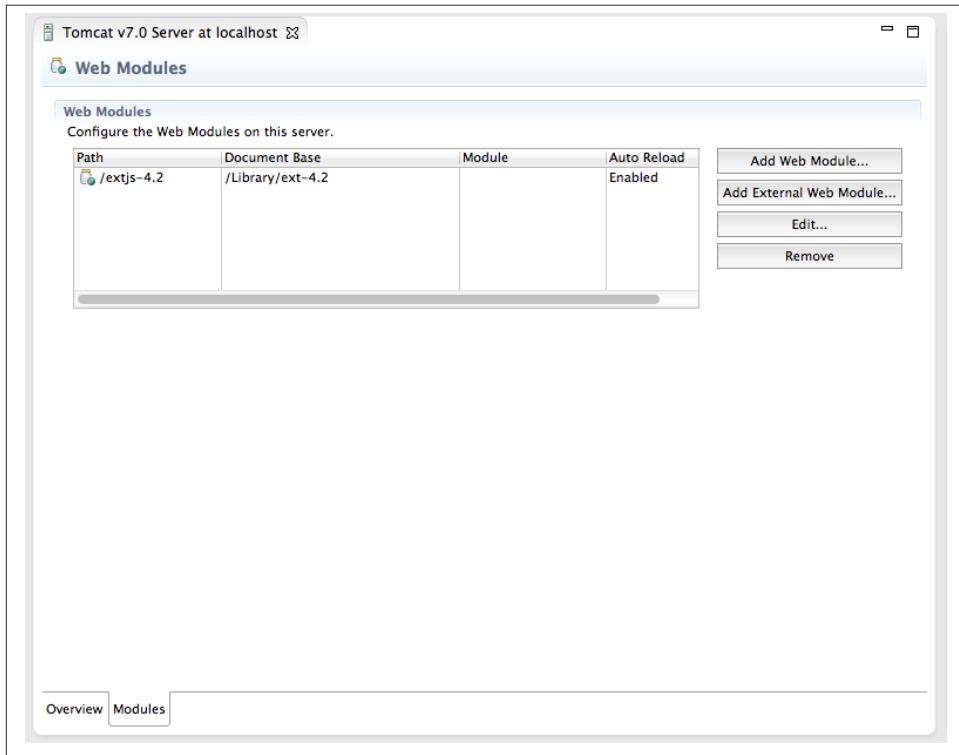


Figure 4-7. Solution 1: Excluding folders in Eclipse

For the second solution, you'll need to add your Ext JS folder as a static Tomcat module. Double-click at the Tomcat name in the Servers view and then click on the bottom tab Modules. Then Click on Add External Web Module. In the popup window find the

folder where your Ext JS is (in my computer it's inside the Library folder as in [Figure 4-8](#)) and give it a name (e.g. /extjs-4.2). Now Tomcat will know that on each start it has to load year another static Web module known as /extjs-4.2. If you're interested in details of such deployment, open up the file server.xml located in your Eclipse workspace in the hidden directory `.metadata/.plugins/org.eclipse.wst.server.core/tmp0/conf`.

To ensure that you did everything right, just enter in your browser the URL <http://localhost:8080/extjs-4.2>, and you should see the welcome screen of Ext JS.



*Figure 4-8. Solution 2: Adding Ext JS to Tomcat as a static module*

In both of these solutions you'll lose the Ext JS context sensitive help, but at least you will eliminate the long pauses caused by Eclipse internal indexing processes. Developing with ExtJS in WebStorm IDE or IntelliJ IDEA IDEs would spare you from all these issues because these IDE's are smart enough to produce context-sensitive help from an external JavaScript library.



If you decided to stick to WebStorm IDE, you can skip Eclipse-related instructions below and just open in browser index.html located in the WebContent directory of the SSC\_Top\_ExtJS project. In any case the browser will render the page that looks as in [Figure 4-10](#) below.

In this section we brought together three pieces of software: Eclipse IDE, Apache Tomcat server, and Ext JS framework. Let's bring one more program to the mix: Sencha CMD. We already went through the initial code generation of Ext JS applications. If you already have a Dynamic Web Project in Eclipse workspace, run Sencha CMD specifying the *WebContent* directory of your project as the output folder, where the generated project will reside. For example, if the name of your Dynamic Web Project is hello2, the Sencha CMD command can look as follows:

```
sencha -sdk /Library/ext-4.2 generate app HelloWorld /Users/yfain11/myEclipseWork-space/hello2/WebContent
```

## Save The Child: The Top Portion of UI

To run the top portion of the UI, Select Eclipse menu File | Import | General | Existing projects into workspace and press the button Next. Then select the option *Select root directory* and press browse to find SSC\_Top\_ExJS on your disk. This will import the entire Dynamic Web Project, and most likely you'll see one error in the Problems view indicating that the target runtime with so-and-so name is not defined. This may happen because the name of the Tomcat configuration in your Eclipse is different from the one in the directory SSC\_Top\_ExJS.

To fix this issue, right-click on the project name and select the menu Properties | Targeted runtimes. Then uncheck the Tomcat name that was imported from our archive and check the name of your Tomcat configuration. This action will make the project SSC\_Top\_ExtJS deployable under your Tomcat server. Right-click on the server name in the Servers view and select Add and Remove menu item. You'll see a popup window similar to [Figure 4-9](#), which depicts a state when the project SSC\_Top\_ExtJS is configured (deployed), but SSC\_Complete\_ExtJS isn't yet.

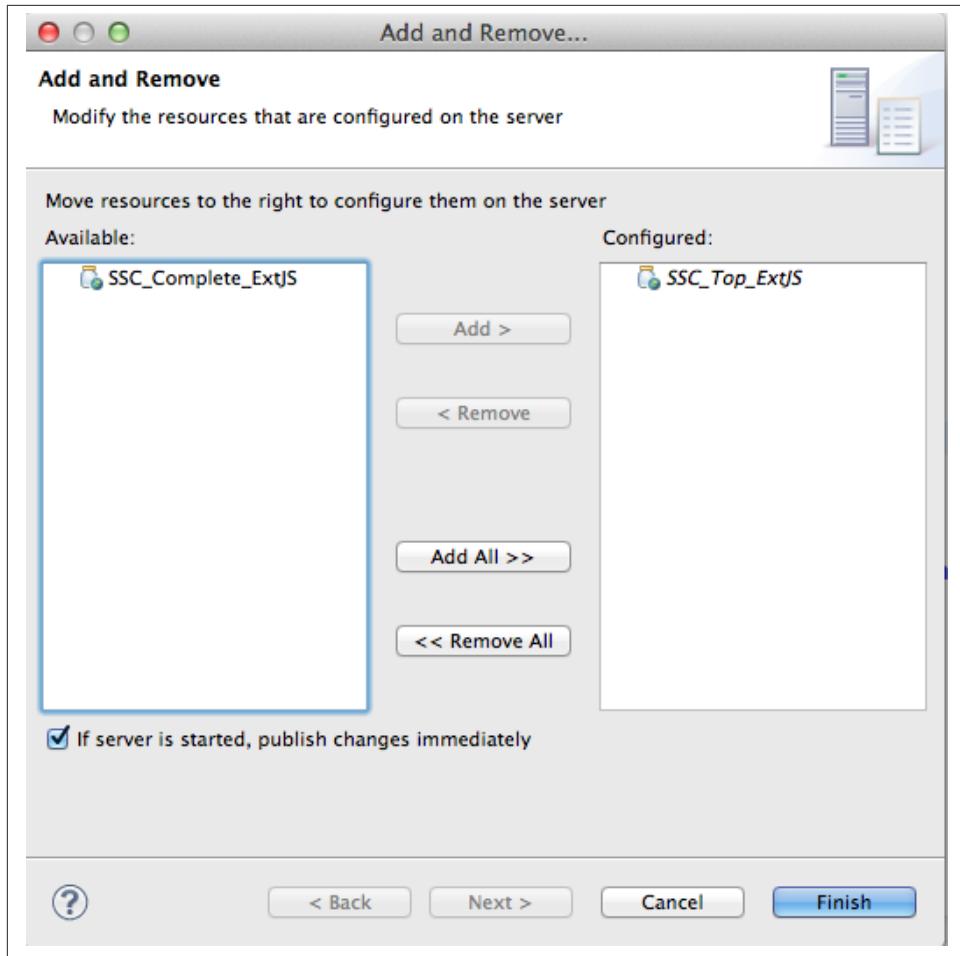


Figure 4-9. Deploying Dynamic Web Project

Right-click on the project name `SSC_Top_ExtJS`, select the menu `Run as | Run on server`. Eclipse may offer to restart the server - accept it, and you'll see the top portion of the `Save The Child` application running in the internal browser of Eclipse that will look as shown on [Figure 4-10](#). You can either configure Eclipse to use your system browser or just enter the URL [http://localhost:8080/SSC\\_Top\\_ExtJS/](http://localhost:8080/SSC_Top_ExtJS/) in the browser of your choice - the Web page will look the same.

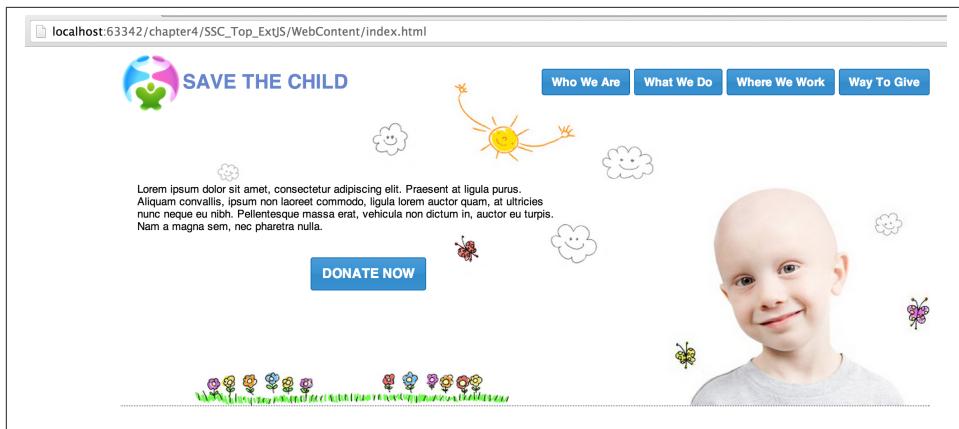


Figure 4-10. Running the SSC\_Top\_ExtJS



Apache Tomcat runs on the port 8080 by default. If you want to change the port number, double-click on the Tomcat name in the Servers view and change the port there.

It's time for a code review. The initial application was generated by Sencha CMD so the directory structure complies with the MVC paradigm. This version has one controller `Donate.js` and three views: `DonateForm.js`, `Viewport.js`, and `Header.js` as shown in [Figure 4-11](#). The images are located under the folder `resources`.

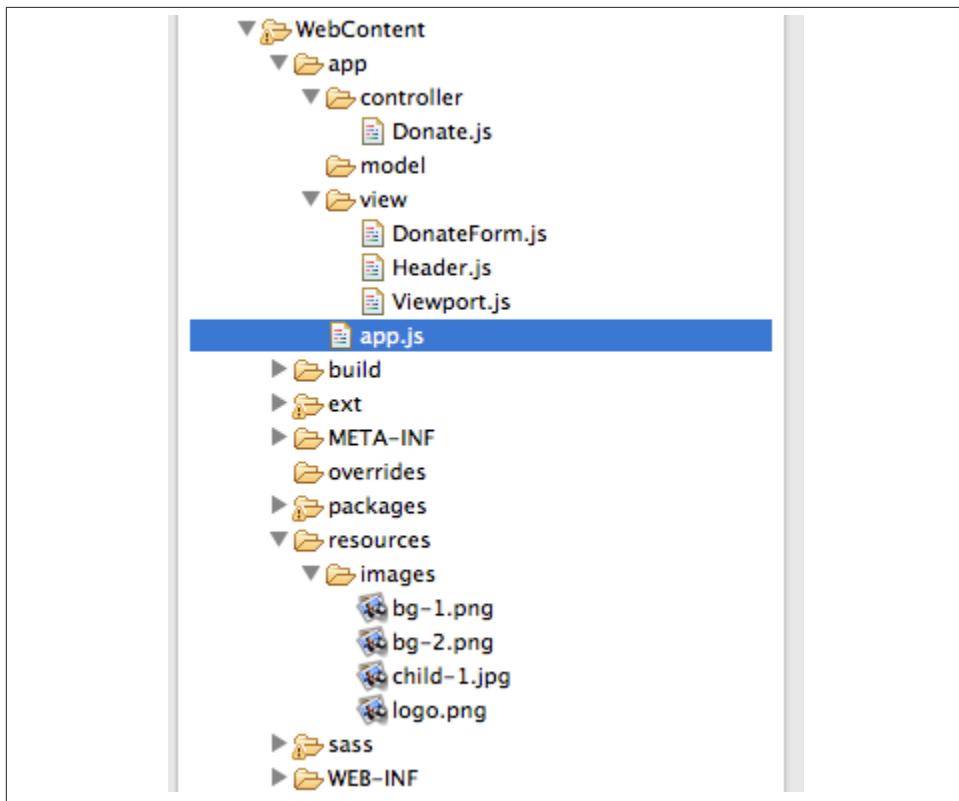


Figure 4-11. Controller, views and images of SSC\_Top\_ExtJS

The app.js is pretty short - it just declares SSC as the application name, views and controllers. By adding the property autoCreateViewport: true we requested the application to automatically load the main window, which must be called Viewport.js and located in the view directory.

```
Ext.application({
    name: 'SSC',

    views: [
        'DonateForm',
        'Header',
        'Viewport'
    ],

    controllers: [
        'Donate'
    ],
}
```

```
    autoCreateViewport: true
});
```

In this version of the application controller `Donate.js` is listening to the events from the view `DonateForm`. It's responsible just for the showing and hiding the `Donate` form panel. We've implemented the same behavior as in the previous version of the SaveThe Child application - click on the `Donate Now` button reveals the donation form. If the application would need to make some AJAX calls to the server, such code would also be placed in the controller. The code of the `Donate` controller looks as follows:

```
Ext.define('SSC.controller.Donate', {
    extend: 'Ext.app.Controller',

    refs: [{
        ref: 'donatePanel',
        selector: '[cls=donate-panel'
    }],

    init: function () { // ①
        this.control({
            'button[action=showform]': { // ②
                click: this.showDonateForm
            },
            'button[action=hideform]': {
                click: this.hideDonateForm
            },
            'button[action=donate]': {
                click: this.submitDonateForm
            }
        });
    },

    showDonateForm: function () { // ③
        this.getDonatePanel().getLayout().setActiveItem(1); // ④
    },

    hideDonateForm: function () {
        this.getDonatePanel().getLayout().setActiveItem(0);
    },

    submitDonateForm: function () {
        var form = this.getDonatePanel().down('form'); // ⑤
        form.isValid();
    }
});
```

- ① The `init()` method is invoked only once on instantiation of the controller.

- ❷ The `control()` method of the controller takes selectors as arguments to find components with the corresponding event listeners to be added. For example, '`button[action=showform]`' means "find a button that has a property `action` with the value `showform`" - it has the same meaning as in CSS selectors.
- ❸ Event handler functions to process show, hide, and submit events.
- ❹ In containers with card layout, you can make one of the components visible (the top one in the card deck) by passing its index to the method  `setActiveItem()`. The `Viewport.js` includes a container with the card layout (see '`cls: 'donate-panel'` in the next code sample).
- ❺ Finding the children of the container can be done using the method `down()` method. in this case we are finding the child `<form>` element of a donate panel. If you need to find the parents of the component use `up()`.



Since MVC paradigm splits the code into separate layers, you can unit test them separately, e.g. test your controllers separately from the Views. Chapter 7 is dedicated to JavaScript testing, and it contains sections "Testing The Models" and "Testing The Controllers" that illustrate how to arrange for separate testing of the models and controllers in the Ext JS version of the Save The Child application.

The top level window is a `SSC.view.Viewport`, which will contain the Header and the Donate form views.

```
Ext.define('SSC.view.Viewport', {
    extend: 'Ext.container.Viewport',
    requires: [
        'Ext.tab.Panel',
        'Ext.layout.container.Column'
    ],
    cls: 'app-viewport',
    layout: 'column',           // ❶
    defaults: {
        xtype: 'container'
    },
    items: [
        {
            columnWidth: 0.5,
            html: '&nbsp;' // Otherwise column collapses
        },
        {
            width: 980,
            cls: 'main-content',
            layout: {
                type: 'vbox',           // ❷
                align: 'stretch'
            }
        }
    ]
});
```

```

        },
        items: [
            {
                xtype: 'appheader'
            },
            {
                xtype: 'container',
                minHeight: 350,
                flex: 1,
            },
            cls: 'donate-panel',           // ③
            layout: 'card',
            items: [
                {
                    xtype: 'container',
                    layout: 'vbox',
                    items: [
                        {
                            xtype: 'component',
                            html: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent ...',
                            maxWidth: 550,
                            padding: '80 20 0'
                        },
                        {
                            xtype: 'button',
                            action: 'showform',
                            text: 'DONATE NOW',
                            scale: 'large',
                            margin: '30 230'
                        }
                    ],
                    {
                        xtype: 'donateform',
                        margin: '80 0 0 0'
                    }
                ],
                {
                    xtype: 'container',
                    flex: 1
                }
            ],
            {
                columnWidth: 0.5,
                html: '&nbsp;'
            }
        ]
    });
}

```

- ① Our viewport has a `column` layout, and will be explained after the [Figure 4-12 Collapse Code of Viewport.js](#) below.
- ② The vertical box layout will display the components from the `items` array one under another : the `appheader` and the `container`, which is explained next.

- ③ The container with the class selector `donate-panel` includes two components, but since they are laid out as `card`, only one of them will be shown at a time: either the one with the “Lorem ipsum” text, or the `donateform`. Which one to show is mandated by the `Donate` controller by invoking the method `setActiveItem()` with the appropriate index.

The following figure shows a snapshot from WebStorm IDE, with collapsed code section just to see the big picture of what are the columns in the column layout - they are marked with arrows.

The screenshot shows the WebStorm IDE interface with the project structure on the left and the code editor on the right. The code editor displays the `Viewport.js` file. Several code blocks are collapsed, indicated by red arrows pointing to the collapse icons. The collapsed sections include the `viewport` configuration, the `main-content` container, the `appheader` component, the `donate-panel` card, and the `donateform` component. The code uses ExtJS's ExtJS API, including `Ext.define`, `Ext.container.Viewport`, `Ext.tab.Panel`, `Ext.layout.container.Column`, `Ext.container.Container`, `Ext.layout.container.Vbox`, and `Ext.form.FormPanel`.

```

Ext.define('SSC.view.Viewport', {
    extend: 'Ext.container.Viewport',
    requires: [
        'Ext.tab.Panel',
        'Ext.layout.container.Column'
    ],
    cls: 'app-viewport',
    layout: 'column',
    defaults: {
        xtype: 'container'
    },
    items: [
        {
            columnWidth: 0.5,
            html: 'Otherwise column collapses'
        },
        {
            width: 900,
            cls: 'main-content',
            layout: {
                type: 'vbox',
                align: 'stretch'
            },
            items: [
                {
                    xtype: 'appheader'
                },
                {
                    xtype: 'container',
                    minHeight: 350,
                    flex: 1,
                    cls: 'donate-panel',
                    layout: 'card',
                    items: [
                        {
                            xtype: 'container',
                            layout: 'vbox',
                            items: [...]
                        },
                        {
                            xtype: 'donateform',
                            margin: '80 0 0 0'
                        }
                    ],
                    ["xtype": "container"]
                },
                {
                    columnWidth: 0.5,
                    html: 'Otherwise column collapses'
                }
            ]
        }
    ]
});

```

Figure 4-12. Collapsed Code of `Viewport.js`



Open its menu Preferences | JavaScript | Libraries and add the file ext-all-debug-w-comments.js as a global library and pressing the button F1 will display available comments about selected Ext JS element. Configuring Ext JS as external library allows you to remove Ext JS files from WebStorm project without losing context sensitive help.

In Ext JS the column layout is used when you are planning to present the information in columns as explained in the [product documentation](#). Even though there are three columns in this layout, the entire content on this page is located in the middle column having the width of 980. The column on the left and the column on the right just hold one non-breakable space each to provide centering of the middle column in monitors with high resolution wider than 980 pixels (plus the browser's chrome).

The width of *0.5, 980, 0.5* means to give the middle column 980 pixels and share the remaining space equally between empty columns.

Note: There is another way to lay out this screen using Horizontal Box hbox with the [pack configuration property](#), but we decided to keep the column layout for illustration purposes.



Consider using [Ext Designer](#) for creating layouts in the wysiwyg mode. .SASS and CSS

Take a look at the project structure shown at [Figure 4-12](#) - it has sass directory, which contains several files with styles: DonateForm.scss, Header.scss, and Viewport.scss. Note that the file name extension is not *css*, but *scss* - it's Syntactically Awesome Stylesheets (SASS). The content of the Viewport.css is shown below. In particular, if you've been wondering where are located the image of the boy and the background flowers - there are right there.

```
.app-viewport {  
    background: white;  
}  
  
.main-content {  
    background: url("images/bg-1.png") no-repeat;  
}  
  
.donate-panel {  
    background: url("images/child-1.jpg") no-repeat right bottom,  
    url("images/bg-2.png") no-repeat 90px bottom;
```

```
border-bottom: 1px dotted #555;  
}
```

**SASS** is an extension of CSS3, which allows using variables, mixins, inline imports, inherit selectors and more with CSS-compatible syntax. The simplest example of SASS syntax is to define a variable that stores some color code, e.g. `$mypanel-color: #cf6cc2;`. Now if you need to change the color you just change the value of the variable in one place rather than trying to find all places in a regular CSS where this color was used. But since modern Web browsers don't understand SASS styles, they have to be converted into regular CSS before deploying your Web applications.

Ext JS includes **Compass**, which is an open-source CSS Authoring Framework built on top of SASS. It includes a number of modules and functions that will save your time for defining such things as border radius, gradients, transitions and more in a cross-browser fashion. For example, you write one SASS line `.simple { @include border-radius(4px, 4px); }`, but Compass will generate the following cross-browser CSS section:

```
-webkit-border-radius: 4px 4px;  
-moz-border-radius: 4px / 4px;  
-khtml-border-radius: 4px / 4px;  
border-radius: 4px / 4px; }
```

See [Compass documentation](#) for more examples like the above. To manually compile your SASS into CSS you can use the command `compass compile` from the Command or Terminal window. This step is also performed automatically during the Sencha CMD application build. In case of the Save The Child application the resulting CSS file is located in `build/SSC/production/resources/SSC-all.css`.

We are not using any extended CSS syntax in our Save The Child application, but since SASS is a superset of CSS, you can use your existing CSS as is - just save it in the `.scss` file. If you'd like to learn more about the SASS syntax, visit the site [sass-lang.com](#), which has tutorials and reference documentation.

In general, Ext JS substantially reduces the need for manual CSS writing by using pre-defined **themes**. Sencha offers a [tutorial](#) explaining how to use SASS and Compass for theming.

Besides SASS, there is another dynamic CSS language called **LESS**. It adds to CSS variables, mixins, operations and functions. It's not used in Ext JS though.

Now let's look at the child elements of the `SSC.view.Viewport`. The `SSC.view.Header` is the simplest view. Since Save The Child does not include a bunch of forms and grids, we'll use the lightest top-level container class `Container` where possible. The class `Container` gives you the most freedom in what to put inside and how to layout its child elements. Our `SSC.view.Header` view extends `Ext.Container` and contains child elements, some of which have the `xtype: component`, and some - `container`:

```

Ext.define("SSC.view.Header", {
    extend: 'Ext.Container',
    xtype: 'appheader',           // ①
    cls: 'app-header',            // ②
    height: 85,
    layout: {                     // ③
        type: 'hbox',
        align: 'middle'
    },
    items: [{                     // ④
        xtype: 'component',
        cls: 'app-header-logo',
        width: 75,
        height: 75
    }, {
        xtype: 'component',
        cls: 'app-header-title',
        html: 'SAVE The Child',
        flex: 1
    }, {
        xtype: 'container',       // ⑤
        defaults: {
            scale: 'medium',
            margin: '0 0 0 5'
        },
        items: [{
            xtype: 'button',
            text: 'Who We Are'
        }, {
            xtype: 'button',
            text: 'What We Do'
        }, {
            xtype: 'button',
            text: 'Where We Work'
        }, {
            xtype: 'button',
            text: 'Way To Give'
        }]
    }]
});

```

- ① We assigned appheader as the xtype value of this view, which will be used as a reference inside the SSC.view.Viewport.
- ② cls is a class attribute of a DOM element. In this case it is the same as writing class=app-header in the HTML element.
- ③ The header uses hbox layout with center alignment

- ④ Child components of the top container are the logo image, the text “Save The Child”, and another container with buttons
- ⑤ A container with button components

Let's review the `DonateForm` view next, which is a subclass of `Ext.form.Panel` and contains the form with radio buttons, fields and labels. This component named `donate form` will be placed under `SSC.view.Header` inside `SSC.view.Viewport`. We've removed some of the lines code to make it more readable, but its full version is included in the source code samples accompanying the book. Here's the first part of the `SSC.view.DonateForm`.

```
Ext.define('SSC.view.DonateForm', {
    extend: 'Ext.form.Panel',
    xtype: 'donateform',
    requires: [ // ①
        'Ext.form.RadioGroup',
        'Ext.form.field.*',
        'Ext.form.Label'
    ],
    layout: { // ②
        type: 'hbox'
    },
    items:[{
        xtype: 'container', // ③
        layout: 'vbox',
        items: [{{
            xtype: 'container',
            items: [{{
                xtype: 'radiogroup',
                fieldLabel: 'Please select or enter donation amount',
                labelCls: 'donate-form-label',
                vertical: true,
                columns: 1,
                defaults: {
                    name: 'amount'
                },
                items: [
                    { boxLabel: '10', inputValue: '10' },
                    { boxLabel: '20', inputValue: '20' }
                    // more choices 50, 100, 200 go here
                ]
            }}]
        }, {
    }]
```

```

        xtype: 'textfield',
        fieldLabel: 'Other amount',
        labelCls: 'donate-form-label'
    }]
},

```

- ❶ `DonateForm` depends on several classes listed in the `requires` property. The Ext JS will check to see if these classes are present in memory, and if not, the loader will load all dependencies first, and only after the `DonateForm` class.
- ❷ Our `DonateForm` uses horizontal box (`hbox`) layout, which means that certain components or containers will be laid out next to each other horizontally. But which ones? The children of the container located in the `items[]` arrays hence they are the ones that will be laid out horizontally in this case. But the above code contains several of `items[]` arrays with different levels of nesting. How quickly find those that belong to the topmost container `DonateForm`? This is the case that clearly shows that having a good IDE can be of great help.

[Figure 4-13](#) shows a snapshot from the WebStorm IDE illustrating how you can find the matching elements in the long listings. The top level `items[]` arrays starts from line 23 and we see that the first element to be laid out by in `hbox` has the `xtype: container`, which in turn has some children. If you'll move the blinking cursor of the WebStorm editor right after the first open curly brace in line 23, you'll see a thin blue vertical line that goes down to line 60. This is where the first object literal ends.

Hence the second object to be governed by the `hbox` layout starts on line 61. You can repeat the same trick with the cursor to see where that object ends and the `fieldcontainer` starts. This might seem like a not overly important tip, but it really saves developer's time.

- ❸ The first element of the `hbox` is a container that internally laid out as a `vbox` (see [Figure 4-14](#)). The `radiogroup` is on top and the `textfield` for entering Other amount at the bottom.

The second part of the `SSC.view.DonateForm` comes next.

```

{
    xtype: 'fieldcontainer', // ❶
    fieldLabel: 'Donor information',
    labelCls: 'donate-form-label',

    items: [{
        xtype: 'textfield',
        name: 'donor',
        emptyText: 'full name'
    }, {

```

```

        xtype: 'textfield',
        emptyText: 'email'
    }
    // address,city,zip code,state and country go here
]
], {
    xtype: 'container',           // ②
    layout: {
        type: 'vbox',
        align: 'center'
    },
    items: [
        {
            xtype: 'label',
            text: 'We accept PayPal payments',
            cls: 'donate-form-label'
        },
        {
            xtype: 'component',
            html: 'Your payment will processed securely by PayPal...'
        },
        {
            xtype: 'button',
            action: 'donate',
            text: 'DONATE NOW'
        },
        {
            xtype: 'button',
            action: 'hideform',
            text: 'I will donate later'
        }
    ]
}
]);
});

```

- ① The **fieldcontainer** is a light-weight Ext JS container useful to group components - the donor information in this case. It's the central element in the hbox container shown in [Figure 4-14](#).
- ② The right side of the hbox is another container with the vbox internal layout to show the “We accept Paypal” message, “DONATE NOW”, and “I’ll donate later” buttons (see [Figure 4-14](#)). These buttons respond to clicks because

WebContent (~/Documents/Farata/Enterprise)

```

9   layout: {
10    type: ' hbox',
11    align: 'stretch'
12  },
13
14  bodyStyle: {
15    backgroundColor: ' transparent'
16  },
17
18  defaults: {
19    margin: ' 0 50 0 0'
20  },
21
22
23  items:[{
24    xtype: ' container',
25    layout: ' vbox',
26
27    items: [{
28      xtype: ' container',
29      width: 200,
30
31      items: [{
32        xtype: ' radiogroup',
33        fieldLabel: 'Please select or enter donation amount',
34        labelAlign: ' top',
35        labelSeparator: '',
36        labelCls: ' donate-form-label',
37
38        vertical: true,
39        columns: 1,
40
41        defaults: {
42          name: ' amount'
43        },
44
45        items: [
46          { boxLabel: ' 10 ', inputValue: ' 10 ' },
47          { boxLabel: ' 20 ', inputValue: ' 20 ' },
48          { boxLabel: ' 50 ', inputValue: ' 50 ' },
49          { boxLabel: ' 100 ', inputValue: ' 100 ' },
50          { boxLabel: ' 200 ', inputValue: ' 200 ' }
51        ]
52      }]
53    },
54    xtype: ' textfield',
55    fieldLabel: ' Other amount',
56    labelAlign: ' top',
57    labelSeparator: '',
58    labelCls: ' donate-form-label'
59  }],
60  xtype: ' fieldcontainer',
61  fieldLabel: ' Donor information',
62  labelAlign: ' top',
63  labelSeparator: ''
64

```

Figure 4-13. Collapsed Code of Viewport.js



Debugging of frameworks that are extensions of JavaScript in Web browsers can be difficult, because while you may be operating with, say Ext JS classes, the browser will receive regular `<div>`, `<p>` and other HTML tags and JavaScript. [Illuminations](#) is a Firebug add-on that allows to inspect elements showing not just their HTML representations, but the corresponding Ext JS classes that were used to create them.

Figure 4-14. *DonateForm.js*: an hbox with three vbox containers

The code review of the top portion of the Save The Child application is finished. Run the SSC\_Top\_ExtJS project and turn on the Chrome Developers Tools. Scroll to the bottom of the Network tab, and you'll see that the browser made about 250 requests to the server and downloaded 4.5Mb in total. Not too exciting isn't it?

On the next runs these numbers will drop to about 30 requests and 1.7Mb transferred - the browser's caching kicked in. These numbers would be better if instead of ext-all.js we'd be linking ext.js, and even better if we'd created a custom build (see Sencha CMB section above) for the Save The Child application merging the application code into one file to contain only those framework classes that were actually used.

## Completing Save The Child

In this section we'll review the code supporting the lower half of the Save The Child UI, which you should import into Eclipse IDE from the directory SSC\_Complete\_ExtJS.

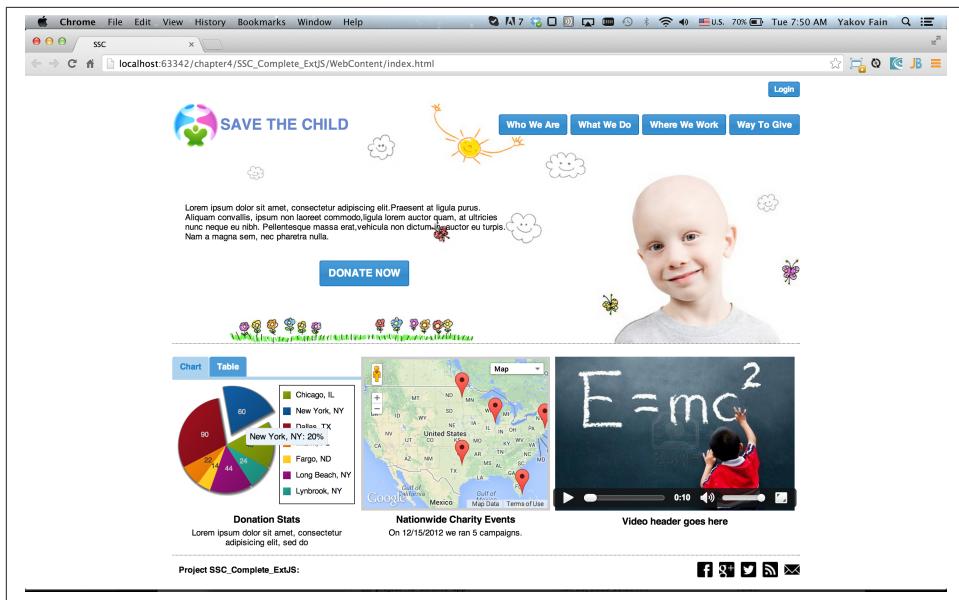


Figure 4-15. Save The Child with live charts

If you see the target runtime error, read the beginning of the section “Save The Child: The Top Portion” for the cure. Stop the Tomcat server if running, and deploy the `SSC_Complete_ExtJS` under Tomcat server in the Servers view (the right-click menu, Add and Remove...). Start Tomcat in Eclipse, right-click on the project and run it on the server. It’ll open up a Web browser pointing at [http://localhost:8080/SSC\\_Complete\\_ExtJS](http://localhost:8080/SSC_Complete_ExtJS) showing the window similar to the one depicted on Figure 4-15.

This version has some additions comparing to the previous ones. Notice the bottom left panel with charts. First of all, the charts are placed inside the panel with tabs: Charts and Table. The same data can be rendered either as a chart or as a grid. Second, the charts became live thanks to ExtJS. We took a snapshot of the Window shown in Figure 4-15 while hovering the mouse over the pie slice representing New York, and the slice has extended from the pie showing a tooltip.

The `SSC_Complete_ExtJS` has more Ext JS classes comparing to `SSC_Top_ExtJS`. You can see more views on Figure 4-16. Besides, we’ve added two classes `Donors.js` and `Campaigns.js` to serve as data stores for the panels with charts and maps.

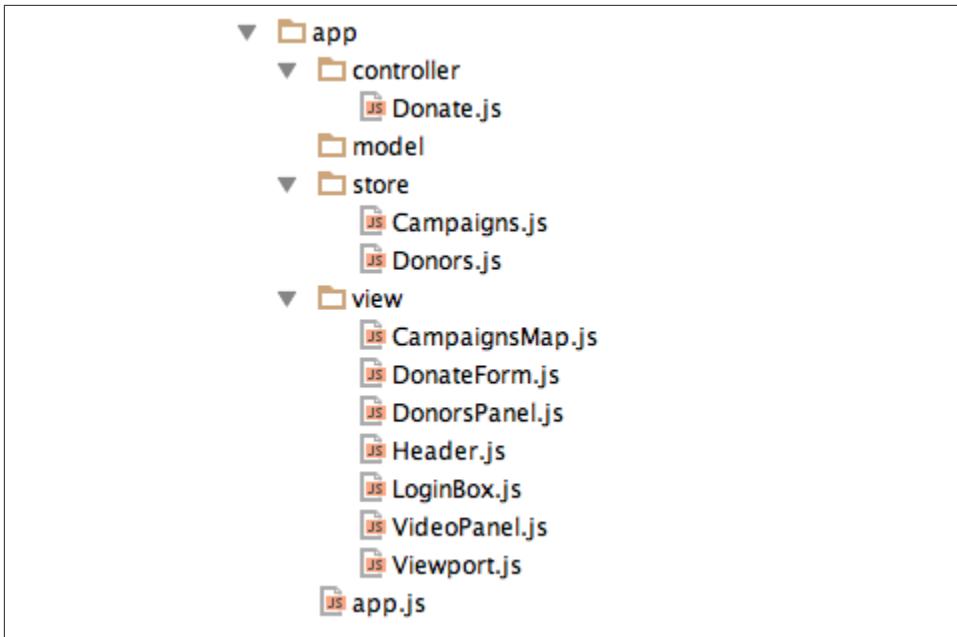


Figure 4-16. JavaScript classes of SSC\_Complete\_ExtJS

## Adding the Login Box

The Login Box view is pretty small and self explanatory:

```
Ext.define("SSC.view.LoginBox", {
    extend: 'Ext.Container',
    xtype: 'loginbox',

    layout: 'hbox',

    items: [
        xtype: 'container',
        flex: 1
    ], [
        xtype: 'textfield',
        emptyText: 'username',
        name: 'username',
        hidden: true
    ], [
        xtype: 'textfield',
        emptyText: 'password',
        inputType: 'password',
        name: 'password',
        hidden: true
    ], [
        xtype: 'button',
```

```

        text: 'Login',
        action: 'login'
    }]
});

'button[action=login]': {
    click: this.showLoginFields
}
...

showLoginFields: function () {
    this.getUsernameBox().show();
    this.getPasswordBox().show();
}

```

## Adding the Video

The bottom portion of the Windows includes several components. The video view simply reuses the HTML <video> tag we used in chapters 4 and 5. Ext JS 4.2 doesn't offer any other solutions for embedding videos. On one hand, sub-classing Ext.Component is the lightest way of including any arbitrary HTML markup. On the other hand, turning HTML into an Ext JS component allows us to use it the same way as any other Ext JS component, e.g. participate in layouts. Here's the code of the VideoPanel.js:

```

Ext.define("SSC.view.VideoPanel", {
    extend: 'Ext.Component',
    xtype: 'videopanel',

    html: [
        '<video controls="controls" poster="resources/media/intro.jpg" width="390px" height="240px" pre',
        '    '<source src="resources/media/intro.mp4" type="video/mp4"/>',
        '    '<source src="resources/media/intro.webm" type="video/webm"/>',
        '    '<p>Sorry, your browser doesn\'t support the video element</p>',
        '</video>'
    ]
});

});

```



Ext JS has a wrapper for the HTML5 <video> tag. It's called `Ext.Video`, and we'll use it in Chapter on Sencha Touch.

## Adding the Maps

Adding the map takes considerably more work on our part. The mapping part is located in the view CampaignsMap.js. Initially we tried to use the `Ext.ux.GMapPanel`, but it

didn't work as expected. As a workaround, we've added the HTML `<div>` element to serve as a map container. The first part of the content of `CampaignsMap.js` is shown below.

```
Ext.define("SSC.view.CampaignsMap", {
    extend: 'Ext.Component',
    xtype: 'campaignsmap',

    html: ['<div class="gmap"></div>'],

    renderSelectors: { // ①
        mapContainer: 'div'
    },

    listeners: { // ②
        afterrender: function (comp) {
            var map,
                mapDiv = comp.mapContainer.dom; // ③

            if (navigator && navigator.onLine) { // ④
                try {
                    map = comp.initMap(mapDiv);
                    comp.addCampaignsOnTheMap(map);
                } catch (e) {
                    this.displayGoogleMapError();
                }
            } else {
                this.displayGoogleMapError();
            }
        }
    },
});
```

- ① Since we've added the map container just by including the HTML `<div>` component, Ext JS will create generated ID for this `<div>`. It's just not a good way to reference an element on the page, since the ID should be unique and we can easily run into conflicting situation. We didn't want to create an ID manually hence used the property `renderSelectors` allows to map an arbitrary name to a DOM selector. When we reference this element somewhere inside Ext JS code using this `renderSelector`, e.g. `this.mapContainer` (`mapContainer` is an arbitrary name here), it returns `Ext.dom.Element` object - an abstraction over the plain HTML element - that eliminates cross-browser API differences.

- ② Sencha documentation states that declaring `listeners` during `Ext.define()` is bad practice and doing it during `Ext.create()` should be preferred. This is an arguable statement. Yes, there is a possibility that the handler function will be created during `define()` but never used during `create()`, which will lead to unnecessary creation of the handler's instance in memory. But the chances are slim. The other consideration is that if listeners are defined during `create()` each instance can handle the same event differently. We'll leave it up to you to decide where's the right place for defining listeners. The good part about keeping listeners in the class definition is that the entire code of the class is located in one place.
- ③ Querying the DOM to find the `mapContainer` defined in the `renderSelectors` property. Note that we are getting the reference to this DOM element after the view is rendered in the event handler function `afterRender`. The object `comp` will be provided to this handler, and it points at the instance of the current component, which is `SSC.view.CampaignsMap`. Think of `comp` as `this` for the component.
- ④ If Google Map is not available, display an error message as in [Figure 4-17](#). This code was added after one of the authors was testing this code while sitting in the plane with no Internet connection. But checking the status of `navigator.onLine` may not be a reliable indicator of the offline status, so we've wrapped it into a `try/catch` block just to be sure.

Next comes the second part of the `CampaignsMap.js`.

```

initMap: function (mapDiv) { // ❶
    // latitude = 39.8097343 longitude = -98.555619900000001
    // Lebanon, KS 66952, USA Geographic center of the contiguous United States
    // the center point of the map
    var latMapCenter = 39.8097343,
        lonMapCenter = -98.555619900000001;

    var mapOptions = {
        zoom      : 3,
        center   : new google.maps.LatLng(latMapCenter, lonMapCenter),
        mapTypeId: google.maps.MapTypeId.ROADMAP,
        mapTypeControlOptions: {
            style  : google.maps.MapTypeControlStyle.DROPDOWN_MENU,
            position: google.maps.ControlPosition.TOP_RIGHT
        }
    };

    return new google.maps.Map(mapDiv, mapOptions);
},
addCampaignsOnTheMap: function (map) {
    var marker,

```

```

infowindow = new google.maps.InfoWindow(),
geocoder   = new google.maps.Geocoder(),
campaigns  = Ext.StoreMgr.get('Campaigns'); // ②

campaigns.each(function (campaign) {
    var title      = campaign.get('title'),           // ③
        location    = campaign.get('location'),
        description = campaign.get('description');

    geocoder.geocode({
        address: location,
        country: 'USA'
    }, function(results, status) {
        if (status == google.maps.GeocoderStatus.OK) {

            // getting coordinates
            var lat = results[0].geometry.location.lat(),
                lon = results[0].geometry.location.lng();

            // create marker
            marker = new google.maps.Marker({
                position: new google.maps.LatLng(lat, lon),
                map     : map,
                title   : location
            });

            // adding click event to the marker to show info-bubble with data from json
            google.maps.event.addListener(marker, 'click', (function(marker) {
                return function () {
                    var content = Ext.String.format(
                        '<p class="infowindow"><b>{0}</b><br/>[1]<br/><i>{2}</i></p>',
                        title, description, location);

                    infowindow.setContent(content);
                    infowindow.open(map, marker);
                };
            })(marker));
        } else {
            console.error('Error getting location data for address: ' + location);
        }
    });
},

displayGoogleMapError: function () {
    console.log('Error is successfully handled while rendering Google map');
    this.mapContainer.update('<p class="error">Sorry, Google Map service isn\'t available</p>');
}
);

```

- ① The rest of the code in this class has the same mapping functionality as described in Chapter 1 in section “Adding Geolocation Support”.

- ② The data for the campaign information are coming from the store Campaigns.js located in the folder store. The store manager can find the reference to the store either if by assigned `storeId` or by name `Campaigns` listed in the `stores` array in the app.js:
- ③ We are configuring the mapping panel to get the information about the campaign title, location, and description from the fields with corresponding names from the store `SSC.store.Campaigns` that's shown right after app.js below.

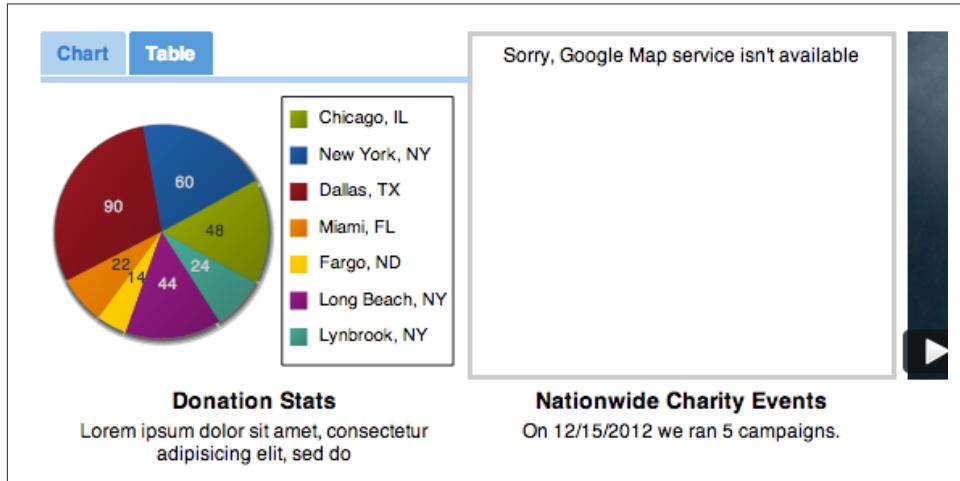


Figure 4-17. If Google Maps server is not responding

```
Ext.application({
    name: 'SSC',
    views: [
        'CampaignsMap',
        'DonateForm',
        'DonorsPanel',
        'Header',
        'LoginBox',
        'VideoPanel',
        'Viewport'
    ],
    stores: [
        'Campaigns',
        'Donors'
    ],
    controllers: [
```

```

        'Donate'
    ],
    autoCreateViewport: true
});

```

In Chapter 2 the information about campaigns was taken from a file with JSON formatted data. In this version the data will be taken from the class `SSC.store.Campaigns` that's shown next. This class extends `Ext.data.JsonStore`, which is a helper class for creating stores based on the JSON data. The class `JsonStore` is a subclass of more generic `Ext.data.Store`, which implements client side caching of Model objects, can load the data via the `Proxy` object, and supports sorting and filtering.

Later, in the Sencha Touch Chapter, you'll see another version of our Save The Child application where all stores are inherited from `Ext.data.Store`. But in this version of our application we are not reading the code from external JSON sources and inheriting from `Ext.data.Store` would suffice.

```

Ext.define('SSC.store.Campaigns', {
    extend: 'Ext.data.JsonStore',
    fields: [
        { name: 'title', type: 'string' }, // ❶
        { name: 'description', type: 'string' },
        { name: 'location', type: 'string' }
    ],
    data: [{} // ❷
        title: 'Lorem ipsum',
        description: 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.',
        location: 'Chicago, IL'
    ], {
        title: 'Donors meeting',
        description: 'Morbi mollis ante at ante posuere tempor.',
        location: 'New York, NY'
    }, {
        title: 'Sed tincidunt magna',
        description: 'Donec ac ligula sit amet libero vehicula laoreet',
        location: 'Dallas, TX'
    }, {
        title: 'Fusce tellus dui',
        description: 'Sed accumsan nibh sapien, interdum ullamcorper velit.',
        location: 'Miami, FL'
    }, {
        title: 'Aenean lorem quam',
        description: 'Pellentesque habitant morbi tristique senectus',
        location: 'Fargo, ND'
    }]
});

```

- ❶ We have not created a separate model class for each campaign, because this information is used only in one place. The `fields` array defines our inline model, which consists of objects (`data`) containing the properties `title`, `description`, and `location`.
- ❷ Hard-coded data for the model

## Adding the Chart and Table Panels

The bottom left area of the Save The Child window is occupied by a subclass of `Ext.tab.Panel`. The name of our view is `SSC.view.DonorsPanel`, and it contains two tabs: Chart and Table. Accordingly, the class definition starts with declaring dependencies for the Ext JS classes that supports charts and a data grid.

Charting is an important part of many enterprise applications, and Ext JS framework offers solid chart drawing capabilities without the need to install any plugins. We'd like to stress that both Chart and Table panels use the same data - they just provide different views of the data. Let's review the code now.

```
Ext.define("SSC.view.DonorsPanel", {
    extend: 'Ext.tab.Panel',
    xtype: 'donorspanel',
    requires: [
        'Ext.chart.Chart',
        'Ext.chart.series.Pie',
        'Ext.grid.Panel',
        'Ext.grid.column.Number',
        'Ext.grid.plugin.CellEditing'
    ],
    maxHeight: 240,
    plain: true, // ❶
    items: [{ // ❷
        title: 'Chart',
        xtype: 'chart',
        store: 'Donors',
        animate: true,
        legend: {
            position: 'right'
        },
        theme: 'Base:gradients',
        series: [{ // ❸
            type: 'pie',
            angleField: 'donors',
            showInLegend: true,
            tips: { // ❹
                trackMouse: true,
                renderer: function (storeItem) {

```

```

var store = storeItem.store,
    total = 0;

store.each(function(rec) {
    total += rec.get('donors');           // ⑤
});

this.update(Ext.String.format('{0}: {1}%',          // ⑥
    storeItem.get('location'),
    Math.round(storeItem.get('donors') / total * 100)));
}

highlight: {
    segment: {
        margin: 20
    }
},
label: {                                         // ⑦
    field: 'location',
    display: 'horizontal',
    contrast: true,
    renderer: function (label, item, storeItem) {
        return storeItem.get('donors');
    }
}
}]
}, {
    title: 'Table',                         // ⑧
    xtype: 'gridpanel',
    store: 'Donors',
    columns: [                                // ⑨
        { text: 'State', dataIndex: 'location', flex: 1},
        { text: 'Donors', dataIndex: 'donors',
            xtype: 'numbercolumn', format: '0', editor: 'numberfield' }
    ],
    plugins: [{                               // ⑩
        ptype: 'cellediting'
    }]
}
]);

```

- ① By default, the top portion of the tab panel was showing a blue background, which we didn't like and turned this style off to give a little cleaner look to the tabs.
- ② The first panel is an instance of xtype chart, which gets the data from the store object Donors.
- ③ Configuring and creating a **Pie Chart**. The width of each sector is controlled by the angleField property, which is mapped to the field donors defined in the store SSC.store.Donors (see the code listing below).

- ④ We've overridden the config `renderer` to provide custom styling for each element. In particular, we've configured `tips` to be displayed on mouse hover.
- ⑤ Calculating total for proper display of the percentages on mouse hover.
- ⑥ The label for each pie sector is retrieved from the field `location` defined in the store `SSC.store.Donors` shown in the code listing below.
- ⑦ Displaying the chart legend on the right side. If the user moves the mouse over the legend, the pie sectors start to animate.
- ⑧ The second tab contains an instance of xtype `gridpanel`. Note that the store object is the same as the Chart panel uses.
- ⑨ The grid has two columns. One of them is a simple text, but the other is rendered as a `numbercolumn` that displays the data according to a format string.

The store `Donors` contains the hard-coded data for our pie chart as well as for the table. In the real world, the data would be retrieved from the server side. Since we were getting ready to consume JSON data (not implemented), our `Donors` class.

```
Ext.define('SSC.store.Donors', {
    extend: 'Ext.data.JsonStore',
    fields: [
        { name: 'donors', type: 'int' }, // ①
        { name: 'location', type: 'string' }
    ],
    data: [ // ②
        { donors: 48, location: 'Chicago, IL' },
        { donors: 60, location: 'New York, NY' },
        { donors: 90, location: 'Dallas, TX' },
        { donors: 22, location: 'Miami, FL' },
        { donors: 14, location: 'Fargo, ND' },
        { donors: 44, location: 'Long Beach, NY' },
        { donors: 24, location: 'Lynbrook, NY' }
    ]
});
```

- ① Defining inline model
- ② Hard-coded data for the model

The data located in the store `SSC.store.Donors` can be rendered not only as a chart, but in a tabular form as well. To switch to a table view shown in [Figure 4-18](#) the user has to click on the tab Table.

State	Donors
Chicago, IL	48
New York, NY	60
Dallas, TX	90
Miami, FL	22
Fargo, ND	14
Long Beach, NY	44
Lynbrook, NY	24

**Donation Stats**

Lorem ipsum dolor sit amet, consectetur  
adipisicing elit, sed do

Figure 4-18. The Table tab

The following code fragment from ‘DonorsPanel’ is all it takes to render the donors’ data as a grid. The xtype of this component is gridpanel. For illustration purposes we made the column Donors editable - double click on the a cell with the number and it’ll turn this field into a numeric field as shown in [Figure 4-18](#) for the location Fargo, ND.

```
{
    title: 'Table',
    xtype: 'gridpanel',
    store: 'Donors', // ①
    columns: [
        { text: 'City/State', dataIndex: 'location', flex: 1},
        { text: 'Donors', dataIndex: 'donors', xtype: 'numbercolumn', format: '0', editor: 'numberfield'},
    ],
    plugins: [{ptype: 'cellediting'} // ②
  ]}
```

- ① Reusing the same store as in chart panel

- ② We are using one of the exiting Ext JS plugins here, namely `Ext.grid.plugin.CellEditing` to allow editing the cells of the `Donors` column. In this example we are using an existing Ext JS editor `numberfield` in the `Donors` column. Since we don't work with decimal numbers here, the editor uses `format: 0`. To make the entire row of the grid editable use the plugin `Ext.grid.plugin.RowEditing`. If you wanted to create some custom plugin for a cell, you'd need to define it by the rules for writing Ext JS plugins.



Modify the any value in the Donor's cell and switch to the Chart panel. You'll see that the size corresponding pie sector has changed accordingly.

The total number of lines of code in `DonorsPanel` and in the store `Donors` is under 100. Being able to create a tab panel with chart and grid with almost no manual coding is quite impressive, isn't it?

### Adding a Footer

To complete Save The Child code review, we need to mention the icons located in the bottom of the `ViewPort.js` shown on [Figure 4-19](#). Usually links at the bottom of the page statically refer to the corresponding social network's account. Integration with social networks is out of this book's scope. But you can study, say Twitter API and implement the functionality to let donors tweet about their donations. The Facebook icon can either have a similar functionality or you may consider implementing automated login to the Save The Child application using OAuth2, which was briefly discussed in the chapter Introduction to Web Application Security.



Figure 4-19. The Viewport footer

This footer was implemented in the code snippet below. We've implemented these little icons as regular images.

```
items: [{  
    xtype: 'component',  
    flex: 1,  
    html: '<strong>Project SSC_Complete_ExtJS:</strong>'  
}, {  
    src: 'resources/images/facebook.png'  
}, {
```

```

        src: 'resources/images/google_plus.png'
    }, {
        src: 'resources/images/twitter.png'
    }, {
        src: 'resources/images/rss.png'
    }, {
        src: 'resources/images/email.png'
    }]

```

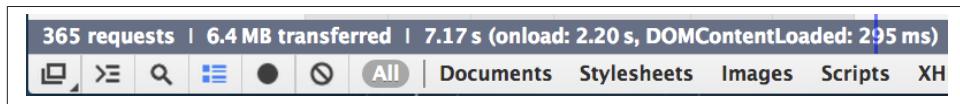


There is a more efficient way to do this by using a numeric character code that renders as image (see the [glyph config property](#)). The [Pictos library](#) offers more than three hundred of such tiny images in both vector or PNG form. you'll see the example of using Pictos fonts in Chapter 12.

Ext JS library contains lots of JavaScript code, but it allows developers produce nice looking applications with a fraction of a code comparing to other frameworks. Also, despite the fact that this version of Save The Child offers more functionality than those from the previous chapters, we've had to write the bare minimum of the CSS code thanks to Ext JS [theming](#).

### Building Production Version of Save The Child

Run the completed version of our application in Chrome browser having Developers Tools turned on. Go to the Network tab and scroll to the bottom. You'll see the message reporting that the browser made 365 requests to the server and downloaded 6.4Mb of content as in [Figure 4-20](#).



*Figure 4-20. The size of development version of Save The Child*

Now let's create production version with all JavaScript merged into one file. Open the Terminal or Command window and change directory into the Eclipse workspace directory where your project was created (e.g. `.../SSC_Complete_ExtJS/WebContent`) and enter the command described in the Sencha CMD section earlier in this chapter:

```
sencha app build
```

The production version of the Save The Child application will be generated in the directory `.../SSC_Complete_ExtJS/WebContent/build/SSC/production`. All your application JavaScript code will be merged with the required classes of Ext JS framework into one file `all-classes.js`, which in our case weighs 1.2Mb. The generated CSS file `SSC-all.css`

will be located in the directory *resources*. All images are there too. This is how the production version of index.html will look like:

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="UTF-8">
    <title>SSC</title>
    <script src="http://maps.googleapis.com/maps/api/js?sensor=false"></script>
    <link rel="stylesheet" href="resources/SSC-all.css"/>
    <script type="text/javascript" src="all-classes.js"></script>
</head>
<body></body>
</html>
```

Deploy the content of the *production* under any Web server and load this version of the application in the Chrome with Developer Tools turned on. This time the number of downloaded bytes is three times lower (2.3Mb). Ask your Web Server administrator to enable gzip or deflate, and the size of the JavaScript will go down from 1.2Mb to 365Kb. The size of other resources will decrease even more. Don't forget that we are loading a 500Kb video file intro.mp4. The number of server requests went down to 55, but more than 30 of them were Google Map API calls.



Figure 4-21. The size of production version of Save The Child

## Summary

Creation of enterprise web applications involves many steps that need to be done by developers. But with the right set of tools the repetitive steps can be automated. Besides, Ext JS class rich component library and themes allows you to lower the amount of manual programming.

Remember the DRY principle - don't repeat yourself. Try to do more with less efforts. This rather long chapter will help you to get started with Ext JS framework. It's an extensive framework, which doesn't allow an easy way out should you decide to switch to another one. But for the enterprise applications that require rich UI, dashboards with fancy charts, advanced data grids Ext JS can be a good choice.

# **Selected Productivity Tools for Enterprise Developers**

The toolbox of an enterprise HTML5 developer contains a number of tools that improve his or her productivity. In this chapter we'll share with you some of the tools that we use.

We'll start this chapter with a brief introduction of Node.js (or simply Node) - the server-side JavaScript framework and Node Packages Managers (NPM). Node and NPM serve as a foundation for the tools covered in this chapter.

Next, we'd like to highlight the a handful of productivity tools that authors of this book use in their consulting projects, namely:

- Grunt is a task runner framework for the JavaScript projects that allows to automate repetitive operations like running tests.
- Bower is a package manager for the web projects that helps in maintaining application dependencies.
- Yeoman is a collection of code-generation tools and best practices.

In addition to the above tools that can be used with various JavaScript frameworks, we'll introduce you to Clear Toolkit for Ext JS, which include the code generator Clear Data Builder - it's created and open sourced by our company, Farata Systems. With Clear Toolkit you'll be able to quickly start the project that utilizes Ext JS framework for the front-end development and Java on the server side.

## **Node.js, V8, and NPM**

Node.js is a server-side JavaScript framework. Node uses V8, the JavaScript engine by Google (Chrome/Chromium also use it). Node provides JavaScript API for accessing

the file system, sockets and running processes which makes it great general purpose scripting runtime. You can find more information about Node at [their website](#).

Many tools are built on top of Node JavaScript APIs. The [Grunt](#) tool is one of them. We will use Grunt later in this book to automate execution of repetitive development tasks.

NPM is a utility that comes bundled with Node. NPM provides unified API and metadata model for managing dependencies in JavaScript projects. A `package.json` file is the project's dependencies descriptor. NPM installs project dependencies using information from `package.json`. NPM uses [community repository](#) for open source JavaScript projects to resolve dependencies. NPM can also use private repositories.

Node and NPM are cross-platform software and binaries available for Windows, Linux and OS X operating systems.

To use this book code samples you need to download and install Node from [their web site](#).

## Automate Everything With Grunt

You should automate every aspect of the development workflow to reduce the cost of building, deploying, and maintaining your application.

In this section we are going to introduce [Grunt](#) - a task runner framework for the JavaScript projects - that can help you with automation of repetitive operations like running tests when the code changes. You can follow [the instructions from Grunt's website](#) to install it on your machine.

Grunt can watch your code changes and automate the process of running tests when the code changes. Tests should help in assessing the quality of our code.

With the Grunt tool you can have a script to run all your tests. If you came from the Java world, you know about Apache Ant, a general-purpose command-line tool to drive processes described *build files* as *targets* in the build.xml file. Grunt also runs the tasks described in scripts. There is a wide range of tasks available today - starting with running automated unit tests and ending with JavaScript code minification. Grunt provides a separate layer of abstraction where you can define tasks in a special DSL (domain-specific language) in Gruntfile for execution.

## The Simplest Grunt File

Let's start with the simplest Grunt project setup. The following two files must be present in the project directory:

- `package.json`: This file is used by NPM to store metadata and a project dependencies.

List Grunt and its plugins that your project needs as *devDependencies* in this file.

- **Gruntfile:** This file is named Gruntfile.js or Gruntfile.coffee and is used to configure or define the tasks and load Grunt plugins.

*Example 5-1. The simplest possible Gruntfile*

```
module.exports = function (grunt) {
  'use strict';

  grunt.registerTask('hello', 'say hello', function(){      // ①
    grunt.log.writeln('Hello from grunt');                  // ②
  });

  grunt.registerTask('default', 'hello');                    // ③
};
```

- ① Register a new task named `hello`.
- ② Print the greeting text using `grunt's log API`.
- ③ With `grunt.registerTask` we define a default task to run when Grunt is called without any parameters.

Each task can be called separately from the command line by passing the task's name as a command line parameter. For example, `grunt hello` would only execute the task named "hello" from the above script.

Let's run this `hello` task with the following command:

```
grunt --gruntfile Grunt_simple.js hello.
```

```
Running "hello" task
Hello from grunt
```

Done, without errors.

## Using Grunt to run JSHint Checks

Now after covering the basics of Grunt tool we can use it for something more interesting than just printing "*hello world*" on the screen. Since JavaScript is a interpreted language there is no compiler to help catch syntax errors. But you can use **JSHint**, an open source tool, which helps with identifying errors in JavaScript code in lieu of compiler. Consider the following JavaScript `code`.

*Example 5-2. A JavaScript array with a couple typos*

```
var bonds = [                                // ①
  'Sean Connery',
  'George Lazenby',
  'Roger Moore',
```

```
'Timothy Dalton',
'Pierce Brosnan',
'Daniel Craig',    // ②
//'Unknown yet actor'
]                  // ③
```

- ❶ We want to define an array that contains names of actors who played James Bond in a canonical series.
- ❷ Here is example of a typo that may cause errors in some browsers. A developer commented out the line containing an array element but kept the coma in previous line.
- ❸ A missing semicolon is a typical typo. Although it is not an error (and many JavaScript developers do consider omitting semicolons as “best practice”), an automatic semicolon insertion (ASI) will get you covered in this case.

## What is a Automatic Semicolon Insertion?

In JavaScript, the semicolons are optional meaning that you can omit a semicolon between two statements written in separate lines. Automatic semicolon insertion is a source code parsing procedure that infers omitted semicolons in certain contexts into your program. You can read more about optional semicolons in JavaScript in the chapter “Optional Semicolons” in *JavaScript. Definitive Guide. 6th Edition* book.

The above code snippet is a fairly simple example that can cause trouble and frustration if you don’t have proper tools to check the code semantics and syntax. Let’s see how JSHint can help in this situation.

JSHint can be installed via NPM with command `npm install jshint -g`. Now you can run JSHint against our code snippet:

```
> jshint jshint_example.js
jshint_example.js: line 7, col 27, Extra comma. (it breaks older versions of IE)
jshint_example.js: line 9, col 10, Missing semicolon. # ①
2 errors          # ②
```

- ❶ JSHint reports the location of error and a short description of the problem.
- ❷ The total count of errors



WebStorm IDE has **built-in support** for JSHint tool. There is 3rd party plugin for Eclipse - [jshint-eclipse](#).

Grunt also has a task to run JSHint against your JavaScript code base. Here is how JSHint configuration in Grunt looks like.

*Example 5-3. A grunt file with JSHint support*

```
module.exports = function(grunt) {
  grunt.initConfig({
    jshint: {
      gruntfile: { // ❶
        src: ['Gruntfile_jshint.js']
      },
      app: {
        src: ['app/js/app.js']
      }
    }
  });

  grunt.loadNpmTasks('grunt-contrib-jshint');
  grunt.registerTask('default', ['jshint']); // ❷
};
```

- ❶ Because Gruntfile is JavaScript file, JSHint can check it as well and identify the errors.
- ❷ The `grunt-contrib-jshint` has to be installed. When `grunt` will be run without any parameters, default task `jshint` will be triggered.

```
> grunt

Running "jshint:gruntfile" (jshint) task
>> 1 file lint free.

Running "jshint:app" (jshint) task
>> 1 file lint free.

Done, without errors.
```

## Watching For the File Changes

Another handy task that to use in developer's environment is the `watch` task. The purpose of this task is to monitor files in pre-configured locations. When the watcher detects any changes in those files it will run the configured task. Here is how a `watch` task config looks like:

*Example 5-4. A watch task config*

```
module.exports = function(grunt) {
  grunt.initConfig({
    jshint: {
      // ... configuration code is omitted
    },
  });
};
```

```

watch: {           // ①
  reload: {
    files: ['app/*.html', 'app/data/**/*.json', 'app/assets/css/*.css', 'app/js/**/*.js',
      'test/test/tests.js', 'test/spec/*.js'], // ②
    tasks: ['jshint']                      // ③
  }
}
});;
grunt.loadNpmTasks('grunt-contrib-jshint'); // ④
grunt.loadNpmTasks('grunt-contrib-watch');
grunt.registerTask('default', ['jshint']);
};

```

- ① The watch task configuration starts here
- ② The list of the files that need to be monitored for changes
- ③ A array of tasks to be triggered after file change event occurs
- ④ The `grunt-contrib-watch` plugin has to be installed.

You can run grunt watch from the command line (keep in mind that it never ends on its own).

```

> grunt watch

Running "watch" task
Waiting...OK
>> File "app/js/Player.js" changed.
Running "jshint:gruntfile" (jshint) task
>> 1 file lint free.

Running "jshint:app" (jshint) task
>> 1 file lint free.

Done, without errors.

Completed in 0.50s at Tue May 07 2013 00:41:42 GMT-0400 (EDT) - Waiting...

```



The article [Grunt and Gulp Tasks for Performance Optimization](#) lists various useful Grunt tasks for optimizing loading of images and CSS.

## Bower

**Bower** is a package manager for Web projects. Twitter has donated it to the open-source community. Bower is a utility and a community driven repository of libraries that help in downloading the third-party software required for the application code that will run

in a Web browser. The Bower's purpose is very similar to NPM, but the latter is more suitable for the server-side projects.

Bower can take care of transitive (dependency of a dependency) dependencies and download all required library components. Each Bower's package has a bower.json file, which contains the package metadata for managing the package's transitive dependencies. Also, bower.json can contain information about the package repository, readme file, license et al. You can find bower.json in the root directory of the package. For example, *components/requirejs/bower.json* is a path for the RequireJS metadata file. Bower can be installed via NPM. The following line shows how to install Bower globally in your system.

```
npm install -g bower
```



Java developers use package managers like Gradle or Maven that have similar to Bower functionality.

Let's start using Bower now. For example, here is a Bower's command to install the library RequireJS.

```
bower install requirejs --save
```

Bower installs RequireJS into *components/requirejs* directory and saves information about dependencies in bower.json configuration file.

Bower simplifies the delivery of dependencies into target platform, which means that you don't need to store dependencies of your application in the source control system. Just keep your application code there and let Bower to bring all other dependencies described in its configuration file.



There are pros and cons for storing dependencies in the source control repositories. Read the [article by Addi Osmani](#) that covers this subject in more detail.

Your application will have its own file bower.json with the list of the dependencies. At this point, Bower can install all required application dependencies with one command - `bower install`, which will deliver all your dependency files into the `components` directory. Here is the content of the file bower.json for our Save The Child application.

```
{
  "name": "ch7_dynamic_modules",
  "description": "Chapter 7: Save The Child, Dynamic Modules app",
```

```
    "dependencies": {  
        "requirejs": "~2.1.5",  
        "jquery": ">= 1.8.0",  
        "qunit": "~1.11.0",  
        "modernizr": "~2.6.2",  
        "requirejs-google-maps": "latest"  
    }  
}
```

Application dependencies are specified in corresponding “dependencies section. The `>=` sign specifies that the corresponding software has to be not older than the specified version.

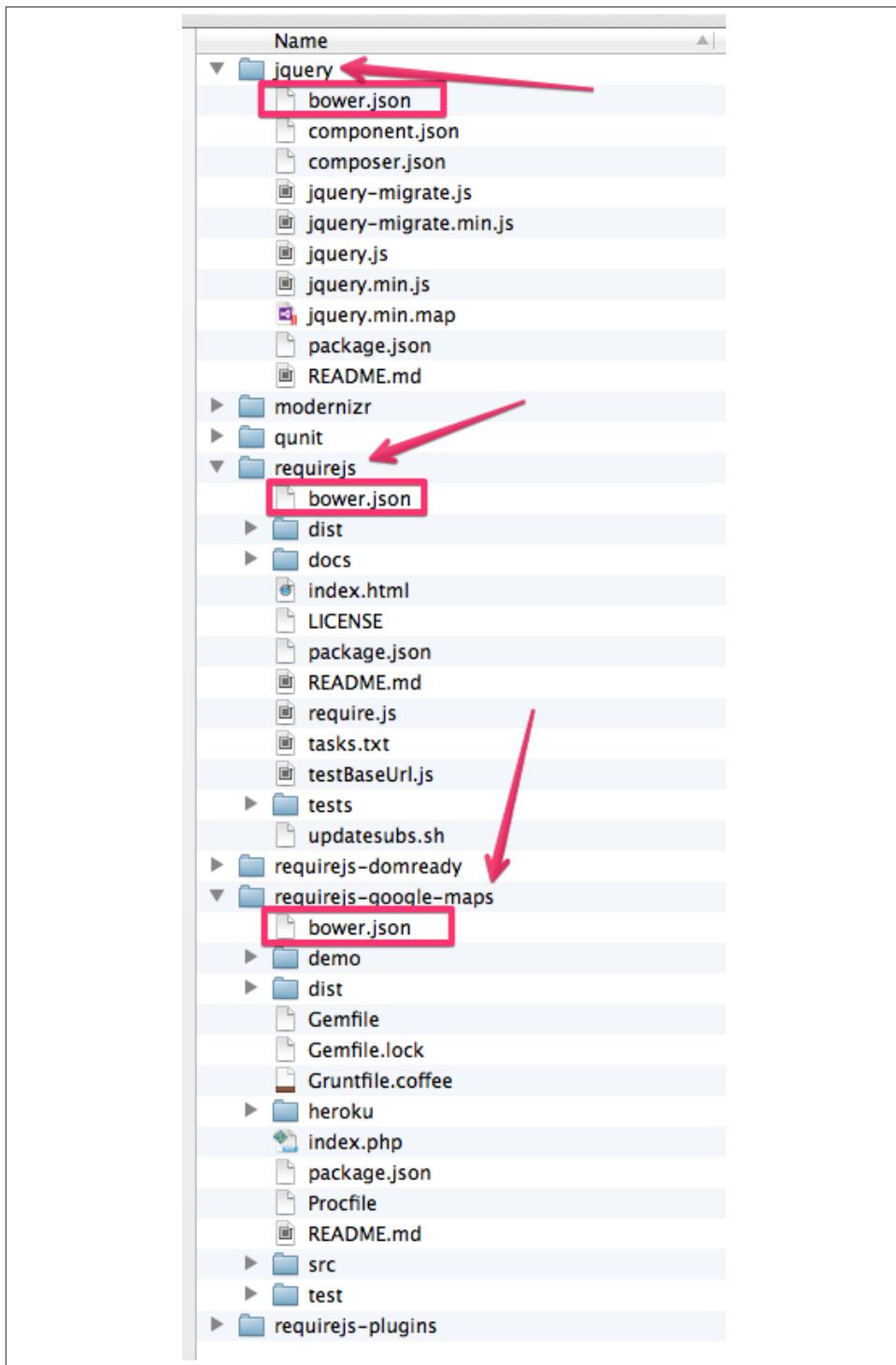


Figure 5-1. Directory structure of application's components

Also, there is a [Bower search tool](#) to find the desired component in its repository.

## Yeoman

**Yeoman** is a collection of tools and best practices that help to bootstrap a new web project. Yeoman consists from three main parts: Grunt, Bower and Yo. Grunt and Bower were explained earlier in this chapter.

Yo is a code-generation tool. It makes the start of the project faster by scaffolding a new JavaScript application. Yo can be installed via NPM similar to the other tools. The following commands shows how to install Yo globally in your system. And if you didn't have Grunt and Bower installed before, this command will install them automatically.

```
npm install -g yo
```

For code-generation, Yo relies on plugins called *generators*. Generator is a set of instructions to Yo and file templates. You can use [Yeoman Generators search tool](#) to discover community-developed generators. At the time of this writing you can use one of about 430 community-developed generators to scaffold your project.

For example, let's scaffold the Getting Started project for RequireJS. RequireJS is a framework that helps to dice code of your JavaScript application into modules. We will cover this framework in details later in «Modularizing Large-Scale JavaScript Projects» chapter.

YEOMAN GENERATORS						
requirejs						
Name	Description	Author	Last modified	Stars	Forks	
<a href="#">footguard</a>	Single Page HTML application (Backbone, RequireJS, Sass, CoffeeScript et Bootstrap)	Mathieu Desv��	about 9 hours ago	14	6	
<a href="#">requirejs</a>	RequireJS, Grunt, Bower and QUnit all working together for awesomeness	danheberden	13 days ago	9	7	
<a href="#">bbr</a>	Backbone.js applications using RequireJS.	Matt Przybylski	13 days ago	3	0	
<a href="#">requirejs-jasmine-karma</a>	A generator for yeoman (RequireJS, Jasmine, Karma)	aleksandara	2 days ago	2	2	
<a href="#">amdblah</a>	A [Yeoman](http://yeoman.io) generator for starting a project with Express, RequireJS, Backbone.js + Handlebars both on server and client side, f18next, hsfeng Moment.js and Bootstrap.	hsfeng	3 days ago	2	0	
<a href="#">pr0d</a>	Single Page HTML application (Backbone, RequireJS, Sass, CoffeeScript et Bootstrap)	Alexandre Koch	about 15 hours ago	1	1	
<a href="#">frontend-php</a>	Frontend generator supporting RequireJS, different test and css frameworks and PHP	Ben Z��rb	2013-11-09T22:09:08Z	1	0	
<a href="#">cornelio</a>	Cornelio WebApp, with Backbone, requirejs, karma, less, bootstrap, Handlebars.	Ricardo Rivas Gonz��lez	2013-11-15T20:01:29Z	0	0	

This generator listing is automatically generated from the npm module database. In order for a Yeoman generator to be listed here, it must be published on npm with the "yeoman-generator" keyword and have a description.

Figure 5-2. Yeoman Generators search tool

The search tool found bunch of generators that have keyword `requirejs` in their name or description. We're looking for generator that called "requirejs" (**highlighted with red square**). When we click on name link, the [Github page of requirejs generator](#) will be displayed. Usually, the generator developers provide a reference of the generator's available tasks.

Next we need to install generator on our local machine with following command:

```
npm install -g generator-requirejs
```

After installation, we can start `yo` command and as a parameter we need to specify generator's name. To start scaffolding a RequireJS application we can use following command:

```
yo requirejs
```

We need to provide answers to the wizard's questions.

Example 5-5. Yeoman prompt

```
  _-----_
 |         |
 |  --(o)--| .-----.
 |-----| |   Welcome to Yeoman,
 |  _`U`_| |   ladies and gentlemen!  |
 /__A__\| |-----|
 | ~ |
```

' . '-' ' | ° ' ' Y ' .

This comes with requirejs, jquery, and grunt all ready to go  
[?] What is the name of your app? requirejs yo  
[?] Description: description of app for package.json  
create Gruntfile.js  
create package.json  
create bower.json  
create .gitignore  
create .jshintrc  
create .editorconfig  
create CONTRIBUTING.md  
create README.md  
create app/.jshintrc  
create app/config.js  
create app/main.js  
create test/.jshintrc  
create test/index.html  
create test/tests.js  
create index.htm

I'm all done. Running bower install & npm install for you to install the required dependencies. If thi

.... npm install output is omitted

You will get all directories and files set up, and you can start writing your code immediately. The structure of your project will be reflecting common best practices from JavaScript community ([refer to following figure](#)).

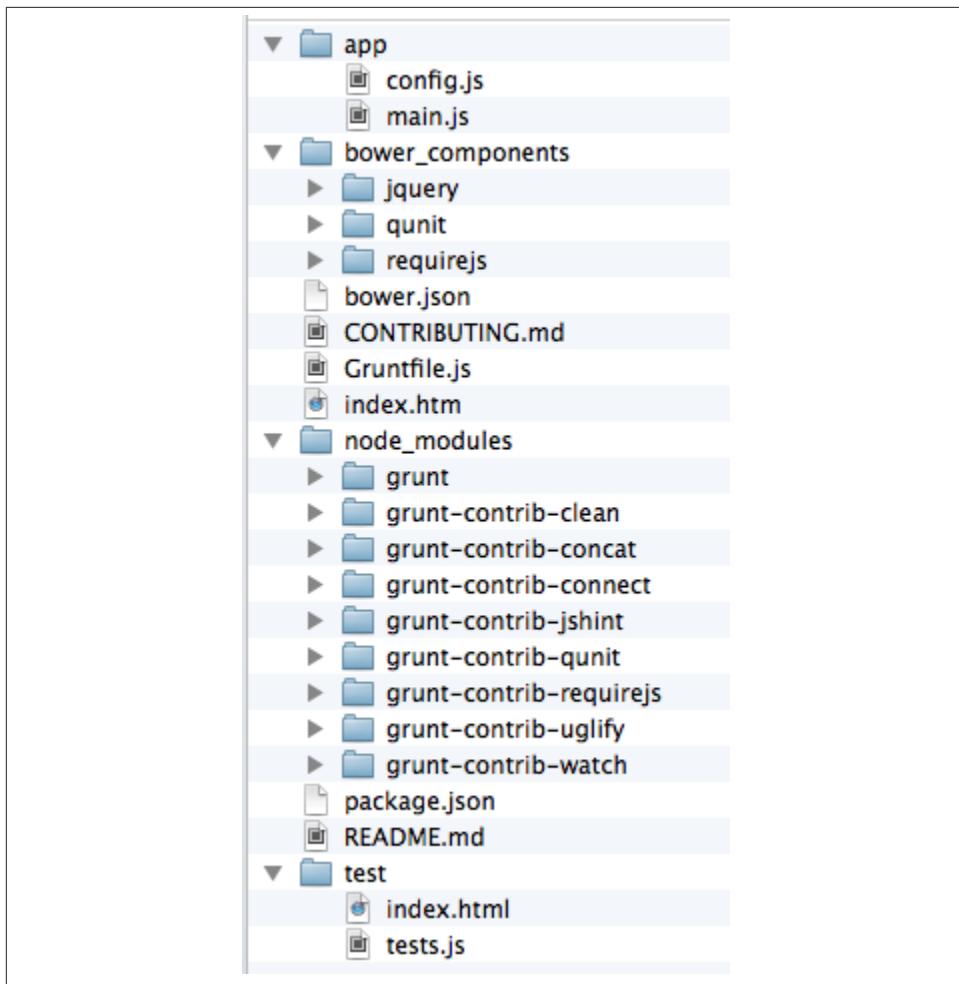


Figure 5-3. Scaffolded RequireJS application directory structure

After executing the `yo` command you will get Grunt set up with following configured tasks:

- `clean`: Clean files and folders.
- `concat`: Concatenate files.
- `uglify`: Minify files with UglifyJS.
- `qunit`: Run QUnit unit tests in a headless PhantomJS instance.
- `jshint`: Validate files with JSHint.
- `watch`: Run predefined tasks whenever watched files change.

- `requirejs`: Build a RequireJS project.
- `connect`: Start a connect web server.
- `default`: Alias for “`jshint`”, “`qunit`”, “`clean`”, “`requirejs`”, “`concat`”, “`uglify`” tasks.
- `preview`: Alias for “`connect:development`” tas\* `preview-live` Alias for “`default`”, “`connect:production`” tasks.

Yeoman also has **generator for generator scaffolding**. It might be very useful if in your want to introduce your own workflow for web project.

The next code generator that we'll cover is a more specific one - it can generates the entire ExtJS-Java application.

## Productive Enterprise Web Development with Ext JS and CDB

Authors of this book work for the company called Farata Systems, which has developed an open source freely available software Clear Toolkit for Ext JS, and the code generator and Eclipse IDE plugin CDB comes with it. CDB is a productivity tool that was created specifically for the enterprise applications that use Java on the server side and need to retrieve, manipulate, and save the data in some persistent storage.

Such enterprise applications are known as *CRUD applications* because they perform Create-Retrieve-Update-Delete operations with data. If the server side of your Web application is developed in Java, with CDB you can easily generate a CRUD application, where Ext JS front end communicates the Java back end. In this section you will learn how jump start development of such CRUD Web applications.

Familiarity with core Java concepts like classes, constructors, getters and setters, and annotations is required for understanding of the materials of this section.

The phrase *to be more productive* means to write less code while producing the results faster. This is what CDB is for, and you'll see it helps you to integrate the client side with the back end using the RPC style and how to implements data pagination for your application. To be more productive, you need to have the proper tools installed and we'll cover this next.

## Ext JS MVC Application Scaffolding

In this section we'll cover the following topics:

- What is Clear Toolkit for Ext JS
- How to create an Ext JS MVC front end for a Java-based project

- How to deploy and run your first Ext JS and Java application on Apache Tomcat server

Clear Toolkit for Ext JS includes the following:

- Clear Data Builder - an Eclipse plugin that supports code generation Ext JS MVC artifacts based on the code written in Java. CDB comes with wizards to start new project with plain Java or with popular frameworks like Hibernate, Spring, MyBatis.
- Clear JS - a set of JavaScript components that extends Ext JS standard components. In particular, it includes a `ChangeObject` that traces the modifications of any item in a store.
- Clear Runtime - Java components that implements server side part of `ChangeObject`, `DirectOptions` an others.

CDB distribution available as plug-in for a popular among Java developers Eclipse IDE. The current update site of CDB is located [here](#). The current version is 4.1.4. You can install this plug-in via the `Install New Software` menu in Eclipse IDE. The [Figure 5-4](#) shows “Clear Data Builder for Ext JS feature” in the list of Installed Software in your Eclipse IDE, which means that CDB is installed.

You have to have work with “Eclipse IDE for Java EE Developers”, which includes plugins for automation of the Web application development.

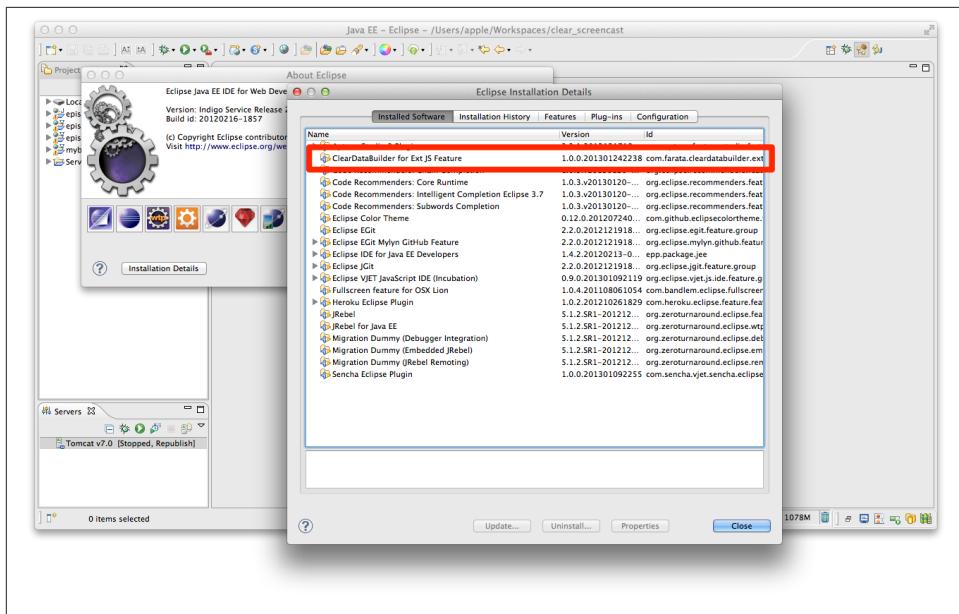


Figure 5-4. Verifying CDB installation

Clear Data Builder comes with a set of prepared examples that demonstrate the integration with popular Java frameworks - MyBatis, Hibernate, and Spring. There is also a plain Java project example that doesn't use any persistence frameworks. Let's start with the creation of the new project by selecting Eclipse menu File → New → Other → Clear. You'll see a window similar to [Figure 5-5](#).

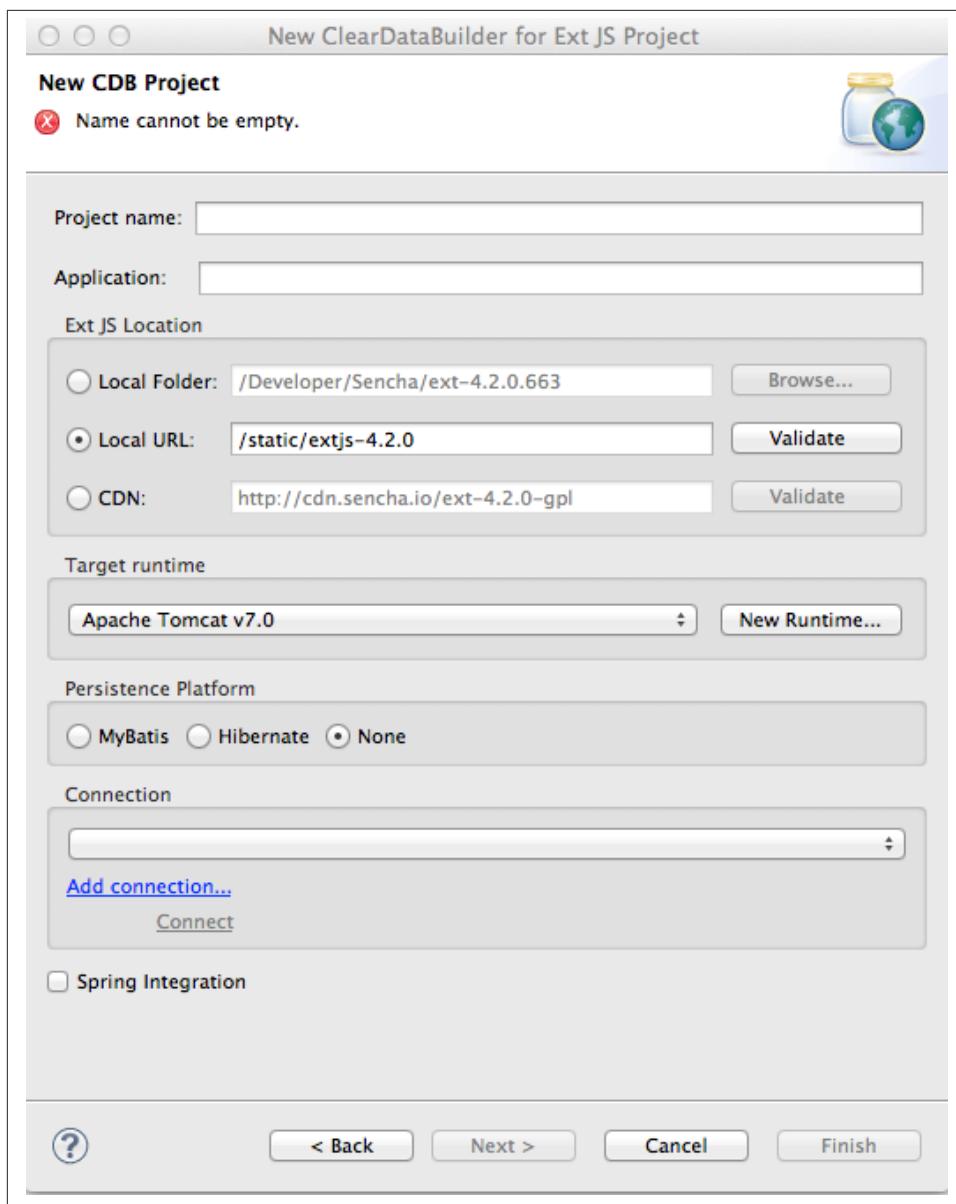


Figure 5-5. New CDB Project Wizard

Name the new project `episode_1_intro`. CDB supports different ways of linking the Ext JS framework to the application. CDB automatically copies the Ext JS framework under the Web server (Apache Tomcat in our case). We're going to use this local Ext JS URL, but you can specify any folder in your machine and CDB will copy the Ext JS file

from there into your project. You can also use Ext JS from the Sencha's CDN, if you don't want to store these libraries inside your project. Besides, using a common CDN will allow Web browser to reuse the cached version of Ext JS.

For this project we are not going to use any server-side persistence frameworks like MyBatis or Hibernate. Just click the button Finish, and you'll see some initial CDB messages on the Eclipse console. When CDB runs for the first time it creates in your project's WebContent folder the directory structure recommended by Sencha for MVC applications. It also generates `index.html` for this application, which contains the link to the entry point of our Ext JS application.

CDB generates an empty project with one sample controller and one view - `Viewport.js`. To run this application, you need to add the newly generated Dynamic Web Project to Tomcat and start the server (right-click on the Tomcat in the Servers view of Eclipse IDE).

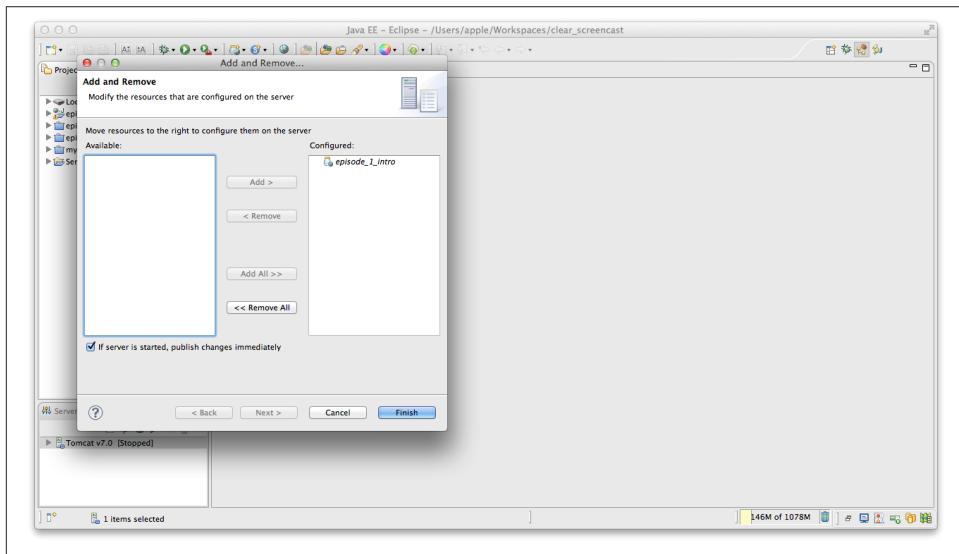


Figure 5-6. Adding web project to Tomcat

Open this application in your Web browser at `http://localhost:8080/episode_1_intro`. Voila! In less than a couple of minutes we've created a new Dynamic Web Project with the Ext JS framework and one fancy button as shown on [Figure 5-7](#).

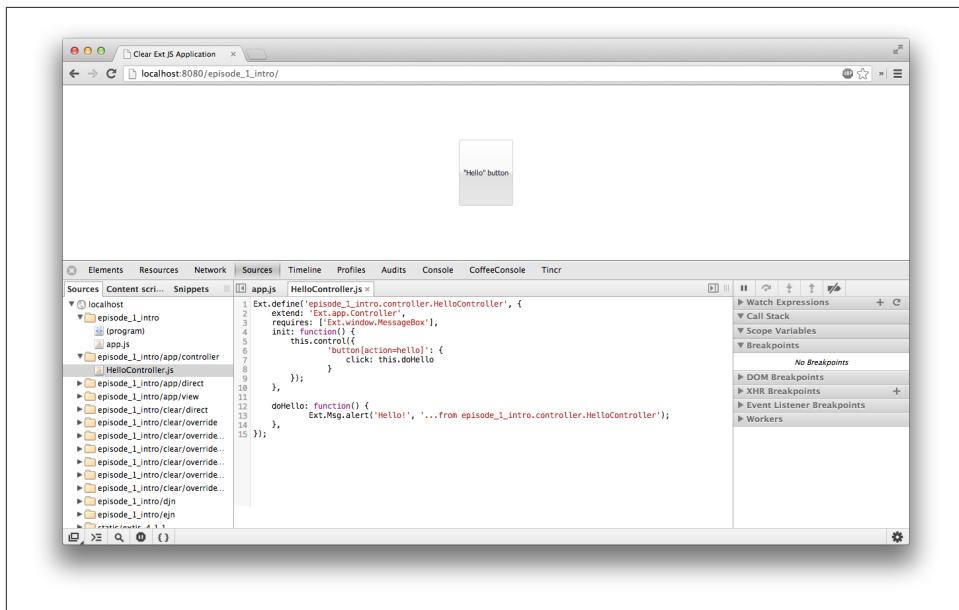


Figure 5-7. Running scaffolded application

The next step is to make something useful out of this basic application.

## Generating a CRUD application

The Part Two of the CDB section covers the process of creation of a simple CRUD application that uses Ext JS and Java. We'll go through the following steps:

- Create a plain old Java object (POJO) and the corresponding `Ext.data.Model`
- Create a Java service and populate `Ext.data.Store` with data from service
- Use the auto-generated Ext JS application
- Extend the auto-generated CRUD methods
- Use `ChangeObject` to track the data changes

Now let's use CDB to create a CRUD application. You'll learn how turn a POJO into an Ext JS model, namely:

- how to populate the Ext JS store from a remote service
- how to use automatically generated UI for that application
- how to extend the UI

- what the `ChangeObject` class is for

First, we'll extend the application from Part One - the CRUD application needs a Java POJO. To start, create a Java class `Person` in the package `dto`. Then add to this class the properties (as well as getters and setters) `firstName`, `lastName`, `address`, `ssn` and `phone` and `id`. Add the class constructor that initializes these properties as shown in the code listing below.

*Example 5-6. Person data transfer object*

```
package dto;

import com.farata.dto2extjs.annotations.JSClass;
import com.farata.dto2extjs.annotations.JSGeneratedId;

@JSClass
public class Person {

    @JSGeneratedId
    private Integer id;
    private String firstName;
    private String lastName;
    private String phone;
    private String ssn;

    public Person(Integer id, String firstName, String lastName,
                  String phone, String ssn) {
        super();
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.phone = phone;
        this.ssn = ssn;
    }

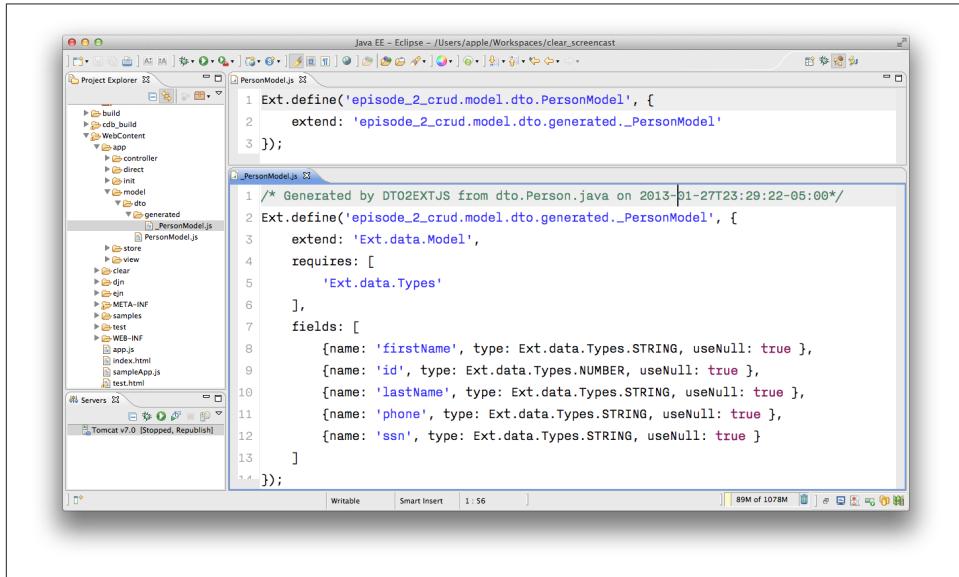
    // Getters and Setters are omitted for brevity
}
```

You may also add a `toString()` method to the class. Now you'll need the same corresponding Ext JS model for the Java class `Person`. Just annotate this Java class with the annotation `@JSClass` and CDB generates the Ext JS model.



CDB integrates into standard Eclipse build life-cycle. You don't need to trigger a code generation procedure manually. If you have "Build Automatically" option selected in Project menu, a code generation starts immediately you've saved the file.

The next step is to annotate the `id` field with the CDB annotation `@JSGeneratedId`. This annotation instructs CDB to treat this field as an auto generated id. Let's examine the directory of Ext JS MVC application to see what's inside the model folder. In the JavaScript section there is the folder `dto` which corresponds to the Java `dto` package where the `PersonModel` resides as illustrated on [Figure 5-8](#).



*Figure 5-8. Generated from Java class Ext JS model*

Clear Data Builder generated two files as recommended by the [Generation Gap pattern](#), which is about keeping the generated and handwritten parts separate by putting them in different classes linked by inheritance. Let's open the person model. In our case the `PersonModel.js` is extended from the `generated_PersonModel.js`. Should we need to customize this class, we'll do it inside the `Person.js`, but this underscore-prefixed file will be regenerated each and every time when we change something in our model. CDB follows this pattern for all generated artifacts - Java services, Ext JS models and stores. This model contains all the fields from our Person DTO.

Now we need to create a Java service to populate the Ext JS store with the data. Let's create a Java interface `PersonService` in the package `service`. This service will return the list of `Person` objects. This interface contains one method -`List<Person> getPersons()`.

To have CDB to expose this service as a remote object, we'll use the annotation called `@JSService`. Another annotation `@JSGenerateStore` will instruct CDB to generate the store. In this case CDB will create the *destination-aware store*. This means that store

will know from where to populate its content. All configurations of the store's proxies will be handled by the code generator. With `@JSFillMethod` annotation we will identify our main read method (the "R" from CRUD).

Also it would be nice to have some sort of a sample UI to test the service - the annotation `@JSGenerateSample` will help here. CDB will examine the interface `PersonService`, and based on these annotations will generate all Ext JS MVC artifacts (models, views, controller) and the sample application.

*Example 5-7. PersonService interface annotated with CDB annotations*

```
@JSService
public interface PersonService {
    @JSGenerateStore
    @JSFillMethod
    @JSGenerateSample
    List<Person> getPersons();
}
```

When the code generation is complete, you'll get the implementation for the service - `PersonServiceImpl`. The store folder inside the application folder (`WebContent/app`) has the Ext JS store, which is bound to the previously generated `PersonModel`. In this case, CDB generated store that binds to the remote service.

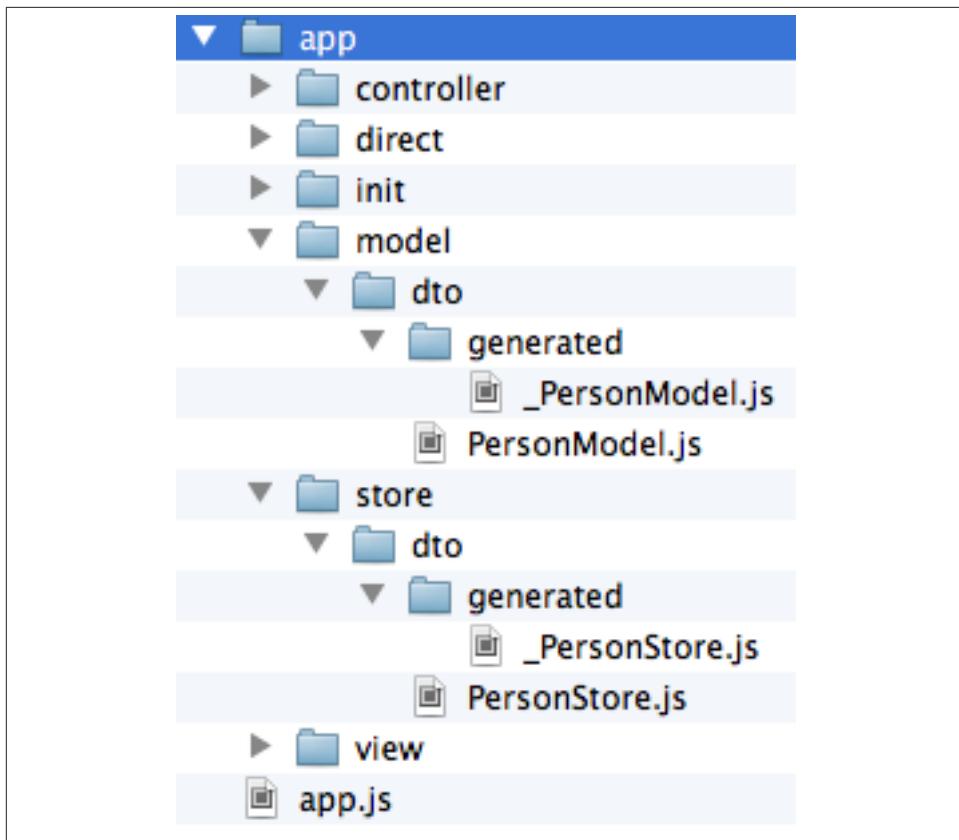


Figure 5-9. Structure of store and model folders

All this intermediate translation from the JavaScript to Java and from Java to JavaScript is done by DirectJNgine, which is a server side implementation of the Ext Direct Protocol. You can read about this protocol in [Ext JS documentation](#).

CDB has generated a sample UI for us too. Check out the samples directory shown on [Figure 5-10](#).

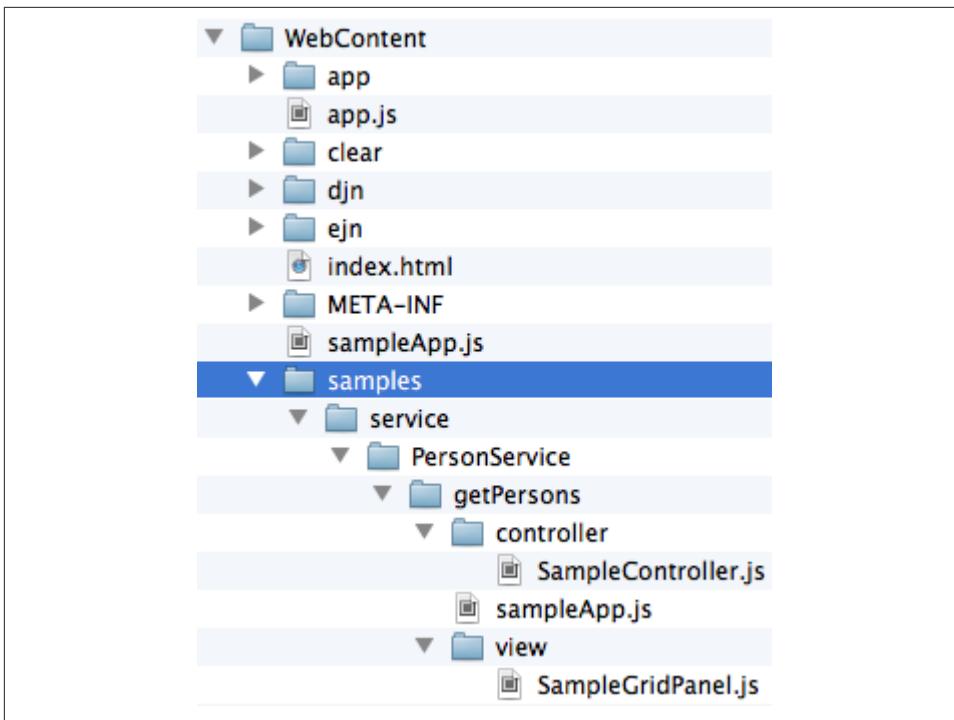


Figure 5-10. Folder with generated UI files

CDB has generated `SampleController.js`, `SampleGridPanel.js`, and the Ext JS application entry point `sampleApp.js`. To test this application just copy the file `SampleController.js` into the controller folder, `SampleGridPanel.js` panel into the view folder, and the sample application in the root of the WebContent folder. Change the application entry point with to be `sampleApp.js` in the `index.html` of the Eclipse project as shown below.

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

This is how the generated UI of the sample application looks like [Figure 5-11](#).

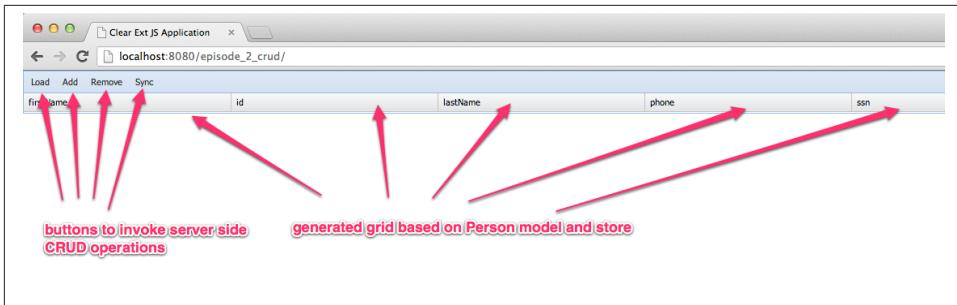


Figure 5-11. Scaffolded CRUD application template

On the server side, CDB also follows the *Generation Gap Pattern* and it generated stubs for the service methods. Override these methods when you're ready to implement the CRUD functionality, similar to the below code sample.

#### Example 5-8. Implementation of PersonService interface

```
package service;
import java.util.ArrayList;
import java.util.List;

import clear.data.ChangeObject;
import dto.Person;
import service.generated.*;

public class PersonServiceImpl extends _PersonServiceImpl { // ①

    @Override
    public List<Person> getPersons() { // ②
        List<Person> result = new ArrayList<>();
        Integer id= 0;
        result.add(new Person(++id, "Joe", "Doe",
                             "555-55-55", "1111-11-1111"));
        return result; // ③
    }

    @Override
    public void getPersons_doCreate(ChangeObject changeObject) { // ④
        Person dto = (Person) deserializeObject(
            (Map<String, String>) changeObject.getNewVersion(),
            Person.class);
        System.out.println(dto.toString());
    }
}
```

```

    }

    @Override
    public void getPersons_doUpdate(ChangeObject changeObject) { // ⑤
        // TODO Auto-generated method stub
        super.getPersons_doUpdate(changeObject);
    }

    @Override
    public void getPersons_doDelete(ChangeObject changeObject) { // ⑥
        // TODO Auto-generated method stub
        super.getPersons_doDelete(changeObject);
    }
}

```

- ① Extend the generated class and provide the actual implementation
- ② The `getPerson()` is our retrieve (fill) method (the R in CRUD)
- ③ For this sample application we can use `java.util.ArrayList` class as in-memory server side storage of the Person objects. In the real world applications you'd use a database or other persistent storage
- ④ `fillMethodName +_doCreate()` is our create method (the C in CRUD)
- ⑤ `fillMethodName +_doUpdate()` is our update method (the U in CRUD)
- ⑥ `fillMethodName +_doDelete()` is our delete method (the D in CRUD)

Click on the Load menu on the UI, and the application will retrieve four persons from our server

To test the rest of the CRUD methods, we'll ask the user to insert one new row, modify three existing ones and remove two rows using the generated Web client. The `Clear.data.DirectStore` object will automatically create a collection of six 'ChangeObject's - one to represent a new row, three to represent the modified ones, and two for the removed rows.

When the user clicks on the Sync UI menu the changes will be sent to the corresponding `do...` remote method. When you `sync()` a standard `Ext.data.DirectStore` Ext JS is POST-ing new, modified and deleted items to the server. When the request is complete the server's response data is applied to the store expecting that some items can be modified by the server. In case of `Clear.data.DirectStore` instead of passing around items, we pass the deltas, wrapped in the `ChangeObject`.

Each instance of the `ChangeObject` contains the following:

- `newVersion` - it's an instance of the newly inserted or modified item. On the Java side it's available via `getNewVersion()`.

- `prevVersion` - it's an instance of the deleted old version of the modified item. On the Java side it's available via `getPrevVersion()`.
- array of `changepropertyName`s if this `ChangeObject` represents an update operation.

The rest of `ChangeObject` details described on the [Clear Toolkit Wiki](#).

The corresponding Java implementation of `ChangeObject` is available on the server side and Clear Toolkit passes `ChangeObject` instances to the appropriate `do*` method of the service class. Take a look at the `getPersons_doCreate()` method from [Example 5-8](#). When the server needs to read the new or updated data arrived from the client your Java class has to invoke the method `changeObject.getNewVersion()`. This method will return the JSON object that you need to deserialize into the object `Person`. This is done in [Example 5-8](#) and looks like this.

```
Person dto = (Person) deserializeObject(
    (Map<String, String>) changeObject.getNewVersion(), Person.class);
```

When the new version of the `Person` object is extracted from the `ChangeObject` you can do with it whatever has to be done to persist it in the appropriate storage. In our example we just print the new person information on the server-side Java console. This is why we said earlier, that it may be a good idea to provide a pretty printing feature on the class `Person` by overriding method `toString()`. Similarly, when you need to do a delete, the `changeObject.getPrevVersion()` would give you a person to be deleted.

## Data Pagination

The pagination feature is needed in almost every enterprise web application. Often you don't want to bring all the data to the client at once - a page by page feed brings the data to the user a lot faster. The user can navigate back and forth between the pages using pagination UI components. To do that, we need to split our data on the server side into chunks, to send them page by page by the client request. Implementing pagination is the agenda for this section. We'll do the following:

- Add the data pagination to our sample CRUD application:
  - Add the `Ext.toolbar.Paging` component
  - Bind both `grid` and `pagingtoolbar` to the same store
  - Use `DirectOptions` class to read the pagination parameters

We are going to improve our CRUD application by adding the paging toolbar component bound to the same store as the grid. The class `DirectOptions` will handle the pagination parameters on the server side.

So far CDB has generated the UI from the Java back end service as well as the Ext JS store and model. We'll refactor the service code from previous example to generate more data (a thousand objects) so we have something to paginate, see below.

*Example 5-9. Refactored implementation of PersonService Interface*

```
public class PersonServiceImpl extends _PersonServiceImpl {  
    @Override  
    public List<Person> getPersons() {  
        List<Person> result = new ArrayList<>();  
        for (int i=0; i<1000; i++){  
            result.add(new Person(i, "Joe", "Doe", "555-55-55",  
                               "1111-11-1111"));  
        }  
        return result;  
    }  
}
```

If you'll re-run the application now, the Google Chrome Console will show that PersonStore is populated with one thousand records. Now we'll add the the Ext JS paging toolbar/paging UI component to the file sampleApp.js as shown below.

*Example 5-10. Sample Application Entry*

```
Ext.Loader.setConfig({  
    disableCaching : false,  
    enabled : true,  
    paths : {  
        episode_3_pagination : 'app',  
        Clear : 'clear'  
    }  
});  
  
Ext.syncRequire('episode_3_pagination.init.InitDirect');  
// Define GridPanel  
var myStore = Ext.create('episode_3_pagination.store.dto.PersonStore',{}); //❶  
Ext.define('episode_3_pagination.view.SampleGridPanel', {  
    extend : 'Ext.grid.Panel',  
    store : myStore,  
    alias : 'widget.samplegridpanel',  
    autoscroll : true,  
    plugins : [{  
        ptype : 'cellediting'  
    }],  
    dockedItems: [  
        {  
            xtype: 'pagingtoolbar', //❷  
            displayInfo: true,  
            dock: 'top',  
            store: myStore //❸  
        }  
    ],  
},  
    {  
        xtype: 'pagingtoolbar', //❷  
        displayInfo: true,  
        dock: 'top',  
        store: myStore //❸  
    }  
],
```

```

columns : [
    {header : 'firstName', dataIndex : 'firstName',
        editor : {xtype : 'textfield'}, flex : 1 },
    {header : 'id', dataIndex : 'id', flex : 1 },
    {header : 'lastName', dataIndex : 'lastName',
        editor : {xtype : 'textfield'}, flex : 1 },
    {header : 'phone', dataIndex : 'phone',
        editor : {xtype : 'textfield'}, flex : 1 },
    {header : 'ssn', dataIndex : 'ssn',
        editor : {xtype : 'textfield'}, flex : 1 }],
tbar : [
    {text : 'Load', action : 'load'},
    {text : 'Add', action : 'add'},
    {text : 'Remove', action : 'remove'},
    {text : 'Sync', action : 'sync'}
]
});
// Launch the application
Ext.application({
    name : 'episode_3_pagination',
    requires : ['Clear.override.ExtJSOverrider'],
    controllers : ['SampleController'],
    launch : function() {
        Ext.create('Ext.container.Viewport', {
            items : [
                {xtype : 'samplegridpanel'}
            ]
        });
    }
});

```

- ① Manual store instantiation - create a separate variable `myStore` for this store with empty `config` object
- ② Adding the `xtype pagingtoolbar` to this component docked items property to display the information and dock this element at the top.
- ③ Now the paging toolbar is also connected to same store.

The next step is to fix the automatically generated controller to take care of the loading of data on click of Load button as shown in the code below.

#### *Example 5-11. Controller for sample application*

```

Ext.define('episode_3_pagination.controller.SampleController', {
    extend: 'Ext.app.Controller',
    stores: ['episode_3_pagination.store.dto.PersonStore'],
    refs: [{ //①
        ref: 'ThePanel',
        selector: 'samplegridpanel'
    }],

```

```

init: function() {
    this.control({
        'samplegridpanel button[action=load]': {
            click: this.onLoad
        }
    });
},
onLoad: function() {
    // returns instance of PersonStore
    var store = this.getThePanel().getStore(); //②
    store.load();
}
);

```

- ❶ Bind the store instance to our grid panel. In controller's `refs` property we're referencing our `simplegrid` panel with `ThePanel` alias.
- ❷ In this case there is no need to explicitly retrieve the store instance by name. Instead, we can use getters `getPanel()` and `getStore()` automatically generated by the Ext JS framework.

When the user clicks the button *next* or *previous* the method `loadPage` of the underlying store is called. Let's examine the `directprovider` URL - the server side router of the remoting calls - to see how the direct request looks like. Open Google Chrome Developer Tools from the menu `View → Developer`, refresh the Web page and go to the Network tab. You'll see that each time the user clicks on the *next* or *previous* buttons on the pagination toolbar the component sends `directOptions` as a part of the request.

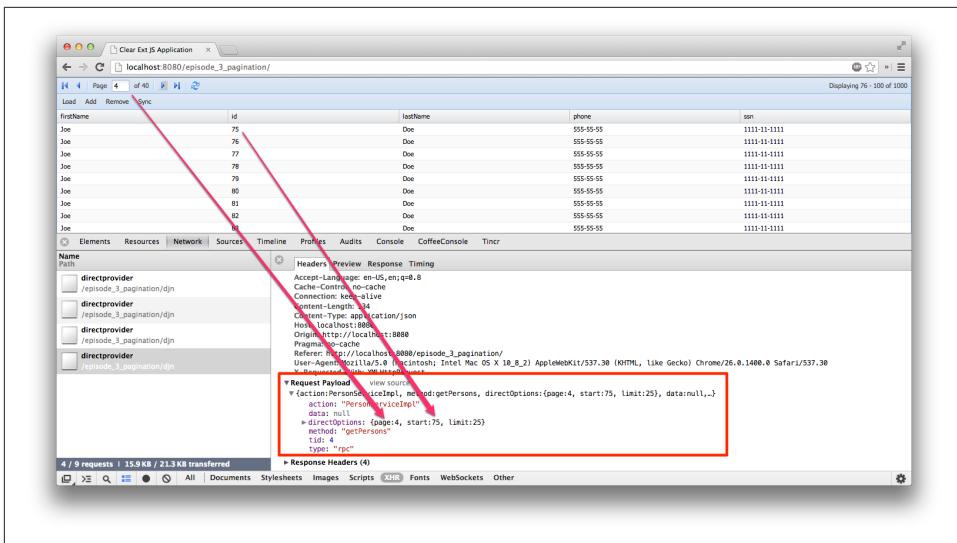


Figure 5-12. Request payload details

The default Ext Direct request doesn't carry any information about the page size. Clear JS has the client side extension of the Ext JS framework that adds some extra functionality to `Ext.data.DirectStore` component to pass the `start` and `limit` values to the server side. At this point, the `directOptions` request property (see Figure 5-12) can be extracted on the server side to get the information about the page boundaries. Let's add some code to the `PersonServiceImpl.java`. At this point the pagination doesn't work. The server sends the entire thousand records, because it doesn't know that the data has to be paginated. We'll fix it in the following listing.

#### Example 5-12. Implementation of PersonService With Pagination

```

package service;
import java.util.ArrayList;
import java.util.List;

import clear.djn.DirectOptions;           //①

import dto.Person;
import service.generated.*;

public class PersonServiceImpl extends _PersonServiceImpl {
  @Override
  public List<Person> getPersons() {
    List<Person> result = new ArrayList<>();
    for (int i=0; i<1000; i++){
      result.add(new Person(i, "Joe", "Doe", "555-55-55", "1111-11-1111"));
    }
  }                                     //②
}
  
```

```

    int start = ((Double)DirectOptions.getOption("start")).intValue();
    int limit = ((Double)DirectOptions.getOption("limit")).intValue();

    limit = Math.min(start+limit, result.size() );    //③
    DirectOptions.setOption("total", result.size());   //④
    result = result.subList(start, limit);           //⑤

    return result;
}
}

```

- ➊ On the server side there is a special object called `DirectOptions`, which comes with Clear Toolkit.
- ➋ We want to monitor the `start` and `limit` values (see [Figure 5-12](#)).
- ➌ Calculate the actual limit. Assign the size of the data collection to the `limit` variable if it's less than the page size (`start+limit`).
- ➍ Notify the component about the total number of elements on the server side by using `DirectOptions.setOption()` method with `total` option.
- ➎ Before returning the result, create a subset, an actual page of data using the method `java.util.List.sublist()` which produces the view of the portion of this list between indexes specified by the `start` and the `limit` parameters.

As you can see from the Network tab in [Figure 5-11](#), we've limited the data load to 25 elements per page. Clicking on next or previous buttons will get you only a page worth of data. The Google Chrome Developers Tools Network tab shows that that we are sending the `start` and `limit` values with every request, and the response contains the object with 25 elements.

If you'd like to repeat all of the above steps on your own, watch [the screencasts](#) where we demonstrate all the actions described in the section on CDB. For the current information about CDB visit [cleardb.io](#).

## Summary

Writing the enterprise web applications can be a tedious and time-consuming process. A developer needs to set up frameworks, boilerplates, abstractions, dependency management, build processes and the list of requirements for a front-end workflow appears to grow each year. In this chapter we introduced several tools that could help you with automating a lot of mundane tasks and make you more productive.

## CHAPTER 6

# Modularizing Large-Scale JavaScript Projects

Reducing the application startup latency and implementing lazy loading of certain parts of the application are the main reasons for modularization.

A good illustration of why you may want to consider modularization is a well designed Web application of the [Mercedes Benz USA](#). This Web application serves people who live in America and either own or consider purchasing cars from this European car manufacturer.

One of their purchasing options is called “European Delivery”. An American resident who chooses this particular package can combine a vacation with her car purchase. She flies to the Mercedes Benz factory in Europe, picks up her car, and has a two week vacation, driving her new vehicle around in Europe. After the vacation is over, the car is shipped to her hometown in the US. Needless to say, this program adds several thousand dollars to the price of the car.

From an application design point of view, we don’t need to include the code that supports the European Delivery to each and every user’s who decided to visit mbusa.com. *If* the user visits the menu Owners and clicks on the European Delivery link, then *and only then* the required code and resources be pushed to the user’s computer or mobile device.

The snapshot in [Figure 6-1](#) was taken after clicking on this link with Chrome Developer tools panel open.

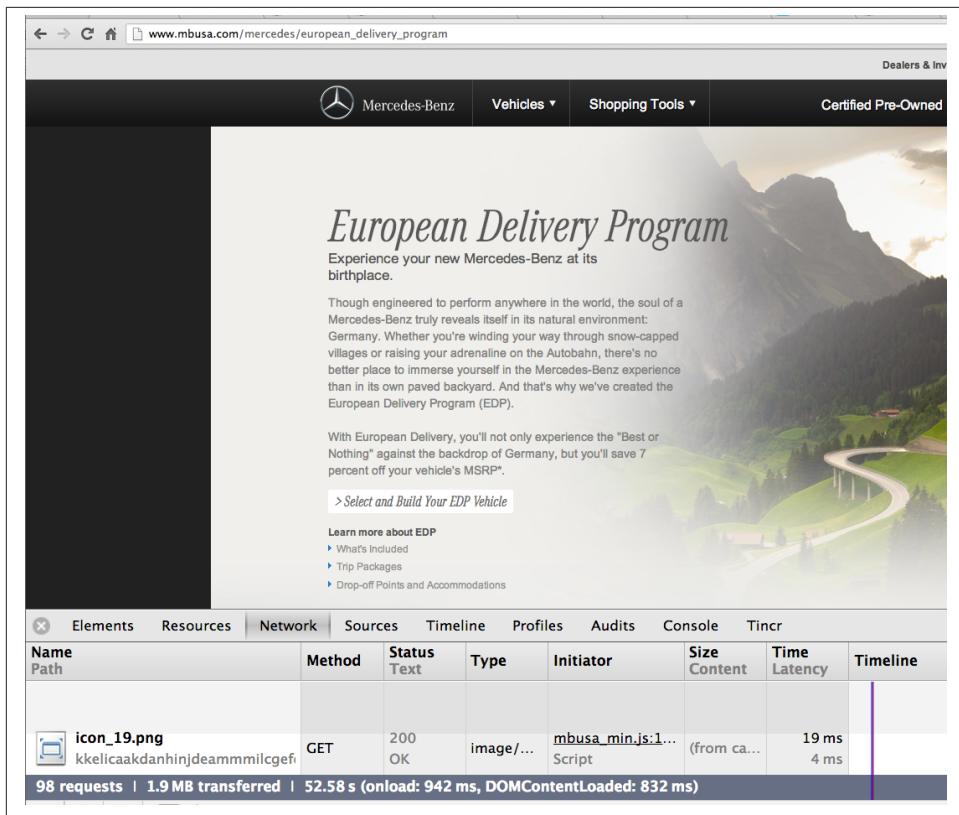


Figure 6-1. MB USA: European Delivery

As you see, 1.9Mb worth of code and other resources have been downloaded as a result of this click. Should the application architects of the MB USA decided to bring this code to the user's device on the initial load of <http://mbusa.com>, the wait time would increase by another second or more. This is unnecessary because only a tiny number of American drivers would be interested in exploring the European Delivery option. This example illustrates the use case where modularization and lazy loading is needed.

Our Save The Child application is not as big as the one by Mercedes Benz. But we'll use it to give you an example of how to build modularized Web applications that won't bring the large and monolithic code to the client's machine, but will load the code on as needed basis. We'll also give an example of how to organize the data exchange between different programming modules in a loosely coupled fashion.

Users consider a Web application fast for one of two reasons: either it's actually fast or it gives an impression of being fast. Ideally, you should do your best to create a Web application that's very responsive.



No matter how slow your Web application is, it should never feel like it's being frozen.

This chapter is about modularization techniques that will allow quick rendering of the first page of your Web application by the user's browser while loading the rest of the application in the background or on demand. We will continue refactoring the Save The Child application to illustrate using modules.

In this chapter we're going to discuss the following frameworks for modularization of JavaScript projects:

- [Browserify](#)
- [RequireJS](#) and [RequireJS Optimizer - r.js](#)
- [ECMAScript 6 \(ES6\) Module Transpiler](#)

## Modularization basics

Modules are code fragments that implement certain functionality and are written using a specific techniques. There is no out-of-the box modularization scheme in JavaScript language. The upcoming ECMAScript 6 specification tends to resolve this by introducing the *module concept* in the JavaScript language itself. This is the future.

You may ask, "Aren't .js files modules already?" Of course, you can include and load each JavaScript file using the `<script>` tag. But this approach is error prone and slow. Developers have to manually manage dependencies and file loading order. Each `<script>` tag results in the additional HTTP call to the server. Moreover, the browser blocks rendering until it loads and executes JavaScript files.

As the application gets larger the number of script files grows accordingly, which is illustrated in the following [code sample](#).

*Example 6-1. Multiple `<script>` tags complicate controlling application dependencies*

```
<!DOCTYPE html>

<html lang="en">
<head>
  <meta charset="utf-8">

  <title>Save The Child | Home Page</title>
  <link rel="stylesheet" href="assets/css/styles.css">
</head>
```

```

<body>
<!-- page body -->

<!-- body content is omitted -->

<script src="components/jquery.js"></script>      ①
<script type="text/javascript" src="app/modules/utils/load-html-content.js"></script>
<script type="text/javascript" src="app/modules/utils/show-hide-div.js"></script>
<script type="text/javascript" src="app/modules/svg-pie-chart.js"></script>
<script type="text/javascript" src="app/modules/donation.js"></script>
<script type="text/javascript" src="app/modules/login.js"></script>
<script type="text/javascript" src="app/modules/utils/new-content-loader.js"></script>
<script type="text/javascript" src="app/modules/generic-module.js"></script>
<script type="text/javascript" src="app/modules/module1.js"></script> ②
<script type="text/javascript" src="app/modules/module2.js"></script>
<script type="text/javascript" src="app/config.js"></script>
<script type="text/javascript" src="app/main.js"></script> ③
</body> ④
</html>

```

- ① Loading the jQuery first because all other modules depend on it.
- ② Other application components may also have internal dependencies on other scripts. Those scripts need to be loaded before the respective components. Having the proper order of these script tags is very important.
- ③ The script for the main Web page should be loaded after all dependencies have finished loading.
- ④ We're putting script elements at the end of the document, as it blocks as little content as possible.

As you can see, we need a better way to modularize applications than simply adding `<script>` tags. As our first step, we can use the Module design pattern and leverage the so-called “Immediately Invoked Function Expressions”.

Next, we’ll introduce and compare two popular JavaScript solutions and modularization patterns - CommonJS and Asynchronous Module Definition (AMD), which are alternative approaches to modularization. Both CommonJS and AMD are specifications defining sets of APIs.

You’ll learn pros and cons of both formats later in the chapter, but the AMD module format plays nicely with the asynchronous nature of Web. You’ll also see the use of the AMD module format and [RequireJS](#) framework to implement the modularized version of the Save The Child application.

Also, you’ll see how to use the RequireJS APIs to implement on-demand (“lazy”) loading of the application components (e.g. What We Do, Ways To Give et al.)

The upcoming ECMAScript 6 specification suggests how to handle modules, and how to start using ES6 module syntax today with the help of third-party tools like [transpiler](#). The ES6 module syntax can be compiled down to existing module solutions like CommonJS or AMD. You can find more details about CommonJS, AMD and the ES6 module format in the corresponding sections of this chapter.

After application modules are asynchronously loaded, they need to communicate to each other. You can explicitly specify the components dependencies, which is fine as long as you have a handful of components. A more generic approach is to handle inter-module communications in a loosely coupled fashion using the *Mediator* pattern, CommonJS or AMD formats. By *loosely coupled* we mean that components are not aware of each other's existence.

The next section reviews various approaches and patterns of modular JavaScript applications.

## Roads To Modularization

Although JavaScript has no built-in language support of modules, the developers' community has managed to find a way for modularization using existing syntax constructs, libraries and conventions to emulate modules-like behavior. In this section, we'll explore a few options for modularizing your application.

- The Module Pattern
- CommonJS
- Asynchronous Module Definition (AMD)

Of these three, the *Module pattern* doesn't require any additional frameworks and works in any JavaScript environment. The *CommonJS* module format is widely adopted for the server-side JavaSctipt, while the *AMD* format is popular in the applications running in Web browsers.

## The Module Pattern

In software engineering, the Module pattern was originally defined as a way to implement encapsulation of reusable code. In JavaScript, the Module pattern is used to emulate the concept of classes. We're able to include both public and private methods as well as variables inside the single object, thus hiding the encapsulated code from other global scope objects. Such encapsulation lowers the likelihood of conflicting function names defined in different scripts that could be used in the same application's scope.

Ultimately, it's just some code in an immediate-invoked function expression (IIFE) that creates a module object in the internal scope of a function and exposes this module to

the global scope using the JavaScript language syntax. Consider the following three code samples illustrating how the Module pattern could be implemented using IIFEs.

*Example 6-2. Creating a closure that hides implementation of login module*

```
var loginModule = (function() {      // ①
    "use strict";

    var module = {};
    var privateVariable = 42;

    var privateLogin = function(userNameValue, userPasswordValue) {      // ②
        if (userNameValue === "admin" && userPasswordValue === "secret") {
            return privateVariable;
        }
    };

    module.myConstant = 1984;
    module.login = function(userNameValue, userPasswordValue) {
        privateLogin(userNameValue, userPasswordValue);
        console.log("login implementation omitted");
    };

    module.logout = function() {
        console.log("logout implementation omitted");
    };

    return module;
})();
```

- ① Assigning the module object that was created in the closure to the variable `loginModule`.
- ② Because of the JavaScript's function scoping, other parts of the code can't access the code inside the closure. With this approach you can implement encapsulation and private members.

*Example 6-3. Injecting the module into the global object*

```
(function(global) {      //①
    "use strict";
    var module = {};

    var privateVariable = 42;
    var privateLogin = function(userNameValue, userPasswordValue) {
        if (userNameValue === "admin" && userPasswordValue === "secret") {
            return privateVariable;
        }
    };

    module.myConstant = 1984;
    module.login = function(userNameValue, userPasswordValue) {
```

```

    privateLogin(userNameValue, userPasswordValue);
    console.log("rest of login implementation is omitted");
};

module.logout = function() {
    console.log("logout implementation omitted");
};

global.loginModule = module;
})(this); // ②

```

- ❶ Instead of exporting the module to a variable as in the previous example, we're passing the global object as a parameter inside the closure.
- ❷ Attaching the newly created object to the global object. After that the `loginModule` object can be accessed from the external application code as `window.loginModule` or just `loginModule`.

*Example 6-4. Introducing namespaces in the global object*

```

(function(global) {
    "use strict";

    var ssc = global(ssc); // ❶
    if (!ssc) {
        ssc = {};
        global(ssc) = ssc;
    }

    var module = ssc.loginModule = {}; // ❷

    module.myConstant = 1984;
    module.login = function(userNameValue, userPasswordValue) {
        console.log("login implementation for " + userNameValue + "and" + userPasswordValue + "omitted");
    };

    module.logout = function() {
        console.log("logout implementation omitted");
    };
})(this);

```

- ❶ Here we have modification of the approach described in previous snippet. To avoid name conflicts, create a namespace for our application called `ssc`. Note that we check for this object existence in the next line.
- ❷ Now we can logically structure application code using namespaces. The global `ssc` object will contain only the code related to the Save The Child application.

The Module patterns works well for implementing encapsulation in a rather small applications. It's easy to implement and is framework-agnostic. However this approach

doesn't scale well because when working with an application with the large number of modules you may find yourself adding lots of boilerplate code checking objects' existence in the global scope for each new module. Also, you need to be careful with managing namespaces: since you are the one who put an object into the global scope, you need to think how to avoid accidental names conflicts.

The Module pattern has a serious drawback - you still need to deal with manual dependency management and manually arrange `<script>` tags in the HTML document.

## CommonJS

**CommonJS** is an effort to standardize JavaScript APIs. People who work on CommonJS APIs have attempted to develop standards for various JavaScript API (similar to standard libraries in Java, Python and etc) including standards for modules and packages. The CommonJS module proposal specifies a simple API for declaring modules, but mainly on the server-side. The CommonJS module format was optimized for non-browser environments since the early days of the server-side JavaScript.

On the Web browser side, you always need to consider potentially slow HTTP communications, which is not the case on the server. One of the solutions suitable for browsers is to concatenate all scripts into a handful of bundles to decrease the number of HTTP calls, which was not a concern for the server-side JavaScript engines because file access is nearly instantaneous. On the server side separation of the code allowed to dedicate each file to exactly one module for ease development, testing and maintainability.

In brief, the CommonJS specification requires the environment to have **three free variables**: `require`, `exports`, and `module`. The syntax to define the module is called *authoring format*. To make the module loadable by a Web browser it has to be transformed into *transport format*.

```
"use strict";
var loginModule = {};
var privateVariable = 42;

var ldapLogin = require("login/ldap");           // ❶
var otherImportantDep = require("modules/util/strings"); // ❷

var privateLogin = function(userNameValue, userPasswordValue) {
    if (userNameValue === "admin" && userPasswordValue === "secret") {
        ldapLogin.login(userNameValue, userPasswordValue);
        return privateVariable;
    }
};

loginModule.myConstant = 1984;
loginModule.login = function(userNameValue, userPasswordValue) {
    privateLogin(userNameValue, userPasswordValue);
```

```

        console.log("login implementation omitted");
    };

    loginModule.logout = function() {
        console.log("logout implementation omitted");
    };

    exports.login = loginModule;           // ❸
    // or
    module.exports = loginModule;         // ❹

    loginModule.printMetadata = function(){
        console.log(module.id);          // ❺
        console.log(module.uri);
    };

```

- ❶ ❷ If a module requires other modules, declare references to those modules inside the current module's scope by using the `require` function. You need to call `require(id)` for each module it depends on. The module id has slashes defining the file path or a URL to indicate namespaces for external modules. Modules are grouped into a packages.
- ❸ The `exports` object exposes the public API of a module. All objects, functions, constructors that your module exposes must be declared as properties of the `exports` object. The rest of the module's code won't be exposed.
- ❹ ❺ The `module` variable provides the metadata about the module. It holds such properties as `id` and a unique `uri` of each module. The `module.export` exposes `exports` object as its property. Because objects in JavaScript are passed as references, the `exports` and `module.exports` point at the same object.



The snippet above may give you an impression that the module's code is executed in the global scope, but it's not. Each module is executed in its own scope which helps to isolate them. This works automatically when you write modules for NodeJS environment running on the server. But to use CommonJS module format in the Web browser you need to use an extra tool to generate transport format from *authoring format*. **Browserify** takes all your scripts and concatenates them into one large file. Besides the module's code the generated transport bundle will contain the boilerplate code that provides CommonJS modules runtime support in the browser environment. This build step complicates the development workflow. Usually developers perform the *code/save/refresh browser* routine, but it doesn't work in this case and requires an extra steps as you need to install the additional build tool and write build scripts.

### Pros to using CommonJS:

- It's a simple API for writing and using modules.
- Such a pattern of organizing modules is widespread in the server-side JavaScript, e.g. NodeJS.

### Cons to using CommonJS:

- Web browsers don't automatically create the scoped variables `require`, `exports`, `module` hence the additional build step is required.
- The `require` method is synchronous, but there is no exact indication if dependent module's values are fully loaded because of the asynchronous nature of Web browsers. There is no event to notify the application that 100% of the required resources is loaded.
- CommonJS API is suitable for loading `.js` files, but it can't load other assets like CSS and HTML.



If you want to write modules in the format that can be used in both browser and server's environments read our suggestions in the “[Universal Module Definition](#)” on page 231 section on this chapter.

Further reading:

- [CommonJS Modules 1.1 specification](#)
- [Node.js Modules Reference](#)
- [Browserify](#)

## Asynchronous Module Definition

The AMD module format itself is a proposal for defining modules where both the module and dependencies can be asynchronously loaded. The AMD API is based on [this specification](#).

AMD began as a draft specification for module format in CommonJS, but since the full agreement about its content was not reached, the further work on module's format moved to the [amdjs Github page](#).

The AMD API have the following main functions:

- `define` for facilitating module definition. This function takes tree arguments:

- The optional module id
- An optional array of modules' IDs of dependencies
- A callback function (a.k.a *factory function*), which will be invoked when dependencies are loaded.

*Example 6-5. The signature of a define function*

```
define(
  module_id,           // ❶
  [dependencies],     // ❷
  function () {
});
```

- ❶ This string literal defines `module_id` that will be used by the AMD loader for loading this module.
- ❷ An optional array of dependencies' ids.

The factory `function{}` above will only be executed once.

For example, the Save The Child application has a menu Way To Give, which in turn depends on other module called `otherContent`. If the user clicks on this menu, we can load the module that can be defined in the `wayToGive.js` as follows:

*Example 6-6. The definition of the wayToGive module*

```
define(["otherContent"], function(otherContent) { // ❶
  var wayToGive;

  console.log("otherContent module is loaded");
  wayToGive = function() {
    return {
      render: function() {
        var dataUrl, newContainerID, whatWeDoButton;

        whatWeDoButton = "way-to-give";
        newContainerID = "way-to-give-container";
        dataUrl = "assets/html-includes/way-to-give.html";
        otherContent.getNewContent(whatWeDoButton, newContainerID, dataUrl); // ❷
        return console.log("way-to-give module is rendered");
      },
      init: function() {
        return console.log("way-to-give init");
      }
    };
  };
  return wayToGive; // ❸
});
```

- ❶ This code doesn't have the optional `module_id`. The loader will use the file name without the `.js` extension as `module_id`. Our module has one dependency on the module called `otherContent`. The dependent module instance will be passed in the factory method as variable `otherContent`.
- ❷ We can start using the dependency object immediately. AMD loader have taken care of loading and instantiation of this dependency.
- ❸ The module returns constructor function to be used for creation of new objects.
  - The `require` function takes two arguments
    - An array of module IDs to load. Module ID is a string literal.
    - A callback to be executed once those modules are available. The modules loaded by IDs are passed into the callback in order. [Here is example](#) of the `require` function usage.

*Example 6-7. The example of `require` function usage*

```
require(["main"], function() {
  console.log("module main is loaded");
});
```

### Pros to using AMD:

- It's a very simple API that has only two functions - `require` and `define`.
- A wide variety of loaders is available. You'll find more coverage on loaders in the [RequireJS section](#).
- The CommonJS module authoring format is supported by the majority of loaders. You'll see an example of modules later in the [Example 6-19](#) section.
- Plugins offer an immense amount of flexibility.
- AMD is easy to debug.

Consider the following error messages that JavaScript interpreter may throw:

*There was an error in /modules/loginModule.js on line 42*

**vs**

*There was an error in /built-app.js on line 1984*

In modularized applications you can easier localize errors.

- Performance: module are loaded only when required hence the initial portion of the application's code become smaller.

### Cons to using AMD:

- The dependency array can get rather large for complex modules.

```
define(
  ["alpha", "beta", "gamma", "delta", "epsilon", "omega"], // ❶
  function(alpha, beta, gamma, delta, epsilon, omega){
    "use strict";
    // module's code omitted
});

```

- ❶ In the real-world enterprise applications the array of dependency modules might be pretty large.

- Human errors can result in mismatch between dependency array and callback arguments.

```
define(
  ["alpha", "beta", "gamma", "delta", "epsilon", "omega"], // ❶
  function(alpha, beta, gamma, delta, omega, epsilon){
    "use strict";
    // module's code omitted
});

```

- ❶ The mismatch of module IDs and factory function arguments will cause module usage problems.

## Universal Module Definition

Universal Module Definition (UMD) is a series of patterns and code snippets that provide compatibility boilerplate to make modules environment-independent. Those patterns can be used to support multiple module formats. UMD is not a specification or a standard. You need to pay attention to UMD patterns in case when your modules will run in more than one type of environment (e.g. a Web browser and on the server side engine running NodeJS). In most cases, it makes a lot of sense to use a single module format.

Here is an example of the module definition in UMD notation. In the following example, the module can be used with the AMD loader and as one of the variations of the Module Pattern.

```
(function(root, factory) {
  "use strict";
  if (typeof define === "function" && define.amd) { // ❶
    define(["login"], factory);
  } else {
    root.ssc = factory(root.login); // ❷
  }
}(this, function(login) { // ❸
  "use strict";
})
```

```

        return {
            myConstant: 1984,
            login: function(userNameValue, userPasswordValue) {
                console.log("login for " + userNameValue + " and " + userPasswordValue);
            },
            logout: function() {
                console.log("logout implementation omitted");
            }
        };
    });
});

```

- ① If the AMD loader is available proceed with defining the module according to AMD specification.
- ② If the AMD loader isn't present, use the factory method to instantiate the object and attach it to the `window` object.
- ③ Passing the top-level context and providing a implementation of a factory function.

You can find more information about UMD and commented code snippets for different situations in [the UMD project repository](#).

## ECMAScript 6 Modules

The ECMAScript 6 (ES6) specification is an evolving draft outlining changes and features for the next version of the JavaScript language. This specification is not finalized yet and the browsers' support for anything defined in ES6 will be experimental at best and cannot be relied upon for Web applications that must be deployed in production mode in multiple browsers.

One of the most important features of ES6 specification is the *module syntax*. Here is an example of some login module definition:

*Example 6-8. Login module definition*

```

export function login(userNameValue, userPasswordValue) {
    return userNameValue + "_" + userPasswordValue;
}

```

The keyword `export` specifies the function or object (a separate file) to be exposed as a module, which can be used from any other JavaScript code as follows:

*Example 6-9. Main application module*

```

import {login} from './login'
var result = login("admin", "password");

```

With the `import` keyword we assign instance of `login()` function imported from `login` module.

## ES6 Module Transpiler

Although ES6 standard is not implemented yet by most of browsers you can use the third party tools to get a taste of upcoming enhancements in JavaScript language. [ES6 Module Transpiler](#) library developed by the [Square](#) Engineers helps using the module authoring syntax from ES6 and compile it down to the transport formats that you learned earlier in this chapter.

Consider the following module `circle.js`:

*Example 6-10. A `circle.js` module*

```
function area(radius) {
    return Math.PI * radius * radius;
}

function circumference(radius) {
    return 2 * Math.PI * radius;
}

export {area, circumference};
```

This module exports two functions: `area()` and `circumference()`.

*Example 6-11. The main application's script `main.js` can use these functions*

```
import { area, circumference } from './circle'; // ①

console.log("Area of the circle: " + area(2) + " meter squared"); // ②
console.log("Circumference of the circle: " + circumference(5) + " meters");
```

- ❶ The `import` keyword specifies the objects we want to use from the module.
- ❷ A sample use of the imported functions

The ES6 Module Transpiler's command `compile-module` can compile the module to be compliant with CommonJS, AMD, or the code that implements the Module pattern. With the `type` command line option you can specify that output format will be: `amd`, `cjs` and `globals`.

```
compile-modules circle.js --type cjs --to ../js/
compile-modules main.js --type cjs --to ../js/
```

- **CommonJS** format

*Example 6-12. `circle.js`*

```
"use strict";
function area(radius) {
    return Math.PI * radius * radius;
}
```

```

function circumference(radius) {
    return 2 * Math.PI * radius;
}

exports.area = area;
exports.circumference = circumference;

```

+

*Example 6-13. main.js*

```

"use strict";
var area = require("./circle").area;
var circumference = require("./circle").circumference;

console.log("Area of the circle: " + area(2) + " meter squared");
console.log("Circumference of the circle: " + circumference(5) + " meters");

```

Should we compiled the modules into the AMD format using the option *amd*, we would have received a different output in the AMD format.

- AMD format

*Example 6-14. circle.js*

```

define("circle",
  ["exports"],
  function(__exports__) {
    "use strict";
    function area(radius) {
      return Math.PI * radius * radius;
    }

    function circumference(radius) {
      return 2 * Math.PI * radius;
    }

    __exports__.area = area;
    __exports__.circumference = circumference;
  });

```

+

*Example 6-15. main.js*

```

define("main",
  ["./circle"],
  function(__dependency1__) {
    "use strict";
    var area = __dependency1__.area;
    var circumference = __dependency1__.circumference;

```

```
        console.log("Area of the circle: " + area(2) + " meter squared");
        console.log("Circumference of the circle: " + circumference(5) + " meters");
    });
}
```

Using the *globals* option in the *compile-modules* command line produces the code that can be used as described in the Module Pattern section earlier in the chapter.

- Browser globals

*Example 6-16. circle.js*

```
(function(exports) {
    "use strict";

    function area(radius) {
        return Math.PI * radius * radius;
    }

    function circumference(radius) {
        return 2 * Math.PI * radius;
    }
    exports.circle.area = area;
    exports.circle.circumference = circumference;
})(window);
```

+

*Example 6-17. main.js*

```
(function(circle) {
    "use strict";
    var area = circle.area;
    var circumference = circle.circumference;

    console.log("Area of the circle: " + area(2) + " meter squared");
    console.log("Circumference of the circle: " + circumference(5) + " meters");

})(window.circle);
```

To learn the up-to-date information on the ES6 browsers' support visit [ECMAScript 6 compatibility table](#).



TypeScript is an open-source language from Microsoft that compiles to JavaScript and brings object-oriented concepts like classes and modules to JavaScript. It has a module syntax, which is very similar to what ES6 standard proposes. The TypeScript compiler can produce CommonJS and AMD module formats. You can learn more about TypeScript from [language specification](#).

## Dicing the Save The Child Application Into Modules

Now that you know the basics of AMD and different modularization patterns, let's see how you can dice our Save The Child application into smaller pieces. In this section we'll apply the AMD-compliant module loader from the framework [RequireJS](#).



[curl.js](#) offers another AMD-compliant asynchronous resource loader. Both curl.js and RequireJS have similar functionality, and to learn how they differ follow [this thread on RequireJS group](#).

Let's start with a brief explanation of the directory structure of the modularized Save The Child application.

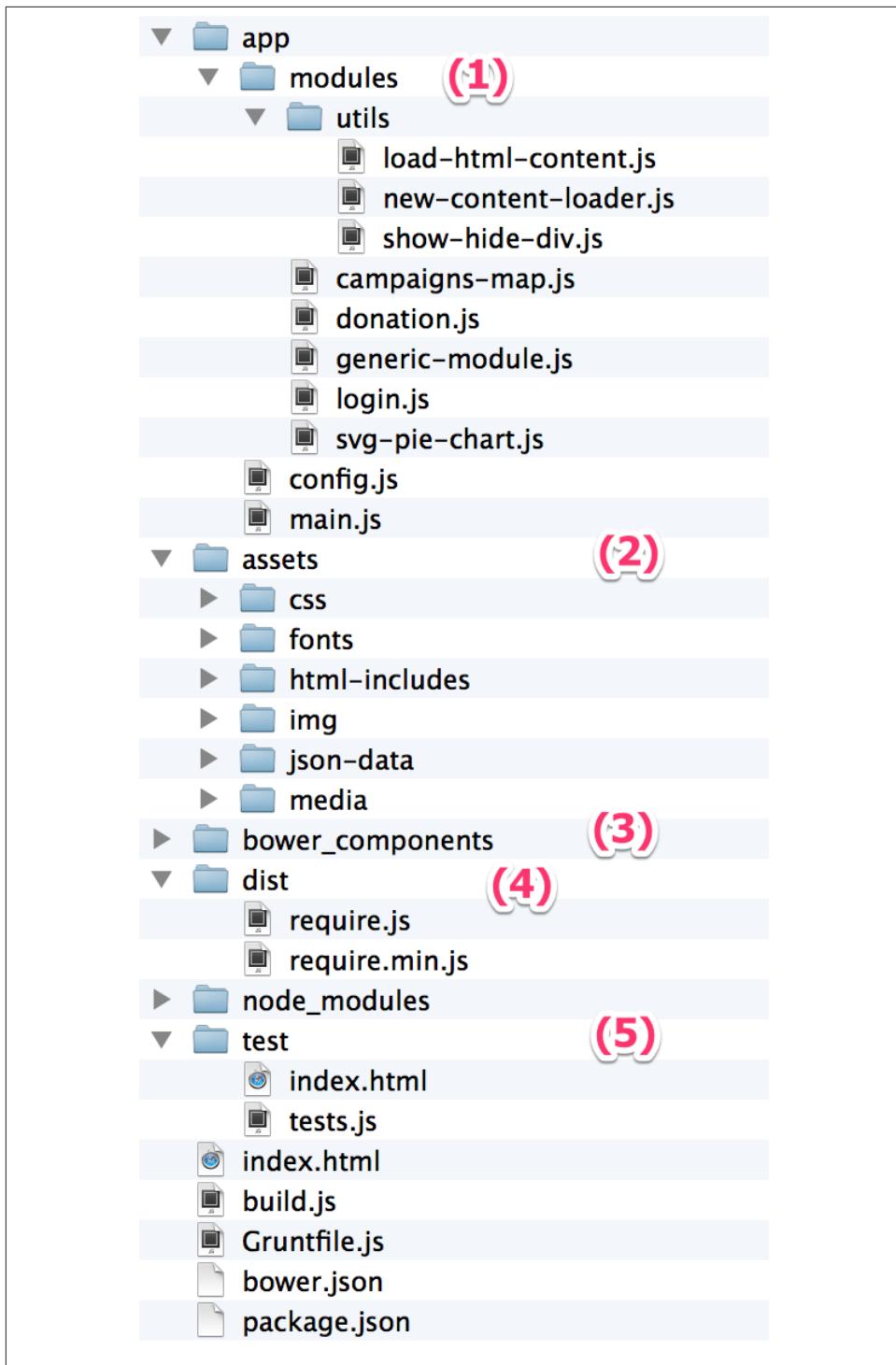
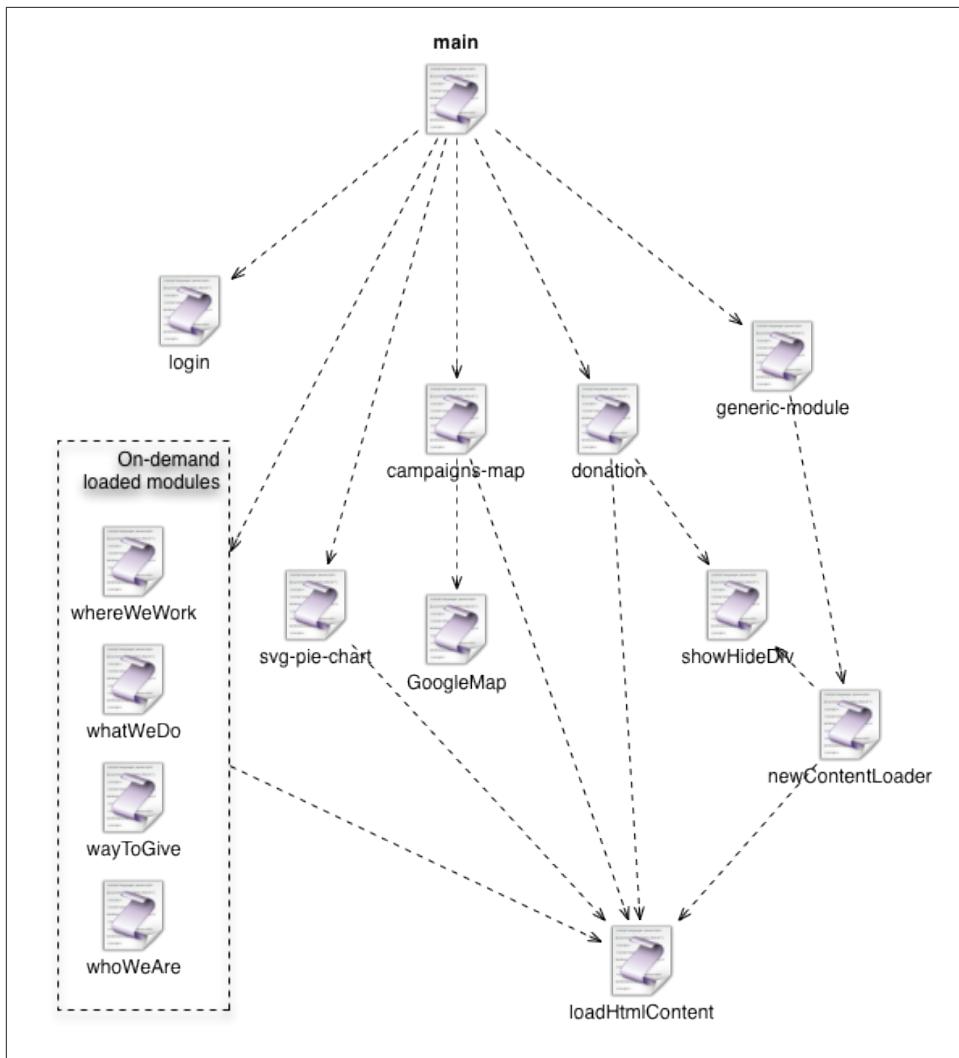


Figure 6-2. A directory structure of Save The Child Application Into Modules | 237

1. All application's JavaScript files reside in the *app/modules* directory.  
Inside the *modules* directory you can have as many nested folders as you want, e.g. *utils* folder.
2. The application assets remain the same as in previous chapters.
3. We keep all Bower-managed dependencies in the *bower\_components* directory such as RequireJS, jQuery etc.
4. The *dist* directory serves as the location for the optimized version of our application. We will cover optimization with *r.js* later in the "[Using RequireJS Optimizer](#)" on [page 245](#) section.
5. The QUnit/Jasmine tests will live in the *test* directory. Testing will be covered in Chapter 8.

We are not going to dice the Save The Child application into multiple modules, but will just show you how to start this process. [Figure 6-3](#) illustrates the modules' dependencies. For example, the `main` module depends on `login`, `svg-pie-chart`, `campaigns-map`, `donation`, and `generic`. There is also a group of modules that will be loaded on demand: `whereWeWork`, `whatWeDo`, `wayToGive`, `whoWeAre`.



*Figure 6-3. The modules graph of Save The Child*

To dice the application into modules you'll need the the modularization framework RequireJS, which can either be downloaded from its github repository or you can install it using a package manager Bower that was explained in previous chapter on automation tools.

Once RequireJS downloaded and placed into the project directory, add it to the index.html file as demonstrated in [Example 6-18](#) snippet.

*Example 6-18. Adding RequireJS to the web page*

```
<!DOCTYPE html>
<head>
    <!-- content omitted -->
</head>
<body>
    <!-- page body -->

<script src="bower_components/requirejs/require.js"
    data-main="app/config"></script> ①

</body>
</html>
```

- ① Once the RequireJS library is loaded it will look for the `data-main` attribute and attempt to load the `app/config.js` script asynchronously. The `app/config.js` will become the entry point of our application.

## Inside RequireJS configuration: config.js

RequiredJS uses a configuration object that includes modules and dependencies that have to be managed by the framework.

```
require.config({           // ①
    paths: {               // ②
        'login': 'modules/login',
        'donation': 'modules/donation',
        'svg-pie-chart': 'modules/svg-pie-chart',
        'campaigns-map': 'modules/campaigns-map',
        'showHideDiv': 'modules/utils/show-hide-div',
        'loadHtmlContent': 'modules/utils/load-html-content',
        'newContentLoader': 'modules/utils/new-content-loader',
        'bower_components': '../bower_components',
        'jquery': '../bower_components/jquery/jquery',
        'main': 'main',
        'GoogleMap': '../bower_components/requirejs-google-maps/dist/GoogleMap',
        'async': '../bower_components/requirejs-plugins/src/async'
    }
});

require(['main'], function () {      // ③
});
```

- ① The [RequireJS documentation](#) has a comprehensive overview of all configuration options. We've included some of them here.

- ❷ The `paths` configuration option defines the mapping for module names and their paths. The `paths` is used for module names and shouldn't contain file extensions.
- ❸ After configuring the modules' paths we're loading the `main` module. The navigation of our application flow starts there.

## Writing AMD Modules

Let's take a closer look at the module's internals that make it consumable by the RequireJS module loader.

```
define(["newContentLoader"], function(contentLoader) {           // ❶
  "use strict";
  var genericModule;

  genericModule = function(moduleId) {    // ❷
    return {
      render: function(button, containerId, dataUrl) {
        contentLoader.getNewContent(button, containerId, dataUrl); // ❸
        console.log("Module " + moduleId + " is rendered...");
      }
    };
  };
  return genericModule;
});
```

- ❶ As we discussed in the “[Asynchronous Module Definition](#)” on page 228 section, the code that you want to expose as a module should be wrapped in the `define()` function call. The first parameter is an array of dependencies. The location of dependencies files is defined [inside the config file](#). The dependency object doesn't have the same name as the dependency string id. The order of arguments in the factory function should be the same as the order in the dependencies array.
- ❷ In this module we export only in the constructor function that returns the object that uses the `render` function to draw the visual component on the screen.
- ❸ The `contentLoader` object loaded from `app/modules/util/new-content-loader.js` (see the `paths` property [in RequireJS config object](#)), is instantiated by RequireJS and is ready to use.

RequireJS also supports the CommonJS module format with a slightly different signature of the `define()` function. This helps to bridge the gap between AMD and CommonJS. If your factory function accepts parameters but no dependency array, the AMD environment assumes that you wish to emulate the CommonJS module environment. The standard `require`, `exports`, and `module` variables will be injected as parameters into the factory.

Here is an example of CommonJS module format with RequireJS.

*Example 6-19. Using CommonJS module format in RequireJS*

```
define(function(require, exports, module) { // ①
    "use strict";
    module.exports = (function() {           // ②
        var dependency = require("dependencyId"); // ③

        function AuctionDTO(_arg) {
            this.auctionState = _arg.auctionState;
            this.item = _arg.item;
            this.bestBid = _arg.bestBid;
            this.auctionId = _arg.auctionId;
            dependency.doStuff();
        }

        AuctionDTO.prototype.toJson = function() {
            return JSON.stringify(this);
        };
    });

    return AuctionDTO;
})();
});
```

- ❶ The factory receives up to three arguments that emulate the CommonJS `require`, `exports`, and `module` variables.
- ❷ Export your module rather than returning it. You can export an object in two ways - assign the the module directly to `module.exports` as shown in this snippet, or set the properties on the `exports` object.
- ❸ In CommonJS, dependencies are assigned to local variables using the `require(id)` function.

## Loading Modules On-Demand

As per the Save The Child modules graph, some components shouldn't load when the application starts. Similar to Mercedes Benz website example, some functionality of Save The Child can be loaded later when user needs it. The user might never want to visit the “Where we work” section. Hence this functionality is a good candidate for the *load on-demand* module. You may want to load such a module on demand when the user clicks the button or selects a menu item.

At any given time a module can be in one of three states:

- not loaded (`module === null`)
- loading is in progress (`module === 'loading'`)

- fully loaded (`module !== null`).

*Example 6-20. Loading the module on demand*

```
var module;
var buttonClickHandler = function(event) {
    "use strict";
    if (module === "loading") {           // ①
        return;
    }
    if (module !== null) {               // ②
        module.render();
    } else {
        module = "loading";             // ③
        require(["modules/wereWeWork"], function(ModuleObject) { // ④
            module = new ModuleObject();
            module.render();
        });
    }
};
```

- ① Checking if module loading in progress.
- ② Don't re-load the same module. If the module was already loaded just call the method to render the widget on the Web page.
- ③ Setting the module into the intermediate state until it's fully loaded.
- ④ Once the `wereWeWork` module is loaded, the callback will receive the reference to this module - instantiate `wereWeWork` and render it on the page.

Let's apply the technique demonstrated in [Example 6-20](#) snippet for Save The Child application to lazy load the “Who We Are”, “What We Do”, “Where We Work” and “What To Give” modules only if the user clicked on the corresponding top bar link.

*Example 6-21. The Main Module*

```
define(['login',
    'donation',
    'campaigns-map',
    'svg-pie-chart',
    'modules/generic-module' // ①
], function() {
    var initComponent, onDemandLoadingClickHandlerFactory;
    onDemandLoadingClickHandlerFactory = function(config) {
        return function(event) { // ②
            if (config.amdInstance === 'loading') {
                return;
            }
            if (config.amdInstance != null) {
                config.amdInstance.render(event.target.id, config.containerId, config.viewUrl);
            } else {
```

```

        config.amdInstance = 'loading';
        require(['modules/generic-module'], function(GenericModule) {
            var moduleInstance;
            moduleInstance = new GenericModule(config.moduleId);
            moduleInstance.render(event.target.id, config.containerId, config.viewUrl);
            config.amdInstance = moduleInstance;
        });
    }
};

initComponent = function(config) {
    config.button.addEventListener('click',
        onDemandLoadingClickHandlerFactory(config), // ❸
        false);
};

return (function() {
    var componentConfig,
        componentConfigArray,
        way_to_give, what_we_do,
        where_we_work,
        who_we_are, _i, _len;
    way_to_give = document.getElementById('way-to-give');
    what_we_do = document.getElementById('what-we-do');
    who_we_are = document.getElementById('who-we-are');
    where_we_work = document.getElementById('where-we-work');
    componentConfigArray = [{ // ❹
        moduleId: 'whoWeAre',
        button: who_we_are,
        containerId: 'who-we-are-container',
        viewUrl: 'assets/html-includes/who-we-are.html'
    }, {
        moduleId: 'whatWeDo',
        button: what_we_do,
        containerId: 'what-we-do-container',
        viewUrl: 'assets/html-includes/what-we-do.html'
    }, {
        moduleId: 'whereWeWork',
        button: where_we_work,
        containerId: 'where-we-work-container',
        viewUrl: 'assets/html-includes/where-we-work.html'
    }, {
        moduleId: 'wayToGive',
        button: way_to_give,
        containerId: 'way-to-give-container',
        viewUrl: 'assets/html-includes/way-to-give.html'
    }];
    for (_i = 0, _len = componentConfigArray.length; _i < _len; _i++) {
        componentConfig = componentConfigArray[_i];
        initComponent(componentConfig); // ❺
    }
    console.log('app is loaded');
});

```

```
});  
});
```

- ❶ The first argument of the `define` function is an array of dependencies.
- ❷ Here we're using the approach described in the [Example 6-20](#) section. This factory function produces the handler for the button click event. It uses RequireJS API to load the module once the user clicked on the button.
- ❸ Instantiate the click handler function using `onDemandLoadingClickHandlerFactory` and assign it to the button defined in the module config.
- ❹ An array of modules that can be loaded on demand.
- ❺ In the last step, we need to initialize each module button with the lazy loading handler.

## RequireJS plugins

RequireJS plugins are special modules that implement specific API. For example, the `text` plugin allows to specify a text file as a dependency, `cs!` translates *CoffeeScript* files into JavaScript. The plugin's module name comes before the `!` separator. Plugins can extend the default loader's functionality.

In the Save The Child application we use the `order.js` plugin that allows to specify the exact order in which the dependencies should be loaded. You can find the full list of the available RequireJS plugins at the following [wiki page](#).

## Using RequireJS Optimizer

RequireJS comes with the optimization tool called `r.js`, which is a utility that performs module optimization. Earlier in this chapter, we've specified the dependencies as an array of string literals that are passed to the top-level `require` and `define` calls. The optimizer will combine modules and their dependencies into a single file based on these dependencies.

Furthermore, `r.js` integrates with other optimization tools like [UglifyJS](#) and [Closure Compiler](#) to minify the content of script files. We are going to use the JavaScript task runner [Grunt](#) that you learned in previous chapter on automation tools.

Let's configure our Grunt project to enable the optimization task. Here's the command to install RequireJS, and Grunt's task packages `clean`, `concat` and `uglify` and save them as development dependencies in the file `package.json`:

*Example 6-22. Adding dependencies to package.json*

```
> npm install grunt-contrib-requirejs\\n  
  grunt-contrib-concat grunt-contrib-clean\\n  
  grunt-contrib-uglify --saveDev
```

The following listing describes the script to setup RequireJS optimizer and the related optimization tasks for Grunt. You'll need to run this script to generate optimized version of the Save The Child application.

Script to setup RequireJS optimizer.

```
"use strict";

module.exports = function (grunt) {

    // Project configuration.
    grunt.initConfig({
        // Task configuration.
        clean: {
            files: ["dist"] // ①
        },
        requirejs: {           // ②
            compile: {
                options: {
                    name: "config",
                    mainConfigFile: "app/config.js",
                    out: "<%= concat.dist.dest %>",
                    optimize: "none"
                }
            }
        },
        concat: {             // ③
            dist: {
                src: ["components/requirejs/require.js", "<%= concat.dist.dest %>"],
                dest: "dist/require.js"
            }
        },
        uglify: {             // ④
            dist: {
                src: "<%= concat.dist.dest %>",
                dest: "dist/require.min.js"
            }
        }
    });

    grunt.loadNpmTasks("grunt-contrib-clean");      // ⑤
    grunt.loadNpmTasks("grunt-contrib-requirejs");
    grunt.loadNpmTasks("grunt-contrib-concat");
    grunt.loadNpmTasks("grunt-contrib-uglify");

    grunt.registerTask("default", ["clean", "requirejs", "concat", "uglify"]); // ⑥
};


```

- ① The `clean` task cleans output directory. In the `files` section of task config we specify what folder should be cleaned.

- ② The `requirejs` task. The configuration properties of `requirejs` task are self-explanatory. `mainConfigFile` points at the same file that `data-main` attribute of RequireJS script tag. `out` parameter specifies output directory where optimized script will be created.
- ③ The `concat` task combines/concatenates optimized modules code and RequireJS loader code.
- ④ The `uglify` task minifies provided files (see `src` properties) using `UglifyJS` - a compressor/minifier tool. UglifyJS produces significantly smaller version of the original script by reducing the **Abstract Syntax Tree (AST)** and changing local variable names to single-letters.
- ⑤ Loading plugins that provide necessary tasks.
- ⑥ The default task to execute all tasks in order.

Run the **Save The Child** application built with RequireJS and monitor the network traffic in Chrome Developer Tools. You'll see many HTTP requests that load modules asynchronously. As you can see from the **from following screenshot**, 12 out of 14 browser's requests are for loading all required modules. The modules that may be loaded on demand are not here.

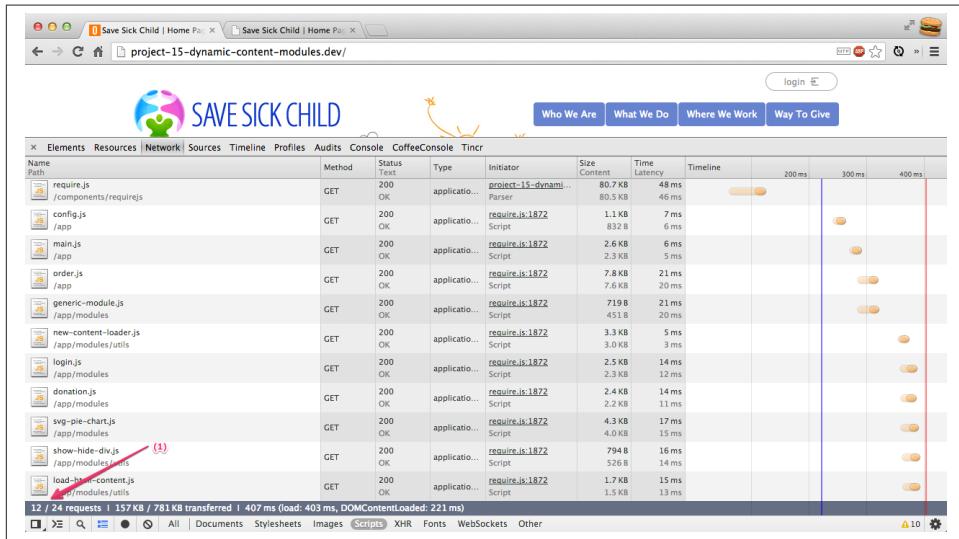


Figure 6-4. Unoptimized version of the Save The Child Application

The next **screenshot** shows loading of the Save The Child application optimized with RequireJS optimizer. We've managed to pack all our modules, their dependencies and

loader's code into a single file, which considerably decreased the number of the server-side calls.

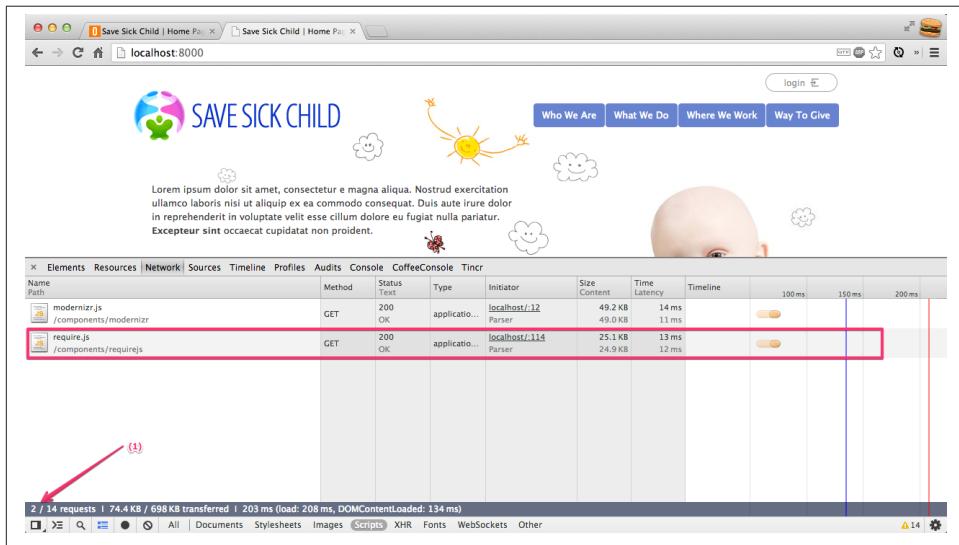


Figure 6-5. Loading of optimized version of the Save The Child



Read more on optimization topics in the RequireJS documentation site under “[Optimization](#)” section.

RequireJS took care of the optimal module loading, but you should properly arrange the inter-module communication. The Save The Child application doesn't have modules that need to exchange data so we'll describe how to properly arrange inter-module communications in a separate application.



Google has created [PageSpeed Insights](#), a Web tool that offers suggestions for improving the performance of your Web application on all devices. Just enter the URL of your application and a second later you'll see some optimization suggestions.

# Loosely-Coupled Inter-Module Communications With Mediator

Almost any complex enterprise Web application consists of a number of components and modules.

A simple approach of arranging communications between the components is to allow all these components directly accessing public properties of each other. This would produce an application with tightly coupled components that know about each other and removal of one component could lead to multiple code changes in the application.

A better approach is to create loosely coupled components that are self-contained, do not know about one another and can communicate with the “outside world” by sending and receiving events.

Creating UI of from reusable components applying messaging techniques requires creation of loosely coupled components. Say you’ve created a window for a financial trader. This window gets a data push from the server showing the latest stock prices. When the trader likes the price he may click on the Buy or Sell button to initiate a trade. The trading engine can be implemented in a separate component and establishing inter-component communications the right way is really important.

As you’ve learned from Chapter 2, *Mediator* is a behavioral design pattern that allows to unify communications of the application components. The Mediator pattern promotes the use of a single shared object that handles (mediates) communication between other objects. None of the components is aware of the others, but each of them knows about a single object - the mediator.

In Chapter 2 we’ve introduced an example of a small fragment of a trader’s desktop. Let’s reuse the same example, but this time not with `postMessage` but with the Mediator object.

In Figure [Figure 6-6](#), the Pricing Panel on the left gets the data feed about the current prices of the IBM stock. When the user clicks on Bid or Ask button, the Pricing Panel just sends the event with the relevant trade information, e.g. a JSON-formatted string containing the stock symbol, price, buy or sell flag, date, etc.

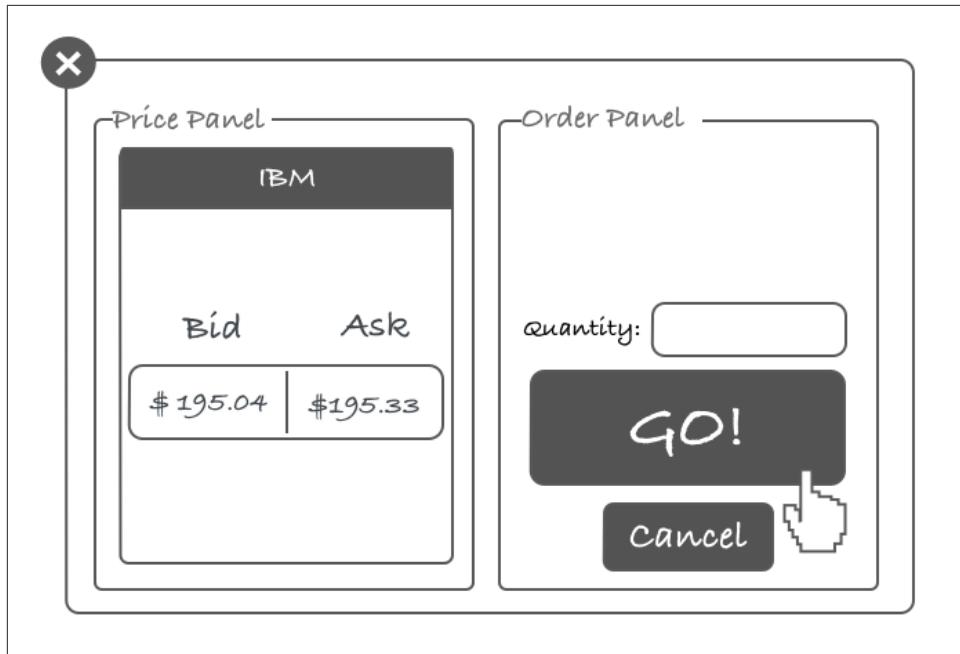


Figure 6-6. Before the trader clicked on the Price Panel



Figure 6-7. After the trader clicked on the Price Panel

Here is a HTML code snipped that implements the above scenario.

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>An example of Mediator Design Pattern</title>
    <script data-main="app/config" src="bower_components/requirejs/require.js"></script>
</head>
<body>
    <h1>mediator and RequireJS example</h1>

    <div id="pricePanel">          ①
        <p>IBM</p>
        <label for="priceInput">Bid:</label>
        <input type="text" id="priceInput" placeholder="bid price"/>

        <label for="priceInput">Ask:</label>
        <input type="text" id="priceInput" placeholder="bid price"/>
    </div>
    <div id="orderPanel">          ②
        <p id="priceText"></p>
        <label for="quantityInput">Quantity:</label>
        <input type="text" id="quantityInput" placeholder="" />
        <button id="goButton">Go!</button>
        <button id="cancelButton">cancel</button>
    </div>

```

```
</div>

</body>
</html>
```

- ❶ This div element contains *Pricing Panel* with Bid and Ask controls.
- ❷ This div element contains *Order Panel* with Quantity, Go and Cancel controls.

As we stated before, we need a Mediator to handle communication between the application components. The components need to register themselves with the Mediator so it knows about them and can route communications. The following snippet is a sample Mediator implementation (we use `define` and `require` from RequireJS here).

*Example 6-23. The implementation of Mediator pattern.*

```
define(function() {
    "use strict";
    return (function() { // ❶
        var components = {};

        function Mediator() {}

        Mediator.registerComponent = function(name, component) { // ❷
            var cmp;
            for (cmp in components) {
                if (components[cmp] === component) {
                    break;
                }
            }
            component.setMediator(Mediator); // ❸
            components[name] = component; // ❹
        };

        Mediator.broadcast = function(event, args, source) { // ❺
            var cmp;
            if (!event) {
                return;
            }
            args = args || [];
            for (cmp in components) {
                if (typeof components[cmp]["on" + event] === "function") { // ❻
                    source = source || components[cmp];
                    components[cmp]["on" + event].apply(source, args);
                }
            }
        };
        return Mediator;
    })();
});
```

- ❶ Return the private object that stores registered components.
- ❷ With the `Mediator.register()` function we can store components in the associative array. The Mediator is a singleton object here.
- ❸ Assigning the mediator instance to the component being registered.
- ❹ Registering component in the array using provided `name` as key.
- ❺ The component can invoke `Mediator.broadcast()` when it has some information to share with other application components.
- ❻ If a component has a function property with the name matching the pattern `"on" + event`, e.g. `onClickEvent`, the Mediator will invoke this function in the context of source object.

The following code sample shows the main entry point of the application that uses the above mediator.

```
define(["mediator", "pricePanel", "orderPanel"], function(Mediator, PricePanel, OrderPanel) { // ❶
  "use strict";
  return (function() {
    Mediator.registerComponent("pricePanel", new PricePanel()); // ❷
    Mediator.registerComponent("orderPanel", new OrderPanel());

    document.getElementById("priceInput").addEventListener("click", function() { // ❸
      if ( !! this.value) {
        return Mediator.broadcast("BidClick", [this.value]); // ❹
      }
    });
  })();
});
```

- ❶ Required modules will be loaded by RequireJS.
- ❷ Registering our components with the Mediator.
- ❸ Adding the `click` event listener for the Bid Price component.
- ❹ When the user clicks on the bid price the Mediator will broadcast the `Bid Click` event to all registered component. Only the component that has this specific event handler with the name matching the pattern `"on" + event` will receive this event.

Let's see how the code of the `PricePanel` and `OrderPanel` components looks like.

```
define(function() {
  "use strict";
  return (function() {
    var mediator;

    function PricePanel() {
```

```

    }

    PricePanel.prototype.setMediator = function(m) { // ①
        mediator = m;
    };

    PricePanel.prototype.getMediator = function() { // ②
        return mediator;
    };

    PricePanel.prototype.onBidClick = function(currentPrice) { // ③
        console.log("Bid clicked on price " + currentPrice);
        this.getMediator().broadcast("PlaceBid", [currentPrice]);
    };

    PricePanel.prototype.onAskClick = function() { // ④
        console.log("Ask clicked");
    };

    return PricePanel;
})();
});

```

- ① The setter of the Mediator object. Mediator injects its' instance during component registration (refer to [Example 6-23](#) snippet).
- ② The getter of the Mediator object.
- ③ The onBidClick event handler. The Mediator will call this function when the BidClick event will be broadcast. Using getter getMediator we can broadcast PlaceBid event to all registered components.
- ④ PROD: Need a callout list item to #4 in the code here.

```

define(function () {
    "use strict";
    return (function () {
        var mediator;

        function OrderPanel() {}

        OrderPanel.prototype.getMediator = function () { // ①
            return mediator;
        };

        OrderPanel.prototype.setMediator = function (m) {
            mediator = m;
        };

        OrderPanel.prototype.onPlaceBid = function (price) { // ②
            console.log("price updated to " + price);
            var priceTextElement = document.getElementById("priceText");

```

```
    priceTextElement.value = price;  
};  
  
return OrderPanel;  
});  
});  
});
```

- ❶ The Mediator's getter and setter have the purpose similar to described in previous snippet.
- ❷ Defining the PlaceBid event handler - `onPlaceBid()`.

As you noticed, both `OrderPanel` and `PricePanel` don't know about the existence of each other, but nevertheless they can send and receive data with the help of intermediary - the Mediator object.

The introduction of the Mediator increases reusability of components by decoupling them from each other. The Mediator pattern simplifies the maintenance of any application by centralizing the navigational logic.

## Summary

The size of any application tends to increase with time, and sooner or later you'll need to decide how to cut it into several loadable blocks of functionality. The sooner you start modularizing your application the better.

In this chapter we've reviewed several options available for writing modular JavaScript using modern module formats. These formats have a number of advantages over using just the classical Module Pattern. These advantages include avoiding creating global variables for each module and better support for static and dynamic dependency management.

Understanding various technologies and frameworks available in JavaScript, combined with the knowledge of different ways of linking modules and libraries is crucial for developers who want their JavaScript applications to be more responsive.



# Test-Driven Development with JavaScript

To shorten the development cycle of your Web application you need to start testing it on the early stages of the project. It seems obvious, but many enterprise IT organizations haven't adopted agile testing methodologies, which costs them dearly. JavaScript is dynamically typed interpreted language - there is no compiler to help in identifying errors as it's done in compiled languages like Java. This means that a lot more time should be allocated for testing for JavaScript Web applications. Moreover, a programmer who didn't introduce testing techniques into his daily routine can't be 100% sure that his code works properly.

The static code analysis and code quality tools such as [Esprima](#) and [JSHint](#) will help in reducing the number of syntax errors and improve quality of your code.



We've demonstrated how to setup [JSHint](#) for your JavaScript project and automate the process of checking your code for the syntax errors in a chapter «Selected Productivity Tools for Enterprise Developers».

To switch to a test-driven development mode, make testing a part of your development process in its early stages rather than scheduling testing after the development cycle is complete.

Introduction of test-driven development may substantially improve your code quality. It is very important to receive the feedback about your code on a regular basis. That's why tests must be automated and should run as soon as you've changed the code.

There are many testing frameworks in JavaScript world, but we'll give you a brief overview of two of them: [QUnit](#) and [Jasmine](#). The main goal of each framework is to test small pieces of code a.k.a. *units*.

We will go through basic testing techniques known as “Test-Driven Development” and “Test First”. You’ll learn how to automate the testing process in multiple browsers with **Testem Runner** or by running tests in so called *headless* mode with **PhantomJS**.

The second part of this chapter is dedicated to setting up a new Save The Child project in the IDE with selected test frameworks.

## Why Test ?

Any software has bugs. But in interpreted languages like JavaScript you don’t have help of compilers that could have pointed you at the potential issues on early stages of development. You need to continue testing code over and over again to catch regression errors, to be able to add new features without breaking the existing ones. Code that is covered with tests is easy to refactor. Tests help to prove correctness of your code. A well tested code leads to better overall design of your programs.

## Testing Basics

In this chapter we’ll discuss the following types of testing:

- Unit testing
- Integration testing
- Functional testing
- Load (a.k.a. stress) testing

### Quality Assurance vs. Use Acceptance Testing

Although, *Quality Assurance* (QA) and *User Acceptance Testing* (UAT) is far beyond the scope of this chapter you need understand the difference.

Software QA (or *Quality Control* (QC)) is the process that helps identify the correctness, completeness, security compliance and the quality of the software. QA testing is *performed by the specialists* (testers, analysts). The goal of QA testing is to ensure that the application complies with a set of the predefined behavior requirements.

UAT is *performed by business users* or subject area experts. The UAT should result in the endorsement that the tested applications/functionality/module meets the agreed upon requirements. The results of UAT gives the confidence to the end-user that the system will perform in production according to specification.

During the QA process the specialist intends to perform all tests trying to break the application. This approach helps finding the errors undiscovered by developers. On the

contrary, during UAT the user runs business-as-usual scenarios and makes sure that business functions are implemented in the application.

Let's go over the strategies, approaches, and tools that will help you in test automation.

## Unit Testing

The *unit test* is a piece of code that invokes a method being tested. It *asserts* some assumptions about the application logic and behavior of the method. Typically you'll be writing such tests using unit-testing framework of your choice. Tests should run fast, be automated with clear output. For example you can test if a function is called with particular arguments, it should return expected result. We will take a closer look on unit testing terminology and vocabulary in a "["Test Driven Development" on page 263](#)" section.

## Integration Testing

Integration testing is a phase when already tested units are combined into a module to test the interfaces between them. You may want to test the integration of your code with the code written by other developers, e.g. some third-party framework. Integration tests ensure that any abstraction layers we build over the third-party code work as expected. Both unit and integration tests are written by application developers.

## Functional Testing

Functional testing is aimed at finding out whether the application properly implements business logic. For example, if the user clicks on a row in the grid with customers, the program should display a form view with specific details about the selected customer. In functional testing business users should define what has to be tested, unlike unit or integration testing where tests are created by software developers.

Functional tests can be performed manually by a real person clicking through each and every view of the web application confirming that it operates properly or reporting discrepancies with the functional specifications.

But there are tools to automate the process of functional testing of Web applications. Such tools allow to record the users' actions and replay them in the automatic mode. Below are brief descriptions of two of such tools - Selenium and CasperJS.

- **Selenium**

Selenium is an advanced browser automation tool suite that has capabilities to run and record user scenarios without requiring developers to learn any scripting languages. Also Selenium has an API for integration with many programming languages like Java, C#, JavaScript and etc. Selenium uses the WebDriver API to talk to the browsers and receive running context information. WebDriver is becoming

the standard API for browser automation. Selenium supports a wide range of browsers and platforms.

- **Casper.js**

CasperJS is a scripting framework written in JavaScript. CasperJS allows to create interaction scenarios like defining and ordering the navigation steps, filling and submitting forms or even scrapping Web content and making Web page screenshots. CasperJS works on top of PhantomJS and SlimerJS browser, which limits the testing runtime environment to WebKit-based and Gecko-based browsers. Still it's a very useful tool when you want to run tests in a Continuous Integration (CI) environment. You can read more about [PhantomJS and SlimerJS](#) and [CI](#) in the corresponding sidebars later in this chapter.

## What is PhantomJS and SlimerJS?

PhantomJS is a headless WebKit-based rendering engine and interpretation with JavaScript API. Think of PhantomJS as a browser that doesn't have any graphical user interface. PhantomJS can execute HTML, CSS, and JavaScript code. Because PhantomJS is not required to render browser's GUI, it can be used in display-less environments (e.g. CI server) to run tests. SlimerJS follows the same idea of headless browser, similar to PhantomJS, but it uses Gecko engine instead.

PhantomJS is built on top of Webkit and [JavaScriptCore](#) (like Safari) and SlimerJS is built on top of Gecko and [SpiderMonkey](#) (like Firefox). You can find the comprehensive list of difference between PhantomJS and SlimerJS APIs in [SlimerJS's documentation site](#).

In our case, Grunt automatically spawns the PhantomJS instance, executes the code of our tests, reads the execution results using PhantomJS API, and prints them out in the console. If you're not familiar with Grunt tasks, please, refer to [Appendix A](#) for additional information about usage Grunt in our Save The Child project.

## Load Testing

Load testing is a process that can help in answering the following questions:

- How many concurrent users can work with your application without bringing your server to its knees?
- Even if your server is capable of serving a thousand users, is your application performance in a compliance with the Service Level Agreement (SLA), if any?

It all comes down to two factors: availability and response time of your application. Ideally, these requirements should be well defined in the SLA document, which should clearly state what metrics are acceptable from the user's perspective. For example, the

SLA can include a clause stating that the initial download of your application shouldn't take longer than 10 seconds for users with a slow connections (under 1Mbps). SLA can also state that the query to display a list of customers shouldn't run for more than five seconds, and the application should be operational 99.9 percent of the time.

To avoid surprises after going live with your new mission-critical web application, don't forget to include in your project plan an item to create and run a set of heavy stress tests. Do this well in advance before your project goes live. With load testing, you don't need to hire a thousand of interns to play the roles of concurrent users to find out whether your application will meet the SLA.

Automated load testing software allows you to emulate required number of users, set up the throttling to emulate a slower connection, and configure the ramp-up speed. For example, you can simulate a situation where the number of users logged on to your system grows at the speed of 50 users every 10 seconds. Stress testing software also allows you to pre-record the users interactions, and then you can run these scripts emulating a heavy load.

Professional stress-testing software allows simulating the load close to the real-world usage patterns. You should be able to create and run mixed scripts simulating a situation in which some users are logging on to your application while others are retrieving the data and performing data modifications. Below are some tools worth considering for load testing.

- **Apache Benchmark**

Apache Benchmark is a simple to use command line tool. For example, with a command `ab -n 10 -c 10 -t 60 http://savesickchild.org:8080/ssc_extjs/` Apache Benchmark will open 10 concurrent connection with the server and will send 10 requests via each connection to simulate 10 visitors working with your Web application during 60 seconds. The number of concurrent connections is the actual number of concurrent users. You can find an Apache Benchmark sample report in [following snippet](#).

*Example 7-1. A sample Apache Benchmark report*

```
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
Server Software:   GlassFish
Server Hostname:  savesickchild.org
Server Port:      8080
Document Path:   /ssc_extjs/
Document Length: 306 bytes
Concurrency Level: 10
Time taken for tests: 60.003 seconds
Complete requests: 17526
Failed requests:  0
```

```

Total transferred: 11988468 bytes
HTML transferred: 5363262 bytes
Requests per second: 292086.73
Transfer rate: 199798.72 kb/s received
Connnection Times (ms)
    min avg max
Connect:   10  13  1305
Processing: 11  14  12
Total:     21  27  1317

```

- **jMeter**

Apache JMeter is a tool with a graphic user interface. It can be used to simulate heavy load on a server, network or an object to test its strength or to analyze overall performance under different load types. You can find more about testing web applications using JMeter in [official documentation](#).

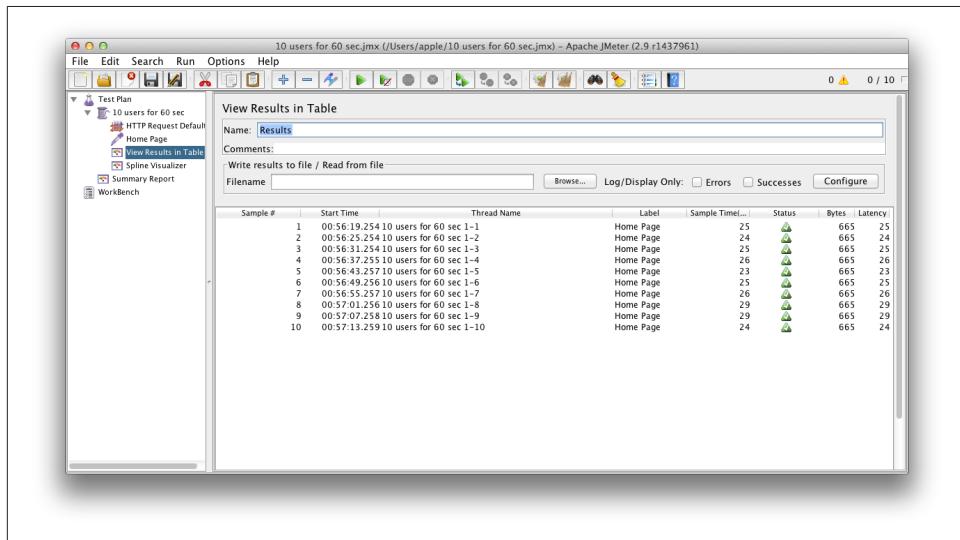


Figure 7-1. JMeter test results output example

- **PhantomJS**

Please, refer the sidebar called [What is PhantomJS](#) to make yourself familiar with this tool. The slide deck titled [“Browser Performance metering with PhantomJS”](#) is yet another good resource for seeing how PhantomJS can be used for performance testing.

# Test Driven Development

The methodology known as Test-Driven Development (TDD) substantially changes the way a traditional software development is done. This methodology wants you to write tests *even before* writing the application code. Instead of just using testing to verify our work *after it's done*, TDD moves the testing into the earlier application design phase. You should use the tests to clarify your ideas about what you are about to program. Here is fundamental mantra of TDD:

- Write a test and make it fail.
- Make the test pass.
- Refactor.
- Repeat.

This technique also referred as “Red-Green-Refactor” because IDE’s and test runners use red color to indicate failed tests and green color to indicate the tests that passed.

When you are about to start programming a class with some business logic, ask yourself, “How can I ensure that this function works as expected?” After you know the answer, write a test JavaScript class that calls this function *to assert* that the business logic gives the expected result.

An assertion is a true-false statement that represents what a programmer assumes about program state, e.g. `customerID >0` is an assertion.

According to [Martin Fowler](#), an assertion is a section of code that works only if certain conditions are true. It’s a conditional statement that is assumed to be always true. Failure of an assertion results in test failure.

Run your test, and it will immediately fail because no application code is written yet! Only after the test is written, start programming the business logic of your application

You should write a simplest possible piece of code to make the test pass. Don’t try to find a generic solution at this step. For example, if you want to test a calculator that needs to return 4 as result of  $2+2$  write the code what simply returns 4. Don’t worry about the performance or optimization at this point in time. Just make the test pass. Once you made it, you can refactor your application code to make it more efficient. Now you might want to introduce a real algorithm for implementing the application logic without worrying about breaking the contract with other components of your application.

A failed unit test indicates that your code change introduced *regression*, which is a new bug in a previously worked software. Automated testing and well-written test cases can reduce the likelihood of a regression in your code.

TDD allows to receive feedback from your code almost immediately. It's better to find that something is broken during development rather than in the application deployed in production.



Learn by heart The Golden Rule Of TDD:

Never write new functionality without a failing test

In addition to business logic, web applications should be tested for proper rendering of UI components, changing view states, dispatching, and handling events.

With any testing framework, your tests will follow same basic pattern. First, you need to setup up the test environment. Second, you run the production code and check that it works as it supposed to. Finally, you need to clean up after the test will run - remove everything that your program has created during setup of the environment.

This pattern for authoring unit tests is called *Arrange-Act-Assert-Reset* (AAAR<sup>1</sup>).

- In the *Arrange* phase you set up the unit of work to test. For example, create a JavaScript objects, prepare dependencies and etc.
- In the *Act* phase you exercise the unit under test and capture the resulting state. You execute your production code in unit test context.
- In the *Assert* phase you verify the behavior through assertions.
- In the *Reset* phase, you reset the environment to the initial state. For example, erase the DOM elements created in the *Arrange* phase. Most of the frameworks provide a “teardown” function that would be invoked after test is done.

Later in this chapter, you'll see how different frameworks implements AAAR pattern.

In next sections we will dive into the testing frameworks for JavaScript.

## Test-Driven Development With QUnit

We'll start our journey to JavaScript testing frameworks with **QUnit**, which was originally developed by **John Resig** as part of jQuery. QUnit now runs completely standalone and doesn't have any jQuery dependencies. While it's still being used by the jQuery Project itself for testing jQuery, jQuery UI and jQuery Mobile code, QUnit can be used to test any generic JavaScript code.

1. [Principles for Test-Driven Development](#)

## **Setup Grunt With Qunit**

In this section you're going learn how to automatically run the Qunit tests using Grunt. Let's setup our project by adding the Qunit framework and tests file. Start with downloading the latest version from using bower.

*Example 7-2. Installing qunit with bower*

```
bower install qunit
```

You need to get only two files - `qunit-* .js` and `qunit-* .css`.

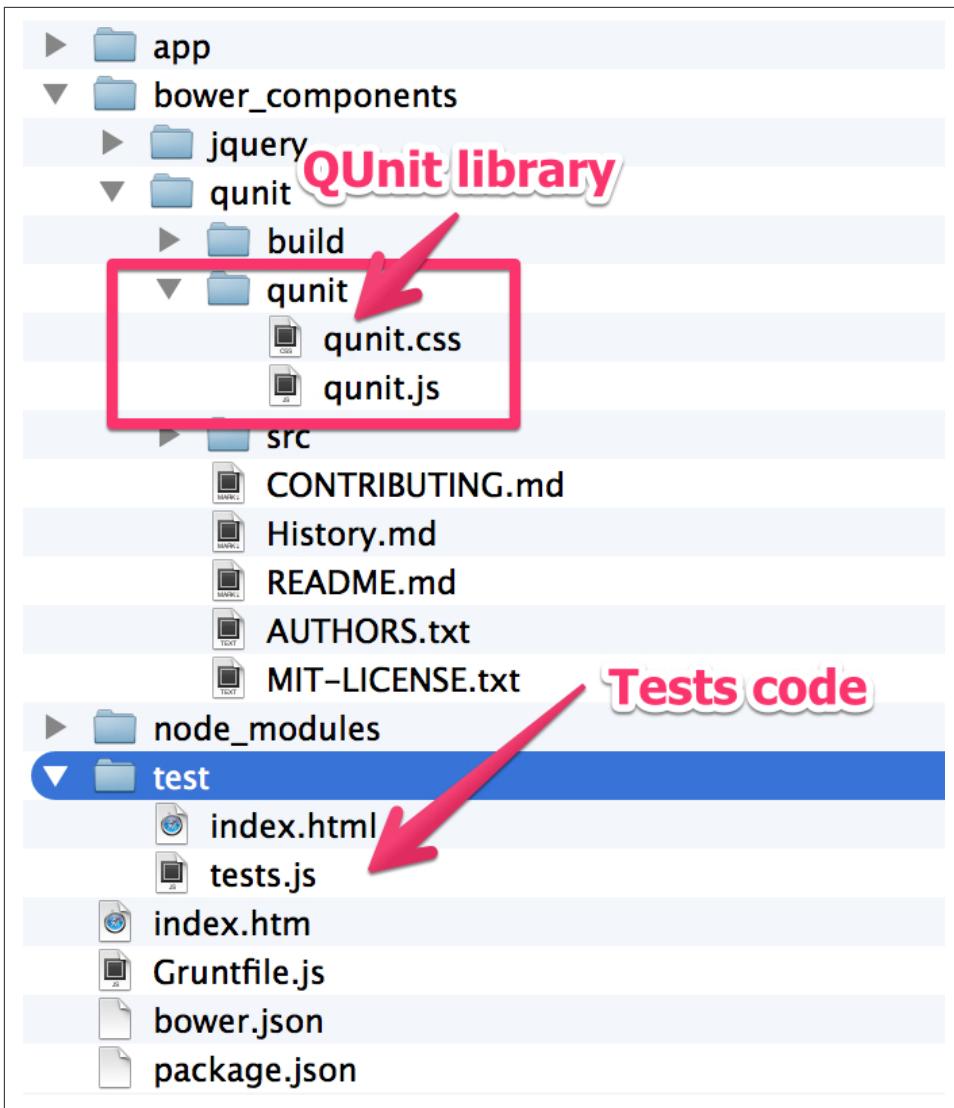


Figure 7-2. QUnit framework in our project

Example 7-3. Our first QUnit test

```
'use strict';
test('my first qunit test', function() {
  ok(2 + 2 === 4, 'Passed!');
});
```

You'll also need a test runner for the test setup. A test runner is an html file contains links to QUnit framework JavaScript file.

*Example 7-4. A test runner*

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Test Suite</title>
  <link rel="stylesheet" href="../bower_components/qunit/qunit/qunit.css" media="screen">
  <script src="../bower_components/qunit/qunit/qunit.js"></script>
  <script src="../bower_components/jquery/dist/jquery.min.js" type="text/javascript"></script> ①
  <script src="test/tests.js" type="text/javascript" charset="utf-8"></script> ②
</head>
<body>
<div id="qunit"></div>③
<div id="qunit-fixture">
  ④
</div>
</body>
</html>
```

- ① In this section we continue working on the jQuery-based version of the Save The Child application. Hence our “production environment” depends on availability of jQuery, so we need to include jQuery in the test runner.
- ② Test files are included too.
- ③ QUnit fills this block with results.
- ④ Any HTML you want to be present in each test. It will be reset for each test.

To run all our tests we need to open the `qunit-runner.html` in browser. (See [Figure 7-3](#)).

*Example 7-5. Grunt config for qunit test runner*

```
module.exports = function(grunt) {
  'use strict';
  grunt.initConfig({
    qunit: {
      all: ['test/qunit-runner.html']
    }
  });
  grunt.registerTask('test', 'qunit');
  grunt.loadNpmTasks('grunt-contrib-qunit'); // ①
};
```

- ① Grunt loads task from local NPM repository. To install this tasks in `node_modules` directory use command `npm install grunt-contrib-qunit`.

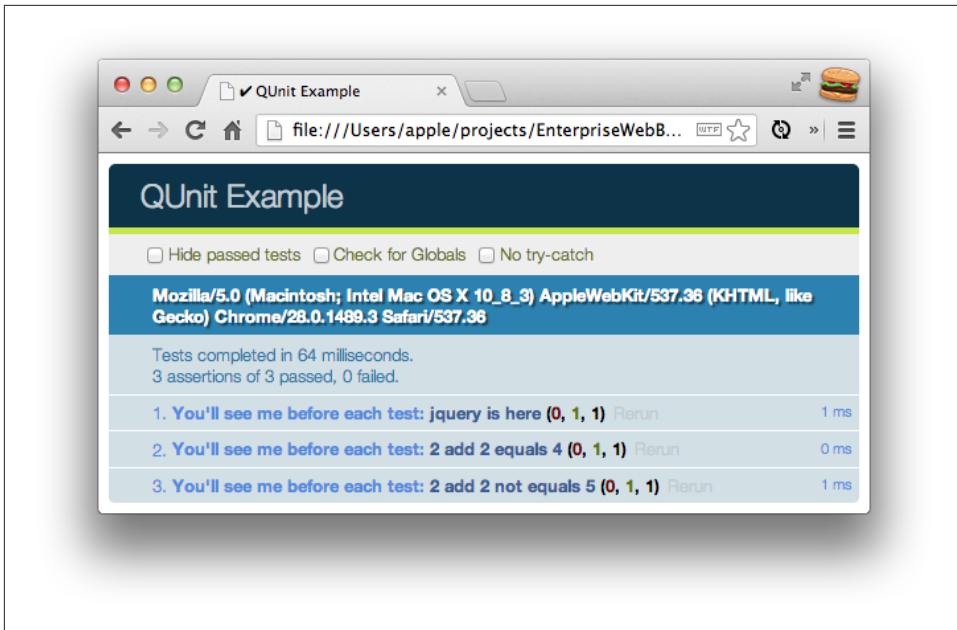


Figure 7-3. A test run results in browser

Now let's briefly review QUnit API components. You can find a typical QUnit script in the following listing.

*Example 7-6. A sample QUnint test*

```
(function($) {
    module('SaveSickChild: login component test', { // ①
        setup: function() { // ②
            // test setup code goes here
        },
        teardown: function() { // ③
            // test cleanup code goes here
        }
    });
    test('jquery is here', function() { // ④
        ok($, "yes, it's here");
    });
    test("2 add 2 equals 4", function() {
        ok(2 + 2 === 4, "Passed!"); // ⑤
    });
    test('2 add 2 not equals 5', function() {
        notEqual(2 + 2, 5, "failed"); // ⑥
    });
})(jQuery); // ⑦
```

- ① A module function allows to combine related tests as group.

- ② Here we can run *Arrange* phase. A `setup` function will be called before each test.
- ③ A `teardown` function will be call after each test respectively. This is our *Reset* phase.
- ④ You need to place code of your test in corresponding `test` function.
- ⑤ Typically, you need to use assertions to make sure the code being tested gives expected results. The function `ok` will examine the first argument to be `true`.
- ⑥ A pair of functions `equal` and `notEqual` will check for the equivalence of the first and second arguments, which could be expressions as well.
- ⑦ A code of the test is wrapped in IIFE and passes `jQuery` object as `\$` variable.

You can find more details about QUnit in [product documentation](#) and [QUnit Cookbook](#).

## Behavior-Driven Development With Jasmine

The idea behind *behavior-driven development* (BDD) is to use the natural language constructs to describe what you think your code should be doing or more specifically, what your functions should be returning.

Similarly to unit tests, with BDD you write short specifications that test one feature at a time. Specifications should be sentences. For example, “Calculator adds two positive numbers”. Such sentences will help you to easily identify the failed test by simply reading this sentence in the resulting report.

Now we'll demonstrate this concept using Jasmine - the BDD framework for JavaScript. Jasmine provides a very nice way to group, execute, and report JavaScript unit tests.

### Setup Grunt with Jasmine

Now let's learn how to execute a Jasmine specification with Grunt. We will cover Jasmine basics in the next section, but for now think of Jasmine as a piece of code that should be executed by Grunt.

Let's start with downloading the latest version of Jasmine using bower.

```
bower install jasmine
```

Unzip `jasmine-standalone-2.0.0.zip` in `dist` directory. Jasmine comes with an example spec (`spec` folder) and an html test runner - `SpecRunner.html`. Let's open the file `SpecRunner.html` in browser [Figure 7-4](#).

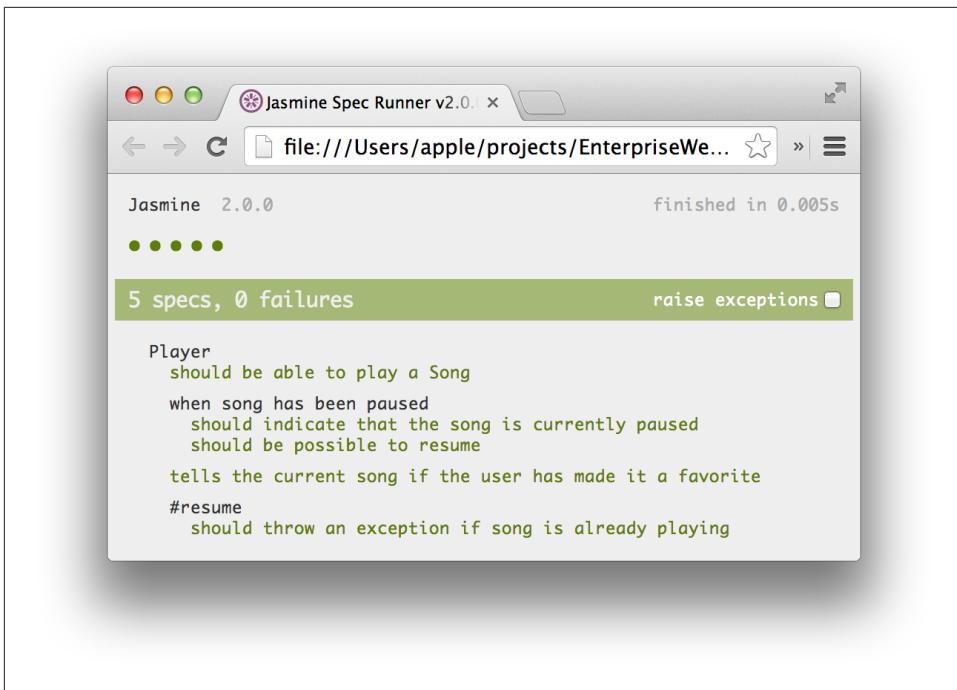


Figure 7-4. Running Jasmine Specs in a Browser

The SpecRunner.html is structured similarly to QUnit html runner. You can run specifications by opening the runner file in browser.

```
<!DOCTYPE HTML>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <title>Jasmine Spec Runner v2.0.0</title>

  <link rel="shortcut icon" type="image/png" href="lib/jasmine-2.0.0/jasmine_favicon.png">
  <link rel="stylesheet" type="text/css" href="lib/jasmine-2.0.0/jasmine.css">

  <script type="text/javascript" src="lib/jasmine-2.0.0/jasmine.js"></script> ①
  <script type="text/javascript" src="lib/jasmine-2.0.0/jasmine-html.js"></script>

  <script type="text/javascript" src="lib/jasmine-2.0.0/boot.js"></script> ②

  ③
  <script type="text/javascript" src="src/Player.js"></script>
  <script type="text/javascript" src="src/Song.js"></script>

  ④
  <script type="text/javascript" src="spec/SpecHelper.js"></script>
  <script type="text/javascript" src="spec/PlayerSpec.js"></script>
```

```
</head>
```

```
<body>
</body>
</html>
```

- ❶ Required Jasmine framework library.
- ❷ Initialize Jasmine and run all specifications when the page is loaded.
- ❸ Include the source files.
- ❹ Include the specification code. It's not required but files that contain specification code can have suffix `*Spec.js`.

Now let's update the Grunfile to run the same sample specifications with the PhantomJS headless browser. Copy the content of `src` folder of your Jasmine distribution into the `app/js` folder of our project, and then copy the content of the `spec` folder into the `test/spec` folder of your project. Also create a folder `test/lib/jasmine` and copy the content of Jasmine distribution `lib` folder there. [Figure 7-5](#)

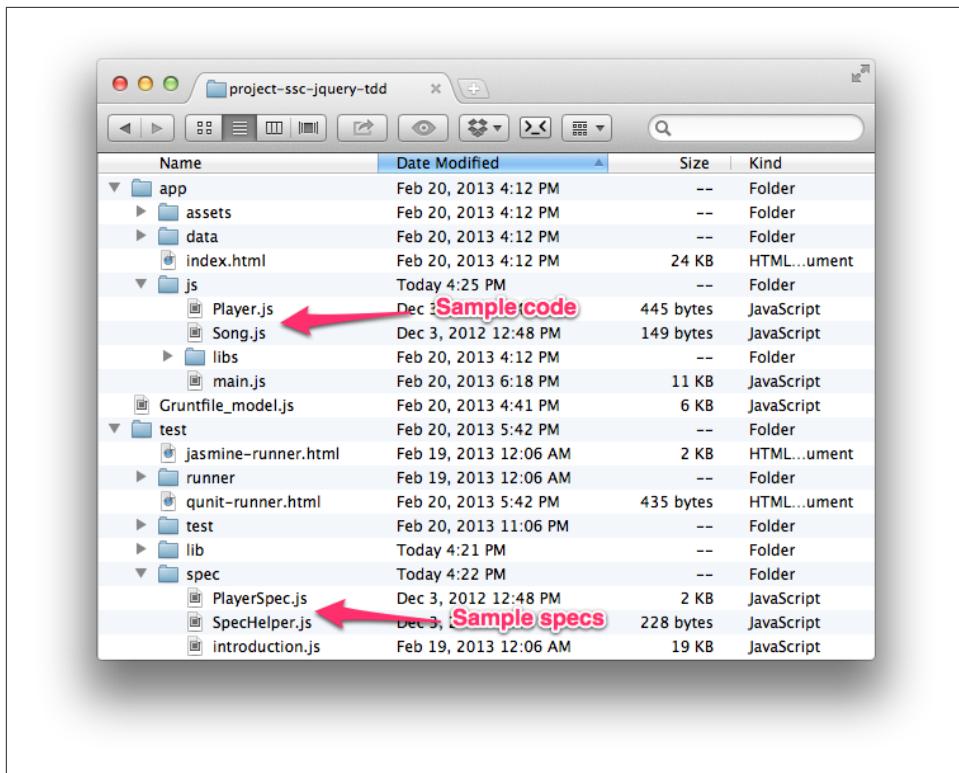


Figure 7-5. Jasmine Specifications in Our Project

Now you need to edit Gruntfile\_jasmine.js to activate Jasmine support.

*Example 7-7. Gruntfile\_jasmine.js with Jasmine running support*

```
module.exports = function(grunt) {
  'use strict';

  grunt.initConfig({
    jasmine: { // ❶
      src: ['app/Player.js', 'app/Song.js'], // ❷
      options: {
        specs: 'test/spec/PlayerSpec.js', // ❸
        helpers: 'test/spec/SpecHelper.js' // ❹
      }
    }
  });

  // Alias the `test` task
  grunt.registerTask('test', 'jasmine');
  // loading jasmine grunt module
```

```
grunt.loadNpmTasks('grunt-contrib-jasmine'); // ⑤  
};
```

- ① Configuring the Jasmine task.
- ② Specifying the location of the source files.
- ③ Specifying the location of Jasmine specs.
- ④ Specifying the location of Jasmine helpers, which will be covered later in this chapter.
- ⑤ Grunt loads task from local NPM repository. To install this tasks in `node_modules` directory use command `npm install grunt-contrib-jasmine`.

To execute tests, run the command `grunt --gruntfile Gruntfile_jasmine.js jasmine`, and you should see something like this:

```
Running "jasmine:src" (jasmine) task  
Testing jasmine specs via phantom  
.....  
5 specs in 0.003s.  
>> 0 failures  
  
Done, without errors.
```

In this example, Grunt successfully executed the tests with **PhantomJS** of all five specifications defined in `PlayerSpec.js`.

## What is Continuous Integration?

Continuous Integration (CI) is a software development practice where members of a team integrate their work frequently, which results in multiple integrations per day.

Introduced by Martin Fowler and Matthew Foemmel, the theory of continuous integration<sup>2</sup> recommends creating scripts and running automated builds (including tests) of your application at least once a day. This allows you to identify issues in the code early.

Authors of this book successfully use an open source framework called **Jenkins** (there are other similar CI servers) for establishing continuous build process.

With Jenkins, you can have scripts that run either at a specified time interval or on each source code repository check-in of the new code. You may also force an additional build process whenever you like. The Grunt command line tool should be installed and be available on CI machine to allow the Jenkins server invoke Grunt scripts and publish test results.

2. [Continuous Integration by Martin Fowler](#)

We use it to ensure continuous builds of the internal and open source projects.

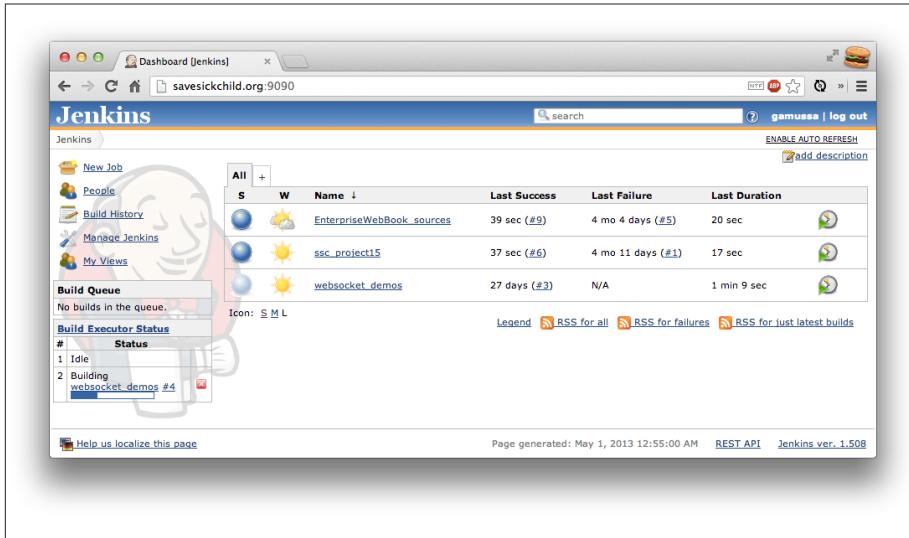


Figure 7-6. Jenkins CI server running at [savesichild.org](http://savesichild.org) website and used to build the sample applications for this book.

In the next section you will learn how write your own specifications.

## Jasmine Basics

After we've set up the tools for running tests, let's start developing tests and learn the Jasmine framework constructs. Every specification file has a set of *suites* defined in the `describe` function. Suites help logically organize code of test specifications.

Example 7-8. *ExampleSpec.js*

```
describe("My function under test should", function() { // ❶
  it("return on", function() { // ❷
    // place specification code here
    //
  });
  describe("another suite", function() { // ❸
    it("spec1", function() {

    });
  });
  it("my another spec", function() { // ❹
    var truth = true;
    expect(truth).toBeTruthy();
  });
});
```

```

it("2+2 = 4", function() {
  expect(2 + 2).toEqual(4); // ⑤
});

```

- ❶ The function `describe()` accepts two parameters - the name of the test suite, and the callback function. The function is a block of code that implements the suite. If for some reasons you would like to skip the suite from execution you can just use method `xdescribe()` and a whole suite will be excluded until you rename it back to `describe()`.
- ❷ The function `it()` also accepts similar parameters - the name of the test specification, and the function that implements this specification. Like in case with suites, Jasmine has a corresponding `xit` method to exclude specification from execution.
- ❸ Each suite can have any number of nested suites.
- ❹ Each suite can have any number of specifications.
- ❺ The code checks to see if `2+2` equals `4`. We used the function `toEqual()`, which is a *matcher*. Define expectations with the function `expect()`, which takes a value, called the actual. It's chained with a matcher function, which takes the expected value (in our case it's `4`) and checks if it satisfies the criterion defined in the matcher.

Various flavors of matchers are shipped with Jasmine framework, and we're going to review a couple the frequently used matchers functions.

- Equality

Function `toEqual()` checks if two things are equal.

- True or False?

Functions `toBeTruthy()` and `toBeFalsy()` checks if something is true or false respectively.

- Identity

Function `toBe()` checks if two things are *the same object*.

- Nullness

Function `toBeNull()` checks if something is `null`.

- Is Element Present

Function `toContain()` check if an actual value is an element of array.

```
expect(["James Bond", "Austin Powers", "Jack Reacher", "Duck"]).toContain("Duck");
```

- Negate Other Matchers

This function is used to reverse matchers to ensure that they aren't `true`. To do that, simply prefix things with `.not`:

```
expect(["James Bond", "Austin Powers", "Jack Reacher"]).not.toContain("Duck");
```

Above we've listed only some of existing matchers. You can find the complete documentation with code examples at [official Jasmine website](#) and at [wiki](#).



There is a large set of jquery-specific matchers available <https://github.com/velesin/jasmine-jquery>

## Specification Setup

Jasmine framework has an API to arrange your specification (based on [\[AAAR\]](#) concept). It includes two methods - `beforeEach()` and `afterEach()`, which allow you to execute some code before and after each spec respectively. It's very useful for instantiation of the shared objects or cleaning up after the tests complete. If you need to fulfill your test with some common dependencies or setup the environment, just place code inside `beforeEach()` method. Such dependencies and environment are known as *fixture*.

### What is Fixture?

Test fixture refers to the fixed state used as a baseline for running tests. The main purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable. Sometimes a fixture also referred as *test context*.

#### Example 7-9. Specification setup with `beforeEach`

```
(function($) {
  describe("DOM manipulation spec", function() {
    var usernameInput;
    var passwordInput;
    beforeEach(function() { // ❶
      usernameInput = document.createElement("input"); // ❷
      usernameInput.setAttribute("type", "text");
      usernameInput.setAttribute("id", "username");
      usernameInput.setAttribute("name", "username");
      usernameInput.setAttribute("placeholder", "username");
      usernameInput.setAttribute("autocomplete", "off");

      passwordInput = document.createElement("input");
      passwordInput.setAttribute("type", "text");
```

```

passwordInput.setAttribute("id", "password");
passwordInput.setAttribute("name", "password");
passwordInput.setAttribute("placeholder", "password");
passwordInput.setAttribute("autocomplete", "off");
});

afterEach(function() { // ③
});

it("jquery should be present", function() {
  expect($).not.toBeNull();
});
it("inputs should exist", function() {
  expect(usernameInput.id).toBe("username");
  expect(passwordInput.id).toBe("password");
});
it("should not allow login with empty username and password and return code equals 0", function() {
  var result = ssc.login(usernameInput, passwordInput); // ④
  expect(result).toBe(0);
});
it("should allow login with user admin and password 1234 and return code equals 1", function() {
  usernameInput.value = "admin"; // ⑤
  passwordInput.value = "1234";
  var result = ssc.login(usernameInput, passwordInput);
  expect(result).toBe(1);
});
});
})(jQuery);

```

- ① This method will be called before each specification.
- ② In the `beforeEach()` method we create two input fields. These two inputs will be available in all specifications of this suite.
- ③ You can place additional cleanup code inside `afterEach()` function.
- ④ A `beforeEach()` function helps to implement *Don't Repeat Yourself* principle in our tests. You don't need to create the dependency elements inside each specification manually.
- ⑤ You can change defaults inside each specification without worrying about affecting other specifications. Your test environment will be reset for each specification.

## Custom Matchers

Jasmine framework is easily extensible, and it allows you to define your own matchers if for some reasons you're unable to find the appropriate matchers in the Jasmine distribution. In such cases you'd need to write a custom matcher. Let's write a matcher that check if a string contains name of the “secret agent” from the defined list of agents.

*Example 7-10. Custom `toBeSecretAgent` matcher*

```
beforeEach(function() {
  'use strict';
  var customMatcher = {
    toBeSecretAgent: function() {
      return {
        compare: function(actual, expected) { //❶
          if (expected === undefined) {
            expected = '';
          }
          var result = {};// ❷
          var agentList = [
            'James Bond',
            'Ethan Hunt',
            'Jason Bourne',
            'Aaron Cross',
            'Jack Reacher'
          ];
          result.pass = agentList.indexOf(actual) !== -1; // ❸
          if (result.pass) { // ❹
            result.message = actual + ' is a supper agent'; // ❺
          } else {
            result.message = actual + ' is not a secret agent';
          }
          return result;
        }
      };
    }
  };
  jasmine.addMatchers(customMatcher);
});
```

- ❶ We need to implement function `compare` that accepts two parameters from `expect` call - `actual` and `expected` values.
- ❷ A function `compare` should return `result` object.
- ❸ This function checks if `agentsList` contains the `actual` value.
- ❹ A `pass` property of `result` object indicates success or failure of matcher execution;
- ❺ We can customize error message (a `message` property of `result` object) if the test fails.

The invocations of this helper can look like this:

```
it("part of super agents", function () {
  expect("James Bond").toBeSecretAgent(); // ❶
  expect("Jason Bourne").toBeSecretAgent();
```

```
expect("Austin Powers").not.toBeSecretAgent(); // ❷
expect("Austin Powers").toBeSecretAgent(); // ❸
});
```

- ❶ Calling the custom matcher.
- ❷ Custom matchers could be used together with the `.not` modifier.
- ❸ This expectation will fail because *Austin Powers* is not in the list of secret agents.

The following custom failure message will be displayed on the console.

```
grunt --gruntfile Gruntfile_jasmine.js test
Running "jasmine:src" (jasmine) task
Testing jasmine specs via PhantomJS
My function under test should
  ✓ return on
    another suite
    ✓ spec1
  ✓ my another spec
  ✓  $2+2 = 4$ 
  X part of super agents
    Austin Powers is not a secret agent (1)

5 specs in 0.01s.
>> 1 failures
Warning: Task "jasmine:src" failed. Use --force to continue.

Aborted due to warnings.
```

“Austin Powers is not a secret agent (1)” is a custom failure message.

## Spies

Test spies are objects that replace the actual functions with the code to record information about the function’s usage through the systems being tested. Spies are useful when determining a function’s success is not easily accomplished by inspecting its return value or changes to the state of objects with which it interacts.

Consider the following example of login functionality. A `showAuthorizedSection()` function will be invoked within `login` function after the user entered the correct user name and password. We need to test that the invocation of the `showAuthorizedSection()` is happening in this sequence.

*Example 7-11. Production code of login function.*

```
var ssc = [];
(function() {
  'use strict';
  ssc.showAuthorizedSection = function() {
    console.log("showAuthorizedSection");
  };
});
```

```

ssc.login = function(usernameInput, passwordInput) {
    // username and password check logic is omitted
    this.showAuthorizedSection();
};

})();

```

And here is how we can test it using Jasmine's spies.

```

describe("login module", function() {
    it("showAuthorizedSection has been called", function() {
        spyOn(ssc, "showAuthorizedSection"); // ①
        ssc.login("admin", "1234"); // ②
        expect(ssc.showAuthorizedSection).toHaveBeenCalledWith(); // ③
    });
});

```

- ❶ The `spyOn` function will replace `showAuthorizedSection()` function with corresponded spy.
- ❷ The `showAuthorizedSection()` function will be invoked within `login()` function in case of successful login.
- ❸ Assertion `toHaveBeenCalledWith()` would be not possible without spy.

## Multi-Browser Testing

The previous section was about executing your test and specification in a headless mode using Grunt and PhantomJS, which is very useful for running tests in the CI environments. While PhantomJS uses WebKit rendering engine, there are browsers that don't use WebKit. It's obvious that running tests manually in each browser is tedious and not productive. To automate testing in all Web browsers, you can use Testem Runner. Testem executes your tests, analyzes its output and prints result on the console. In this section you'll learn how to install and configure Testem to run Jasmine tests.

### Installation

Testem uses Node.js APIs and can be installed with NPM:

```
npm install testem -g
```

### Testem Configuration file

Testem runner will just pick any of JavaScript in your project directory. If testem can identify any tests among that `.js` files it will run it. But Testem tasks can be customized using a configuration file.

You can configure Testem to specify which files should be included in testing. Testem starts with trying to find the configuration file `testem.json` in the project directory. A sample `testem.json` is shown in [following listing](#).

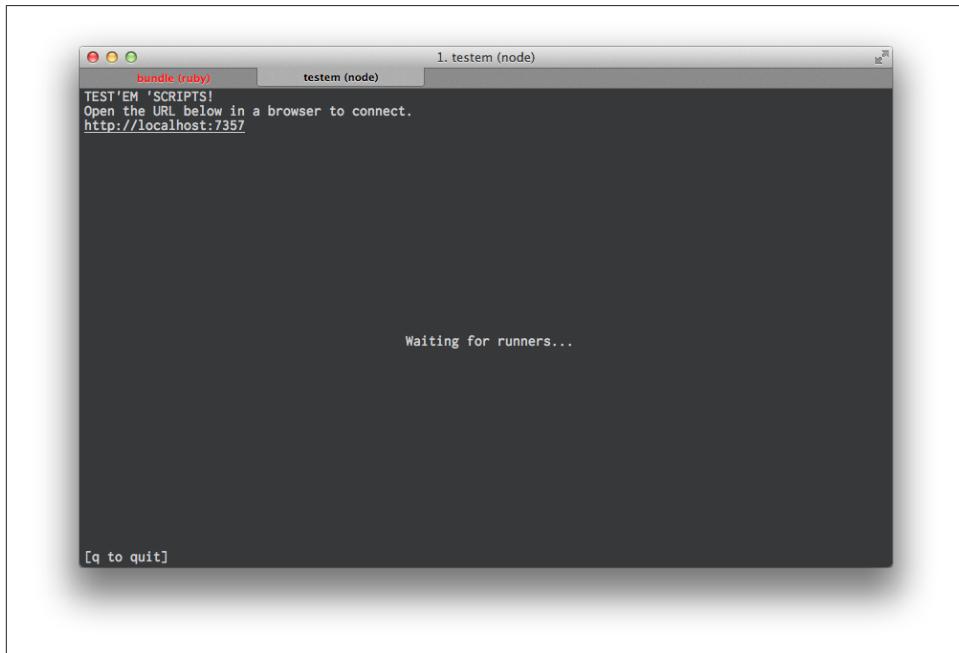
*Example 7-12. A Testem Configuration File*

```
{  
  "framework": "jasmine2",           // ①  
  "src_files": [                     // ②  
    "ext/ext-all.js",  
    "test.js"  
  ]  
}
```

- ❶ The `framework` directive is used to specify the test framework. Testem supports QUnit, Jasmine and many more frameworks. You can find full list of supported frameworks on testem [github page](#).
- ❷ The list of test and production code source files.

## Running Tests

Testem supports two running modes: test-driven development mode (*tdd-mode*) and continuous integration (*ci-mode*). (For more about continuous integration, see the note on [CI](#). In tdd-mode, testem starts the development server.



*Figure 7-7. Testem tdd-mode.*

In tdd-mode, Testem doesn't spawn any browser automatically. On the contrary, you'd need to open this url in the browser you want run test against to connect it to Testem server. From this point on, Testem will execute tests in all connected browsers. On the next **screenshot** you can see we added different browsers including mobile version of Safari (running on iOS simulator).

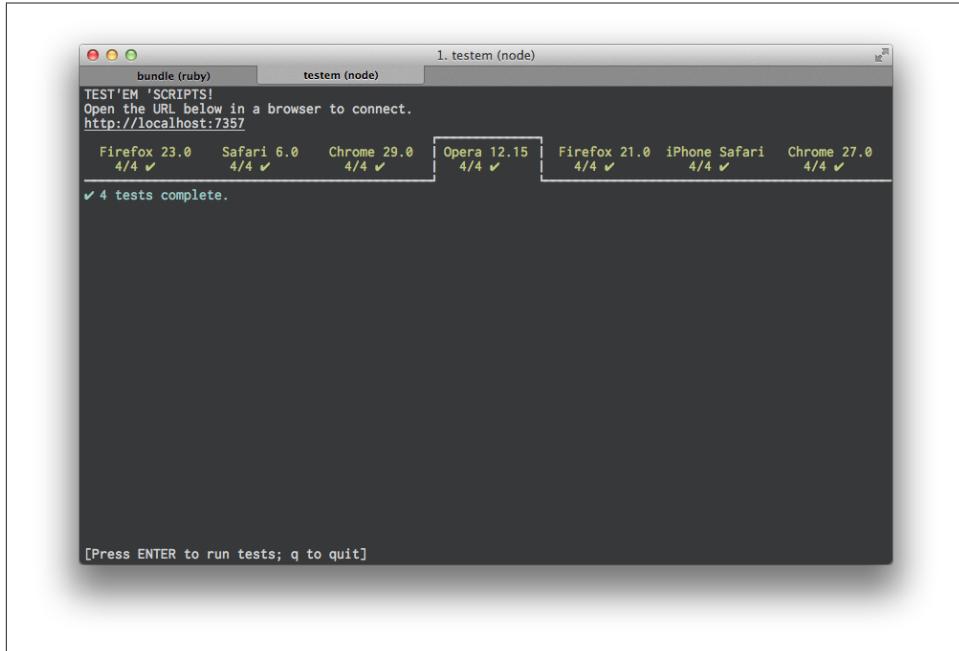


Figure 7-8. Testem is running the tests on the multiple browsers.

Because the Testem server itself is an HTTP server, you can connect remote browsers to it as well. For example, the **following screenshot** shows Internet Explorer 10 running under Windows 7 virtual machine connected to the testem server.

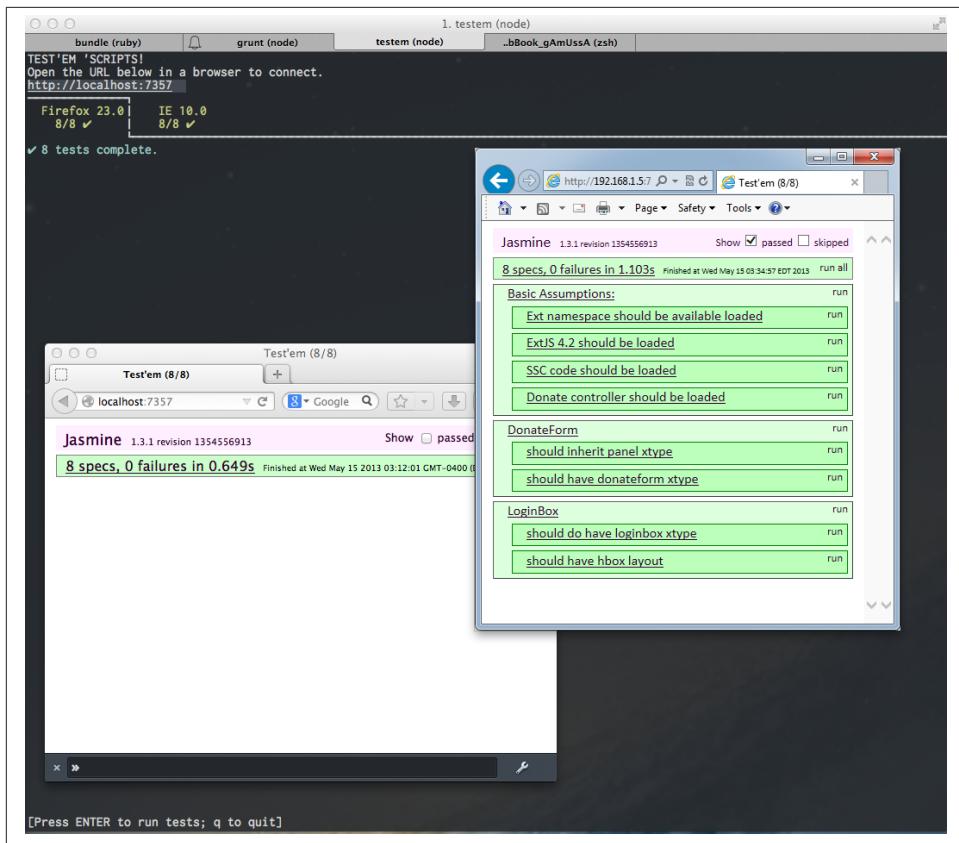


Figure 7-9. Using testem to test code on remote IE 10

Running the tests with testem runner can be combined with previously introduced Grunt tool. The [next screenshot](#) shows two tests in parallel: testem runs tests on the real browsers and and grunt runs tests on the headless PhantomJS.

```

TEST'EM 'SCRIPTS!
Open the URL below in a browser to connect.
http://localhost:7357
IE 10.0 | Firefox 23.0 | Chrome 28.0
4/4 ✓ | 4/4 ✓ | 4/4 ✓
4 tests complete.

1. testem (node)
3. grunt watch (node)
...sc-jquery-tdd (zsh)

e.
Aborted due to warnings.
Completed in 5.24s at Mon May 06 2013 11:49:27 GMT-0400 (EDT) - Waiting...OK
>> File "test/spec/BasicAssumptions.js" changed.
Running "sencha_jasmine:keepRunner" (sencha_jasmine) task
Running "sencha_jasmine_wrapper:keepRunner" (sencha_jasmine_wrapper) task
Testing jasmine specs via phantom
...
4 specs in 0.13s.
>> 0 failures
Done, without errors.

Completed in 5.01s at Mon May 06 2013 11:49:51 GMT-0400 (EDT) - Waiting...OK
>> File "test/spec/BasicAssumptions.js" changed.
Running "sencha_jasmine:keepRunner" (sencha_jasmine) task
Running "sencha_jasmine_wrapper:keepRunner" (sencha_jasmine_wrapper) task
Testing jasmine specs via phantom
...
Basic Assumptions: :: something truthy: failed
  Expected false to be truthy. (1)
4 specs in 0.158s.
>> 1 failures
Warning: Task "sencha_jasmine_wrapper:keepRunner" failed. Use --force to continue.
e.
Aborted due to warnings.
Completed in 8.86s at Mon May 06 2013 12:16:14 GMT-0400 (EDT) - Waiting...OK
>> File "test/spec/BasicAssumptions.js" changed.
Running "sencha_jasmine:keepRunner" (sencha_jasmine) task
Running "sencha_jasmine_wrapper:keepRunner" (sencha_jasmine_wrapper) task
Testing jasmine specs via phantom
...
4 specs in 0.241s.
>> 0 failures
Done, without errors.

Completed in 7.61s at Mon May 06 2013 12:17:41 GMT-0400 (EDT) - Waiting...

```

[Press ENTER to run tests; q to quit]

Figure 7-10. Using testem and grunt watch side-by-side

Testem supports live reloading mode. This means that Testem will watch file system for changes and will execute tests in all connected browsers automatically. You can force to test run by switching to console and hitting the *Enter* key.

In CI mode testem will examine system for all of the available browsers in your system and will execute tests on it. You can get list of the browsers that testem can use to run tests with the `testem launchers` command. [Here is sample output](#) after running this command.

```
# testem launchers
Have 5 launchers available; auto-launch info displayed on the right.
```

Launcher	Type	CI	Dev
Chrome	browser	✓	
Firefox	browser	✓	
Safari	browser	✓	
Opera	browser	✓	
PhantomJS	browser	✓	

Now you can run our test simultaneously in all browsers installed in your computer - Google Chrome, Safari, Firefox, Opera and PhantomJS - with one command:

```
testem ci
```

*Example 7-13. An output of testem ci command*

```
# Launching Chrome      # ①
#
# Launching Firefox     # ②
# ....
TAP version 13
ok 1 - Firefox Basic Assumptions: Ext namespace should be available loaded.
ok 2 - Firefox Basic Assumptions: ExtJS 4.2 should be loaded.
ok 3 - Firefox Basic Assumptions: SSC code should be loaded.
ok 4 - Firefox Basic Assumptions: something truthy.

# Launching Safari      # ③
#
# Launching Opera        # ④
# ....
ok 5 - Opera Basic Assumptions: Ext namespace should be available loaded.
ok 6 - Opera Basic Assumptions: ExtJS 4.2 should be loaded.
ok 7 - Opera Basic Assumptions: SSC code should be loaded.
ok 8 - Opera Basic Assumptions: something truthy.

# Launching PhantomJS    # ⑤
#
1..8
# tests 8
# pass 8

# ok
....
```

① The tests are run on Chrome...

② ... Firefox

③ ... Safari

④ ... Opera

⑤ ... and on headless WebKit - PhantomJS

Testem uses **TAP** format to report test results.

## Testing DOM

As we discussed in Chapter 1, Document Object Model is a standard browser API that allows a developer to access and manipulate page elements. Pretty often your JavaScript code needs to access and manipulate the HTML page elements in some way. Testing DOM is the crucial part of testing your client side JavaScript. By design, the DOM standard defines a browser-agnostic API. But in the real world, if you want to make sure

that your code works in the particular browser you need to run the test inside this browser.

Earlier in this chapter we've introduced the Jasmine method `beforeEach()`, which is the right place for setting all required DOM elements and making them available in the specifications.

*Example 7-14. Using jQuery APIs to create the required DOM elements before run the spec*

```
describe("spec", function() {
  var usernameInput;
  beforeEach(function() { // ❶
    usernameInput = $(document.createElement("input")).attr({ // ❷
      type: 'text',
      id: 'username',
      name: 'username'
    })[0];
  });
});
```

- ❶ Inside the `beforeEach()` method we're using the API to manipulate the DOM programmatically. Also, if you're using an HTML test runner you can add the fixture using HTML tags. But we don't recommend this approach because pretty soon you will find that the test runner will become unmaintainable and clogged with tons of fixture HTML code.
- ❷ Create an `<input>` element using jQuery APIs, which will turn into the following HTML:

```
<input type="text" id="password" name="password" placeholder="password" autocomplete="off">
```

The jQuery selectors API is more convenient for working with DOM than a standard JavaScript DOM API. But in the future examples we will use the `jasmine-fixture` library for easier setup of the DOM fixture. Jasmine-fixture uses similar to jQuery selectors syntax for injecting HTML fixtures. With this library you will significantly decrease the amount of repetitive code while creating the fixtures.

Let's see how the example from [previous code snippet](#) looks like with the `jasmine-fixture` library.

*Example 7-15. Using jasmine.fixture to setup the DOM before spec run*

```
describe("spec", function() {
  var usernameInput;
  beforeEach(function() {
    usernameInput = affix('input[id="username"][type="text"][name="username"]')[0]; // ❶
  });

  it("should not allow login with empty username and password and return code equals 0", function() {
```

```

    var result = ssc.login(usernameInput, passwordInput); // ②
    expect(result).toBe(0);
  });
});

```

- ① Using the `affix()` function provided by the *jasmine-fixture* library and expressiveness of CSS selectors we can easily setup required DOM elements. More examples of possible selectors could be found at the [documentation page](#) of *jasmine-fixture*.
- ② Now when all requirements for our production code (`login()` function) are satisfied we can run it in the context of a test and assert the results.

As you can see, testing of the DOM manipulation code is much like any other type of unit testing. You need to prepare a fixture (a.k.a. the testing context), run the production code and assert the results.

## Save The Child With TDD

### The Test-Driven ExtJS version of *SaveSickChild.org*

We assume that you've read the materials from Chapter 6, and in this section you'll apply your newly acquired Ext JS skills. As a reminder, Ext JS framework encourages using MVC architecture. The separation of responsibilities between Views, Models and Controllers makes an ExtJS application a perfect candidate for unit testing. In this section you'll learn how to test the Ext JS version of the Save The Child application from Chapter 6.

#### Harnessing ExtJS application

Let's create a skeleton application that can provide for our classes under the test familiar environment.

*Example 7-16. A HTML-runner for Jasmine and ExtJS application*

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title id="page-title">ExtJS Jasmine Tester</title>
  <link rel="stylesheet" type="text/css" href="test/bower_components/jasmine/lib/jasmine-core/jasmin
  <script type="text/javascript" src=ext/ext-all.js></script> // ①

  <script type="text/javascript" src="test/bower_components/jasmine/lib/jasmine-core/jasmine.js"></s
  <script type="text/javascript" src="test/bower_components/jasmine/lib/jasmine-core/jasmine-html.js"
  <script type="text/javascript" src="test/bower_components/jasmine/lib/jasmine-core/boot.js"></scri

  <script type="text/javascript" src="test/bower_components/jasmine-fixture/dist/jasmine-fixture.mir

```

```

<script type="text/javascript" src="test.js"></script> // ③

</head>
<body>

</body>
</html>

```

- ① Adding ExtJS framework dependencies.
- ② Adding the Jasmine framework dependencies.
- ③ This is our skeleton ExtJS application that will setup “friendly” environment for components under the test. You can see content of test.js in the [following listing](#).

*Example 7-17. A ExtJS testing endpoint*

```

Ext.Loader.setConfig({
    disableCaching: false,
    enabled: true,
    paths: {
        Test: 'test', // ①
        SSC: 'app' // ②
    }
});

var application = null;

Ext.onReady(function() {
    application = Ext.create('Ext.app.Application', {
        name: 'SSC', // ③
        requires: [
            'Test.spec.AllSpecs' // ④
        ],
        controllers: [
            'Donate' // ⑤
        ],
        launch: function() {
            Ext.create('Test.spec.AllSpecs');
        }
    });
});

```

- ① Ext JS loader needs to know the location of the testing classes...
- ② ... and about location of production code.
- ③ Create a skeleton application in the namespace of the production code to provide the execution environment.
- ④ The AllSpec class will be requesting loading of the rest of the specs. We will show code of AllSpec class in [next listing](#)

- ⑤ The skeleton application will test the controllers from the production application code

AllSpec class.

```
Ext.define('Test.spec.AllSpecs', {
    requires: [ // ❶
        'Test.spec.BasicAssumptions'
    ]
});
```

- ❶ The `requires` property includes an array of Jasmine suites. All further tests will be added to this array. Ext JS framework will be responsible for loading and instantiation all test classes.

Here is how our typical test suite will look like.

#### *Example 7-18. A BasicAssumptions class*

```
Ext.define('Test.spec.BasicAssumptions', {}, function() { // ❶
    describe("Basic Assumptions: ", function() { // ❷
        it("Ext namespace should be available loaded", function() {
            expect(Ext).toBeDefined();
        });
        it("SSC code should be loaded", function() {
            expect(SSC).toBeDefined();
        });
    });
});
```

- ❶ Wrap the Jasmine suite into an Ext JS class.
- ❷ The rest of the code is very similar to the Jasmine code sample shown earlier in this chapter.

After setting up the testing harness for Save The Child application we will suggest testing strategy for ExtJS applications. Let's start with testing the models and controllers followed by testing the views.

## Testing The Models

*SaveSickChild.org* home page displays the information about fund raising campaigns using chart and table views backed by collection of `Campaign` models. A `Campaign` model should have three properties: `title`, `description`, and `location`. The `title` property of the model should have a default value - `Default Campaign Title`. The `location` property of the model is required field.

In a spirit of TDD, let's write the **specification** what will meet the requirements described above.

*Example 7-19. CampaignModelAssumptions specification*

```
Ext.define('Test.spec.CampaignModelAssumptions', {}, function() {
    'use strict';
    beforeEach(function() {
        });

    afterEach(function() {
        Ext.data.Model.cache = []; // ①
    });

    describe('SSC.model.Campaign model', function() {
        it('exists', function() { // ②
            var model = Ext.create('SSC.model.Campaign', {});
            expect(model.$className).toEqual('SSC.model.Campaign');
        });
        it('has properties', function() { // ③
            var model = Ext.create('SSC.model.Campaign', {
                title: 'Donors meeting',
                description: 'Donors meeting agenda',
                location: 'New York City'
            });
            expect(model.get('title')).toEqual('Donors meeting');
            expect(model.get('description')).toEqual('Donors meeting agenda');
            expect(model.get('location')).toEqual('New York City');
        });
        it('property title has default values', function() { // ④
            var model = Ext.create('SSC.model.Campaign');
            expect(model.get('title')).toEqual('Default Campaign Title');
        });
        it('requires campaign location', function() { // ⑤
            var model = Ext.create('SSC.model.Campaign');
            var validationResult = model.validate();
            expect(validationResult.isValid()).toBeFalsy();
        });
    });
});
```

- ① By default, Ext.data.Model caches every model created by the application in a global in-memory array. We need to clean up the ExtJS model cache after each test run.
- ② Instantiate the Campaign model class to check that it exists.
- ③ Next we need to check if model has all required properties.
- ④ The property title has a default value.
- ⑤ Validation will fail on the empty location property.

```

Ext.define('SSC.model.Campaign', {
    extend: 'Ext.data.Model',
    fields: [
        {
            name: 'title',
            type: 'string',
            defaultValue: 'Default Campaign Title'
        },
        {
            name: 'description',
            type: 'string'
        },
        {
            name: 'location',
            type: 'string'
        }
    ],
    validations: [
        {
            field: 'location',
            type: 'presence'
        }
    ]
});

```

## Testing The Controllers

Controllers in ExtJS are classes like any others and should be tested the same way. In the next example, let's test *Donate Now* functionality. When the user clicks *Donate Now* button of the Donate panel controller's code should validate the user input and submit the data to the server. Since we are just testing controller's behavior, we're not going to submit the actual data. We'll use Jasmine spies instead.

### Example 7-20. Donate controller specification

```

Ext.define("Test.spec.DonateControllerSpec", {}, function () {
    describe("Donate controller", function () {
        beforeEach(function () {
            // controller's setup code is omitted
        });
        it('should exists', function () {           // ①
            var controller = Ext.create('SSC.controller.Donate');
            expect(controller.$className).toEqual('SSC.controller.Donate');
        });
        describe('donateNow button', function () {
            it('calls donate on DonorInfo if form is valid', function () {
                var donorInfo = Ext.create('SSC.model.DonorInfo', {});
                var donateForm = Ext.create('SSC.view.DonateForm', {});
                var controller = Ext.create('SSC.controller.Donate');
                spyOn(donorInfo, 'donate');           // ②
                spyOn(controller, 'getDonatePanel').and.callFake(function () { // ③
                    donateForm.down = function () {

```

```

        return {
            isValid: function () {
                return true;
            },
            getValues: function () {
                return {};
            }
        };
    );
    return donateForm;
});
spyOn(controller, 'newDonorInfo').and.callFake(function () { //④
    return donorInfo;
});
controller.submitDonateForm();
expect(donorInfo.donate).toHaveBeenCalled(); // ⑤
);
});
});
});
});

```

- ① First, you need to test if controller's class is available and can be instantiated.
- ② With the help of Jasmine's `spyOn()` function substitute the `DonorInfo` model's `donate()` function.
- ③ We're not interested in the view's interaction - only the contract should be tested. At this point, some methods can be substituted with the fake implementation to let the test pass. In this case, the specification tests the situation when form's valid.
- ④ Next you need to inject emulated controller dependencies. The function `donate()` was replaced by the spy.
- ⑤ Finally, you can assert if the function was called by the controller.

The function under the test looks as follows:

```

Ext.define('SSC.controller.Donate', {
    extend: 'Ext.app.Controller',
    refs: [{
        ref: 'donatePanel',
        selector: '[cls=donate-panel]'
    }],
    init: function() {
        'use strict';
        this.control({
            'button[action=donate]': {
                click: this.submitDonateForm
            }
        });
    }
});

```

```

    },
    newDonorInfo: function() {           // ①
        return Ext.create('SSC.model.DonorInfo', {});
    },
    submitDonateForm: function() {
        var form = this.getDonatePanel().down('form');
        if (form.isValid()) {           // ②
            var donorInfo = this.newDonorInfo();
            Ext.iterate(form.getValues(), function(key, value) { // ③
                donorInfo.set(key, value);
            }, this);
            donorInfo.donate();   // ④
        }
    }
});

```

- ① The factory method for creating a new instance of the `SSC.model.DonorInfo` class.
- ② If the form is valid, read data from the form fields...
- ③ ... and populate properties of corresponding object.
- ④ `DonorInfo` can be submitted by calling the `donate()` method.

## Testing The Views

UI Tests can be divided into two constituent parts: interaction tests and component tests. Interaction tests simulate real-world scenarios of application usage as if a user is using the application. It's better to delegate the interaction tests to the functional testing tools like Selenium or CasperJS.

There is another UI testing tool worth to mention especially, in the context of testing ExtJS applications - **Siesta**. Siesta allows to perform testing of the DOM and simulate user interactions. Siesta written in JavaScript and uses Siesta for unit and UI testing. There are two editions of Siesta - lite and professional.

Component Tests isolate independent and reusable pieces of your application to verify their display, behavior and contract with other components (see [testing of the controllers](#)). Let's see how we can do that. Consider [following example](#)

### *Example 7-21. Example Title Needed Here*

```

Ext.define('Test.spec.ViewsAssumptions', {}, function () {
    function prepareDOM(obj) {           // ①
        Ext.DomHelper.append(Ext.getBody(), obj);
    }
    describe('DonateForm ', function () {
        var donateForm = null; // ②
    })
});

```

```

beforeEach(function () {
    prepareDOM({tag: 'div', id: 'test-donate'}); // ③
    donateForm = Ext.create('SSC.view.DonateForm', { // ④
        renderTo: 'test-donate'
    });
});
afterEach(function () {
    donateForm.destroy(); // ⑤
    donateForm = null;
});
it('should have donateform xtype', function () {
    expect(donateForm.isXType('donateform')).toEqual(true); // ⑥
});
});
});

```

- ① A helper function for fixture DOM elements creation.
- ② A reusable scoped variable.
- ③ Create fixture test div.
- ④ Create a fresh form for every test to avoid test pollution.
- ⑤ Destroy the form after every test so we don't pollute the environment.
- ⑥ In this test, you need to make sure that the `DonateForm` component has `donate form` xtype.

## Setting up the IDE for TDD

In this section we will setup WebStorm to use the described above tools inside this IDE. We will show how to integrate Grunt tool with WebStorm to run grunt tasks from there.

### Integrate the Grunt Tool with WebStorm

Let's start with the Grunt setup. Currently, there is no native support of the Grunt tool in WebStorm IDE. Since Grunt is a command line tool, you can use a general launching feature of the WebStorm IDE and configure it as an *External Tool*. Open the WebStorm preferences and navigate to *External Tools* section to get access to the external tools configuration as in [Figure 7-11](#).

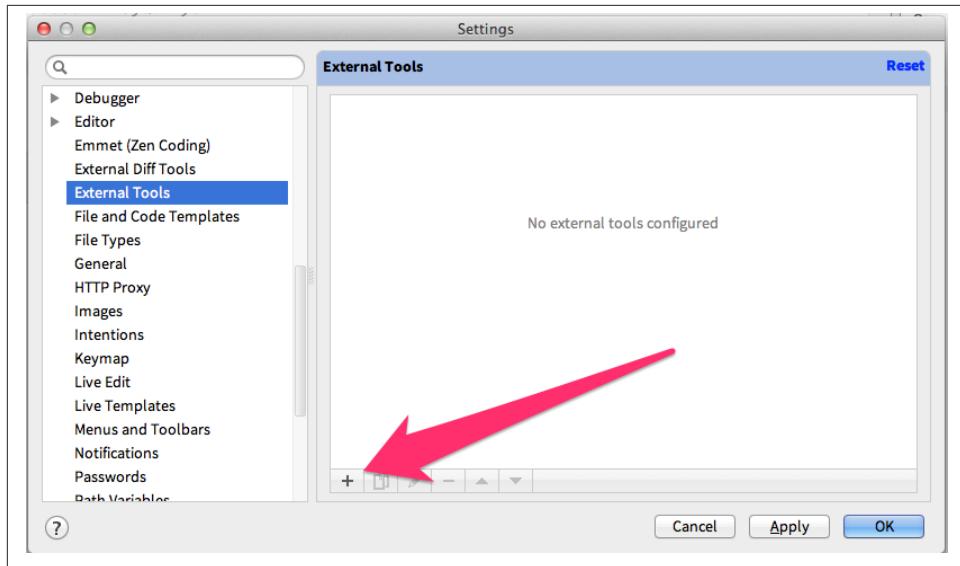


Figure 7-11. External Tools configuration window in WebStorm

Click the + button to create new External Tool configuration.

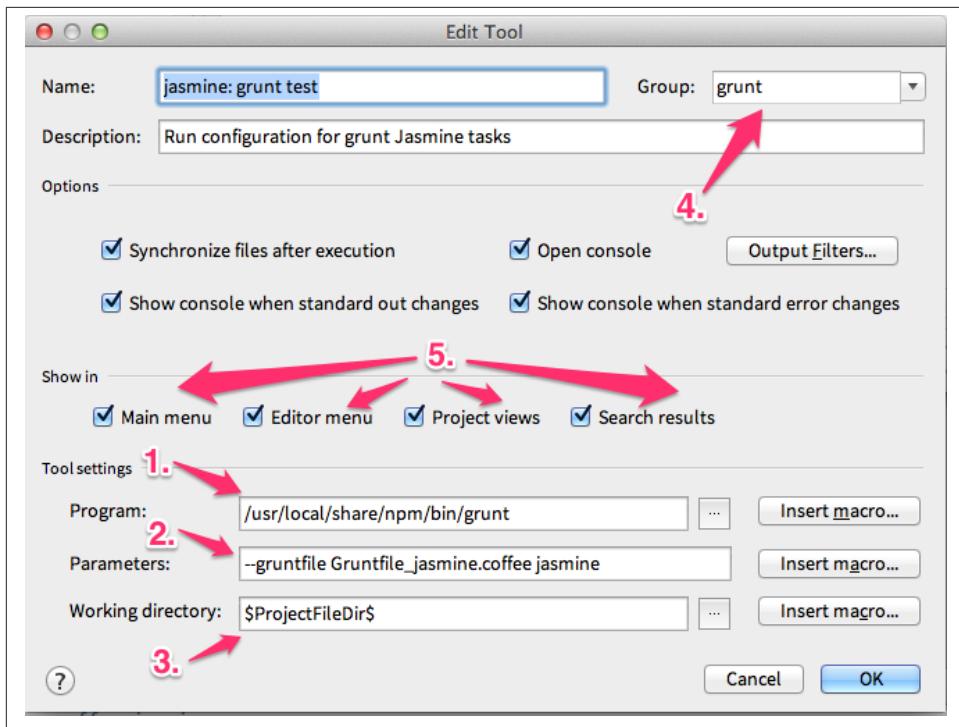


Figure 7-12. External Tool configuration

1. You need to specify the full path to the application executable.
2. Some tools require command line parameters. In this example, we explicitly specify the task runner configuration file (with the `--gruntfile` command line option) and the task to be executed.
3. Also you need to specify the *Working Directory* to run the Grunt tool. In our case, grunt configuration file is located in the root of our project. WebStorm allows to use macros to avoid hard-coded paths. Most likely, you don't want to setup external tools for each new project, just create a universal setup. In our example we use the `$ProjectFileDir$` macro which will be resolved as current WebStorm project folder root.
4. WebStorm allows you to organize related tasks into logical groups.
5. You can configure how to access the external tool launcher.

When all of the above steps are complete you can find the launcher under *Tools* menu as well under *Main menu* *Editor menu*, *Project views* and etc.

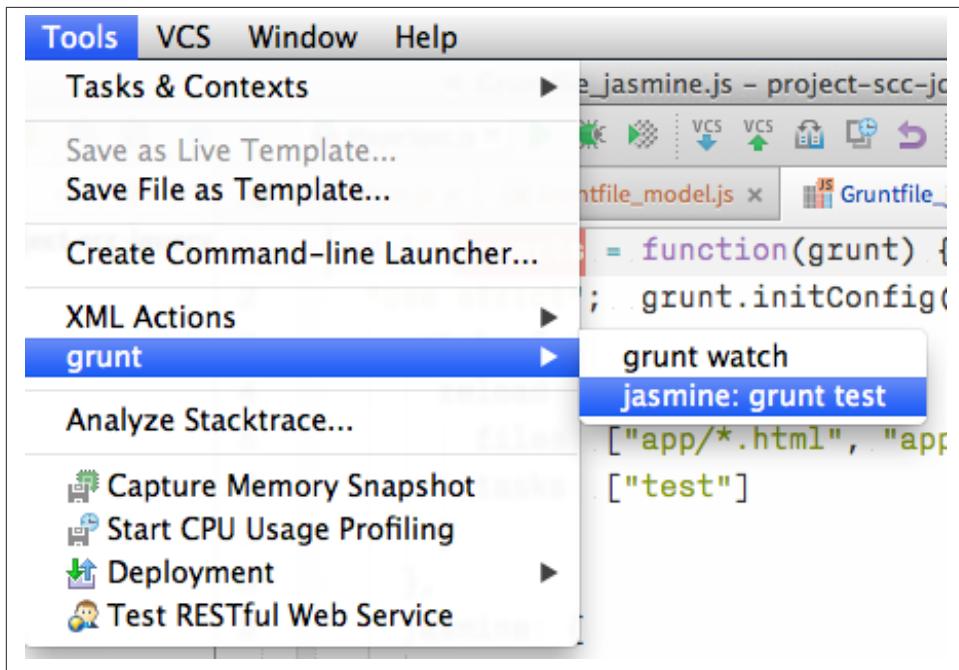


Figure 7-13. Grunt launcher available under Tools/grunt menu

Unit tests are really important as a mean to get a quick feedback from your code. You can work more efficient if you manage to minimize context switching during your coding flow. Also, you don't want to waste time digging through the menu items of your IDE, so assigning a keyboard shortcut for launching external tool is a good idea.

Let's assign a keyboard shortcut for our newly configured external tool launcher. Go to the *Keymap* section in WebStorm Preferences. Use the filter to find our created launcher `jasmine: grunt test`. Specify either the Keyboard or the Mouse shortcut by double clicking on the appropriate list item.

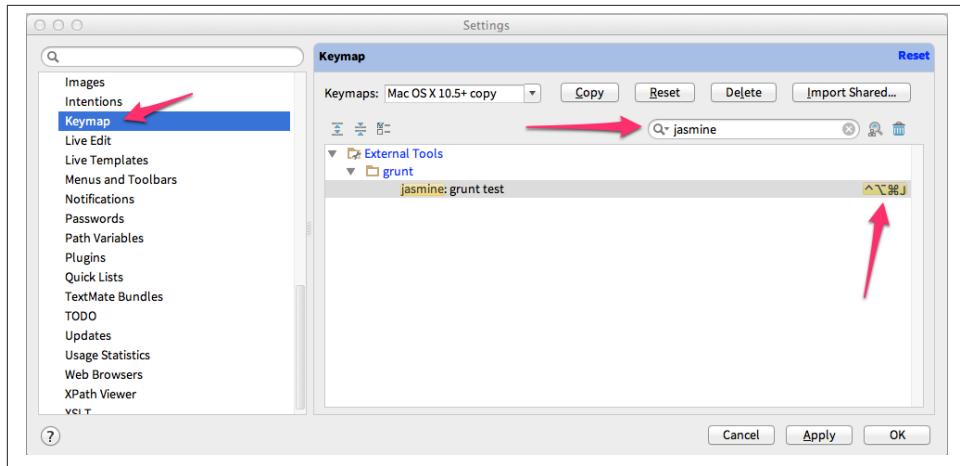


Figure 7-14. Setup keyboard shortcut for grunt launcher

By pressing a combination of keys specified in the previous screen, you will be able to launch the grunt with Jasmine tests with one click of a button(s). WebStorm will redirect all the output from the grunt tool into its Run window.

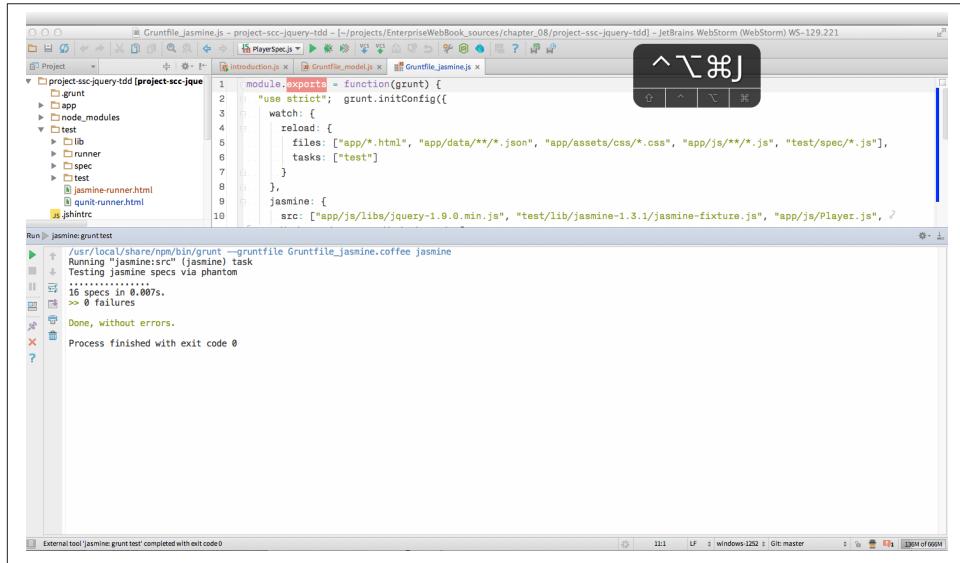


Figure 7-15. Grunt output in WebStorm

## Summary

Testing is one of the most important processes of software development. Well organized testing helps keeping the code in a good and working state. It's especially important in interpreted languages like JavaScript where there is no compiler to provide a helping hand to find lots of errors on very early stages.

In this situation, static code analysis tools, like JSHint (discussed in «Selected Productivity Tools for Enterprise Developers» chapter), could become very handy in helping with identifying typos and enforcing best practices accepted by the JavaScript community.

In enterprise projects developed with compiled languages people often debate if test-driven development is really beneficial. With JavaScript it's non-debatable unless you have unlimited time and budget and are ready to live with unmaintainable JavaScript.

The enterprises that have adopted test-driven development (as well as behavior-driven development) routines make the application development process safer by including test scripts in the continuous integration build process.

Automating unit tests reduces the number of bugs and decreases the amount of time developers need to spend manually testing their code. If automatically launched test scripts (unit, integration, functional, and load testing) don't reveal any issues, you can rest assured that the latest code changes did not break the application logic, and that the application performs according to SLA.

## References

[fowler] Martin Fowler. *Refactoring. Improving the Design of Existing Code* 2002 p.212



# Upgrading HTTP To WebSocket

This chapter is about upgrading from the HTTP protocol to a more responsive HTML5 WebSocket. It starts with a brief overview of the existing legacy Web networking, and then you'll learn why and how to use WebSockets.

We're going to show that WebSocket protocol has literally no overhead compared to HTTP. You might be consider to use WebSocket for developing the following types of applications:

- Live trading/auctions/sports notifications.
- Live collaborative writing.
- Controlling medical equipment over the web.
- Chat applications.
- Multi-player online games.
- Real-time updates in social streams.

For the next version of Save The Child application we're going to use WebSocket to implement an online auction communication layer. The goal is to let individuals and businesses purchase hand-made crafts and arts made by children. All proceeds will go to help The Children.

The goal is to let you see the advantages of changing the way of client-server communications on the Web. You'll clearly see the advantages of WebSocket over a regular HTTP by monitoring the network traffic with such tools as Wireshark and Google Chrome Developer Tools.

All the server-side functionality supporting this chapter is written in Java, using Java API for WebSocket [reference implementation](#), which is a part of Java EE 7 specification. We are using the [latest release of the Glassfish Application Server](#). If you don't know Java, just treat this server-side setup as a service that supports WebSocket protocol. For

Java developers interested in diving into the server-side, we'll provide the source code and brief comments as a part of the code samples that come with this book.

We'll show and compare the server-side data push done with Server-Sent Events and WebSocket. Also you'll see a brief overview of chosen frameworks like Portal and Atmosphere that can streamline your WebSocket application development.

## Near Real Time Applications With HTTP

The HTTP protocol is the lingua franca of today's Web applications, where client-server communications are based on the request-response paradigm. On the low level, Web browsers establish a TCP/IP connection for each HTTP session. Currently there are 3 basic options that developers use for the browser-server communication: polling, long polling, and streaming. These options are hacks on the top of a half-duplex (a one way street) HTTP protocol to simulate real-time behavior. (By *real-time* we mean the ability to react to some event as it happens. Lets discuss each of them).

### Polling

With *polling*, your client code sends requests to the server after based on some pre-configured interval (e.g. using JavaScript `setInterval()` function). Some of the server's responses will be empty, if the requested data is not ready yet as illustrated in [Figure 8-1](#). For example, if you're running an online auctions and sends the request to see the updated bids, you won't receive any data back unless someone placed a new bid.

Visualize a child seating on back seat of your car and asking every minute, "Have we arrived yet?". And you're politely replying, "Not just yet" - this is similar to an empty server response. There is no valuable payload for this kid, but she's still receiving some "metadata". HTTP polling may result in receiving verbose HTTP response headers bearing no data load, let alone distracting the driver (think the server) from performing other responsibilities.

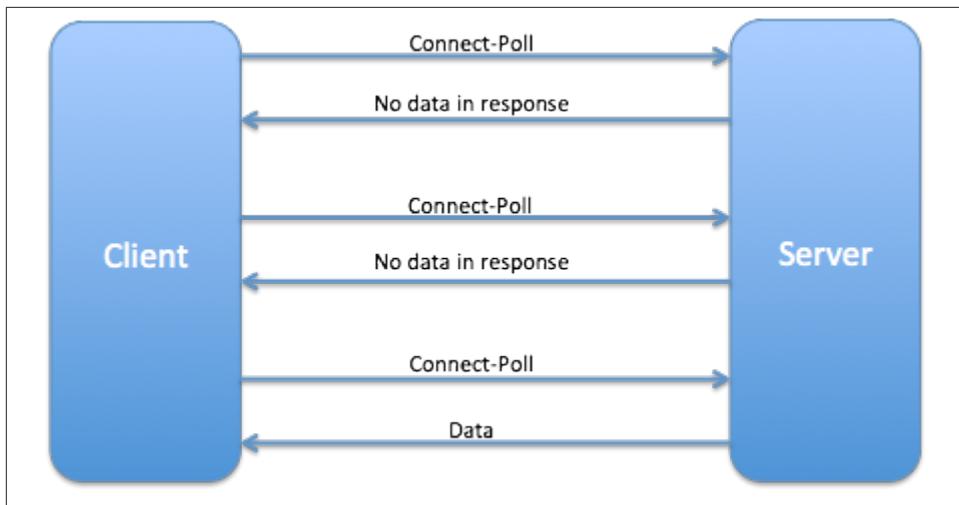


Figure 8-1. Polling

## Long Polling

*Long polling* (see [Figure 8-2](#)) starts similarly to polling: the client sends the HTTP request to the server. But in this case instead of sending an empty response back, the server waits till the data for the client becomes available. If the requested information is not available within the specified time interval, the server sends an empty response to the client, closes, and re-establishes the connection.

We'll give you one more analogy to compare polling and long polling. Imagine a party at the top floor of a building equipped with a smart elevator that goes up every minute and opens the door just in case if one of the guests wants to go down to smoke a cigarette. If no one enters the elevator, it goes to the ground level and in 60 seconds goes up again. This is the polling scenario. But if this elevator would go up, and waited till someone would actually decide to go down (or got tired of waiting), then we could call it a long polling mode.

From the HTTP specification perspective described trick is legitimate: the long polling mode may seem as if we deal with the slow-responding server. That is why this technique also referred as *Hanging GET*. If you see an online auction that automatically modifies the prices as people bid on items it looks as if the server pushes the data to you. But the chances are, this functionality was implemented using long polling, which is not a real server-side data push, but its emulation.

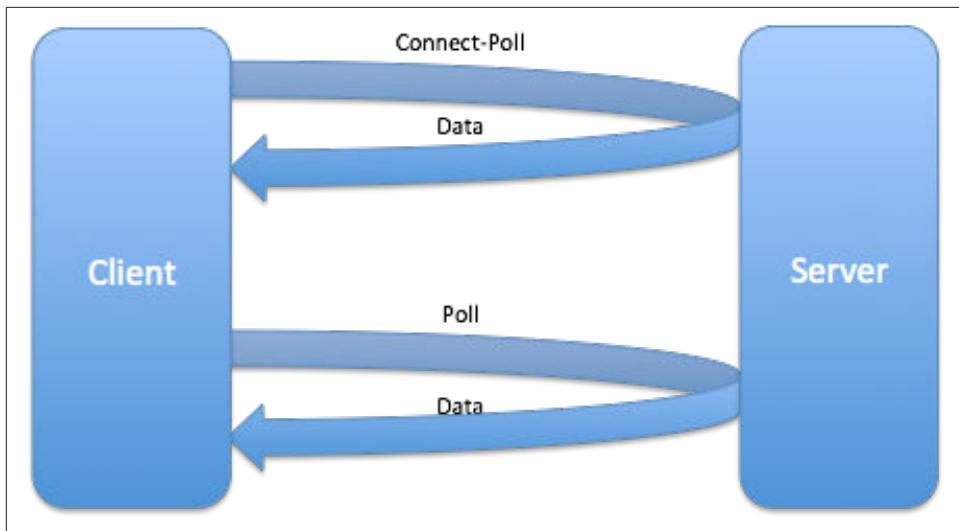


Figure 8-2. Long Polling

## HTTP Streaming

The client sends the request for data. As soon as the server gets the data ready, it starts streaming (adding more and more data to the response object) without closing the connections. The server pushes the data to the client pretending that the response never ends. For example, requesting a video from Youtube.com results in streaming data (frame after frame) without closing the HTTP connection.

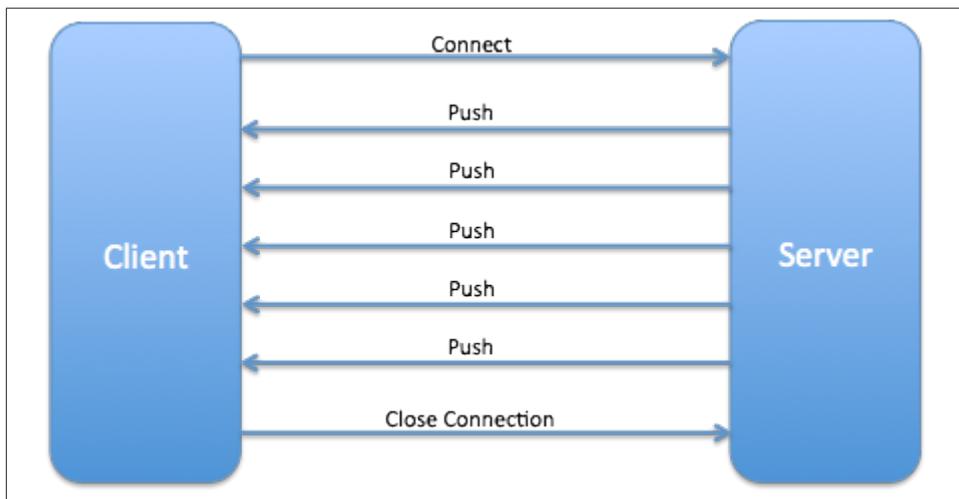


Figure 8-3. HTTP Streaming

Polling and streaming can be used as a fall-back for legacy browsers that don't support the HTML5 APIs *Server-Sent Events* and *WebSocket*.

## Server-Sent Events

Before diving into the WebSocket protocol lets get familiar with the standardized way of implementing **Server-Sent Events**. W3C has published an API for Web browsers to allow them to subscribe to the events sent by the server. All modern browsers support the `EventSource` object, which can handle events arriving in the form of DOM events. This is not a request-response paradigm, but rather a unidirectional data push - from server to browser. The following code snippet shows how a Web browser can subscribe and listen to server-sent events.

```
var myEventSource = (function() {
  'use strict';
  var eventSource;
  if ( !! window.EventSource) {
    eventSource =
      new EventSource('http://localhost:8080/donate_web/api/donations/events'); // ❶
  } else {
    // notify user that her browser doesn't support SSE
  }

  eventSource.addEventListener('open', function() { // ❷
    // Connection was opened.
  }, false);

  eventSource.addEventListener('create', function() { // ❸
    // do something with data
  }, false);

  eventSource.addEventListener('update', function() { // ❹
    // do something with data
  }, false);

  eventSource.addEventListener('error', function(e) {
    if (e.readyState === EventSource.CLOSED) {
      // Connection was closed.
    }
  }, false);

  return eventSource;
})();
```

- ❶ Create a new `EventSource` object. At this point the browser will send the GET request to the specified server-side endpoint to register itself on the server.
- ❷ Add handlers for the `open` and `error` events.
- ❸ Handle messages in `create` events by processing of the `e.data` content.

- ④ Handle messages in update events by processing of the e.data content.

The above samples create listeners to subscribe specifically `create` and `update` events, but if you'd like to subscribe to any events, you could have used the following syntax:

```
eventSource.onmessage(function(e){  
    // process the content of e.data here  
});
```

SSE is a good technique for the use cases when the client doesn't need to send the data to the server. A good illustration of such server might be Facebook's New Feed Page. A server can automatically update the client with new data without client's request.

In the above example the server sends two types of custom events `create` and `update` to notify subscribed clients about updated donation data so the active clients can monitor the fund-raising process. You can create as many custom events as needed by the application.

The server sends events as text messages that start with `data:` and end with a pair of new line characters, for example:

```
'data: {"price": "123.45"}\n\n'
```

SSE is still HTTP-based, and it requires server's support of the combination of HTTP 1.1 keep-alive connections and the `text/event-stream` content type in HTTP response. The overhead is minimal - instead of hundreds of bytes in request and response headers, the server sends only the responses when the data has changed.

## Introducing WebSocket API

Reducing kilobytes of data to 2 bytes is more than “a little more byte efficient”, and reducing latency from 150ms (TCP round trip to set up the connection plus a packet for the message) to 50ms (just the packet for the message) is far more than marginal. In fact, these two factors alone are enough to make WebSocket seriously interesting to Google.

source: <http://ietf.org/mail-archive/web/hybi/current/msg00784.html>

— HTML spec editor at Google  
*Ian Hickson*

WebSocket is a bi-directional full-duplex frame-based protocol. According to [RFC 6455](#) - the Internet Engineering Task Force (IETF) standard document - the goal of WebSocket technology is to provide a mechanism for Web applications that need two-way communications with servers. This technology doesn't rely on HTTP hacks or on opening multiple connections using `XMLHttpRequest` or `<iframe>` and long polling. The idea behind WebSocket is not overly complicated:

- Establish a socket connection between the client and the server using HTTP for the initial handshake.

- Switch the communication protocol from HTTP to a socket-based protocol.
- Send messages in both directions simultaneously (a.k.a. full duplex mode).
- Send messages independently. This is not a request-response model as both the server and the client can initiate the data transmission which enables the real server-side push.
- Both the server and the client can initiate disconnects too.

You will get a better understanding of each of these statements after reading the section [WebSocket API](#) later in this chapter.

The WebSocket protocol defines two new [URI schemes](#) ws and wss for unencrypted and encrypted connections respectively. The ws (WebSocket) URI scheme is similar to HTTP URI scheme and identifies that a WebSocket connection will be established using TCP/IP protocol without encryption. The 'wss' (WebSocket Secure) URI scheme identifies that the traffic over that connection will be protected via Transport Layer Security (TLS). The TLS connection provides such benefits over TCP connection as data confidentiality, integrity, and the endpoint authentication. Apart from the scheme name, WebSocket URI schemes use generic URI syntax.

## WebSocket Interface

The W3C expert group uses [Interface Description Language](#) to describe what the WebSocket interface should look like. This is how it was defined:

```
[Constructor(DOMString url, optional (DOMString or DOMString[]) protocols)] //❶
interface WebSocket : EventTarget {
  readonly attribute DOMString url;

  const unsigned short CONNECTING = 0;           //❷
  const unsigned short OPEN = 1;
  const unsigned short CLOSING = 2;
  const unsigned short CLOSED = 3;
  readonly attribute unsigned short readyState;
  readonly attribute unsigned long bufferedAmount;

  // networking
  [TreatNonCallableAsNull] attribute Function? onopen;      //❸
  [TreatNonCallableAsNull] attribute Function? onerror;
  [TreatNonCallableAsNull] attribute Function? onclose;
  readonly attribute DOMString extensions;
  readonly attribute DOMString protocol;                  //❹
  void close([Clamp] optional unsigned short code, optional DOMString reason);

  // messaging
  [TreatNonCallableAsNull] attribute Function? onmessage;
    attribute DOMString binaryType;
  void send(DOMString data);                         //❺
```

```
    void send(ArrayBufferView data);
    void send(Blob data);
};
```

- ❶ The constructor requires an endpoint URI and optional sub-protocols names. A sub-protocol is an application-level protocol layered over the WebSocket Protocol. The client-side application can explicitly indicate which sub-protocols are acceptable for the conversation between the client and server. That string will be send to the server with the initial handshake as `Sec-WebSocket-Protocol` GET request header field. If the server supports one of the requested protocols it selects at most one of acceptable protocols and echoes that value in the same header parameter `Sec-WebSocket-Protocol` in the handshake's response. Thereby the server indicates that it has selected that protocol. It could be a custom protocol or one of standard application level protocols (see “[Save The Child Auction Protocol](#)” on page 338). For example, it's possible to transfer the SOAP or XMPP messages over the WebSocket connection. We'll discuss the handshake in the “[WebSocket Handshake](#)” on page 309 section.
- ❷ At any given time the WebSocket can be in one of four states.
- ❸ These are the callback functions of WebSocket object that will be invoked by the browser once the appropriate network event is dispatched.
- ❹ This property contains the name of the sub-protocol used for the conversation. After the successful handshake this property populated by the browser with value from servers response parameter `Sec-WebSocket-Protocol` as described in <1>.
- ❺ The WebSocket object can send text or binary data to the server using one of the overloaded `send()` methods.

## The Client-Side API

After introducing the WebSocket interface let's see the code example illustrating how the client's JavaScript can use it.

```
var ws;
(function(ws) {
    "use strict";
    if (window.WebSocket) { // ❶
        console.log("WebSocket object is supported in your browser");
        ws = new WebSocket("ws://www.websocket.org/echo"); // ❷
        ws.onopen = function() {
            console.log("onopen");
        }; // ❸
        ws.onmessage = function(e) {
            console.log("echo from server : " + e.data); // ❹
        };
        ws.onclose = function() { // ❺
    }
})()
```

```

        console.log("onclose");
    };
    ws.onerror = function() {
        console.log("onerror"); // ⑥
    };
}

} else {
    console.log("WebSocket object is not supported in your browser");
}
})(ws);

```

- ❶ Not all Web browsers support WebSocket natively as of yet. Check if the `WebSocket` object is supported by the user's browser.
- ❷ Instantiate the new `WebSocket` object with passing an endpoint URI as constructor parameter.
- ❸ Set the event handlers for `open`, `message`, `close` events.
- ❹ The `MessageEvent` is dispatched when the data is received from the server. This message will be delivered to the function assigned to the `WebSocket` object's `onmessage` property. The `e.data` property of the message event will contain the received message.
- ❺ Handle closing connection (more details in “[Closing The Connection](#)” on page [314](#) section).
- ❻ Handle errors.

## WebSocket Handshake

Any network communications that use WebSocket protocol start with an opening “handshake”. This handshake upgrades the connection from HTTP to the WebSocket protocol. It's an upgrade of HTTP protocol to a message-based communications. (We will discuss messages (a.k.a. frames) later in this chapter)

Why upgrade from HTTP instead of starting with the TCP as a protocol in the first place? The reason is that the WebSocket operates on the same ports (80 and 443) as HTTP and HTTPS do. It's an important advantage that the browser's requests are routed through the same ports because arbitrary socket connections may not be allowed by the enterprise firewalls for security reasons. Also, many corporate networks only allow certain outgoing ports. And HTTP/HTTPS ports are usually included in so called *white lists*.



You learn more about TCP in the chapter “[Building Blocks of TCP](#)” and more about HTTP in Part III of [\*High Performance Browser Networking\*](#).

The protocol upgrade is initiated by the client request, which also transmits a special key with the upgrade request. The server processes this request and sends back a confirmation for the upgrade. This ensures that a WebSocket connection can be established only with an endpoint that supports WebSocket. Here is what the handshake can look like in the client's request:

```
GET HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: echo.websocket.org
Origin: http://www.websocket.org
Sec-WebSocket-Key: i9ri`Af0gSsKwUlmLjIkGA==
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: chat
```

This client sends the GET request for the protocol upgrade. The Sec-WebSocket-Key is just a set of random bytes. The server takes these bytes and appends to this key a special Global Unique Identifier (GUID) string 258EAFA5-E914-47DA-95CA-C5AB0DC85B11, then it creates the Secure Hash Algorithm SHA1 hash from it performs the base64 encoding. The resulting string of bytes needs to be used by both the server and the client, and this string won't be used by the network endpoints that do not understand the WebSocket protocol. Then this value would be copied in the Sec-WebSocket-Accept header field. The server computes the value and sends the response back confirming the protocol upgrade.

```
HTTP/1.1 101 Web Socket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Accept: Qz9Mp4/YtIjPccdpbvFEm17G8bs=
Sec-WebSocket-Protocol: chat
Access-Control-Allow-Origin: http://www.websocket.org
```

The WebSocket protocol uses the 400 Bad Request HTTP error code to signal the unsuccessful upgrade. The handshake can also include a sub-protocol request and the WebSocket version information but you can't include arbitrary other headers. We can't transmit the authorization information. There are two ways around this. You can either transmit the authorization information as the first request (e.g. unique clientId can be passed as part of the HTTP request header or HTML wrapper) or put it into the URL as a query parameter during the initial handshake. Consider the following example:

```
var clientId = "Mary1989"; // ①
ws = new WebSocket("ws://www.websocket.org/echo/" + clientId); // ②
```

- ① The client sets the clientId value (can be obtained from some LDAP server).
- ② The client connects to the WebSocket endpoint with an extra URI parameter which will be stored on server for future interactions.

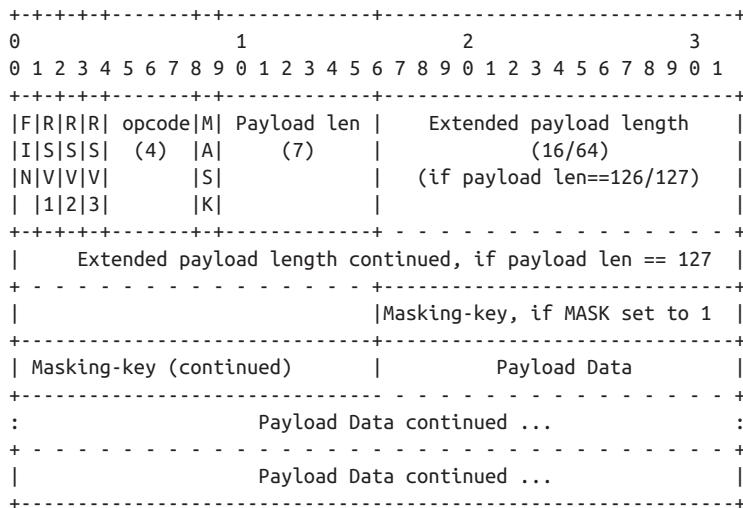
Because WebSocket protocol creates a bi-directional (socket-to-socket) connection, the server has access to the conversation session associated with such a connection. This session can be associated with clientId and be stored on server.



A client can have as many WebSocket connections with the server as needed. But servers can refuse to accept connections from hosts/IP addresses with an excessive number of existing connections or disconnect resource-hogging connections in case of high data load.

## The WebSocket Frame Anatomy

The WebSocket handshake is the first step to switching to the message framing protocol, which will be layered over TCP. In this section we're going to explore how the WebSocket data transfer works. The WebSocket is not a stream based protocol like TCP - it's message based. With TCP a program sends stream of bytes, which has to have a specific indication that the data transfer ends. The WebSocket specification simplifies this by putting a frame around every chunk of data, and the size of the frame is known. JavaScript can easily handle these frames on the client because each frame arrives packaged in the event object. But the server side has to work a little harder as it needs to wrap each piece of data into a frame before sending it to the client. A frame can look like this:



- **FIN (1 bit)**

This bit indicates if this frame is the final in the message payload. If a message has under 127 bytes it fits into a single frame and this bit will always be set.

- **RSV1, RSV2, RSV3 (1 bit each)**

These bits are reserved for future protocol changes and improvements. They must contain zeros as they are not being used at this time.

- **opcode** (4 bits)

The frame type is defined using opcode. Here are the most used opcodes:

- 0x00 This frame continues the payload.
- 0x01 This frame includes UTF-8 text data.
- 0x02 This frame includes the binary data.
- 0x08 This frame terminates the connection.
- 0x09 This frame is a Ping.
- 0xA This frame is a Pong.

- **mask** (1 bit)

This indicates if the frame is masked.



The client must mask all the frames being sent to the server. The server must close the connection upon receiving a frame that is not masked. The server must not mask any frames that it sends to the client. The client must close a connection if it detects a masked frame. In that case of such error, client or server can send Close frame with 1002 status code - protocol error. All these actions are done automatically by Web browsers and Web server that implements [WebSocket specification](#).

- **payload\_len** (7 bits, 7+16 bits, or 7+64 bits)

The length of the payload. WebSocket frames come in the following length brackets:

- 0-125 indicate the length of the payload.
- 126 means that the following two bytes indicate the length.
- 127 means the next 8 bytes indicate the length.

- **masking-key** (32 bits)

This key is used to *XOR* the payload with.

- **payload data**

This indicates the actual data. The length of block is defined in the **payload\_len** field.

## The Heartbeats

Properly design distributed application has to have a way to ensure that each tier of the system is operational even if there is no active data exchange between the client and the server. This can be done by implementing so called *heartbeats* - a small messages that simply ask the other party, “Are you there?”. For example, proxy servers and content-filtering hardware can terminate idle connections or the server could simply go down. If a client doesn’t send any requests, say for 20 seconds, but the server went down, the client will know about it only when it does the next `send()`. Heartbeats will keep the connection alive to ensure that it won’t look idling. In the WebSocket jargon heartbeats are implemented with *ping and pong* frames. The browser sends the *ping* opcode `0x9` at any time to ask the other side to *pong* back (the opcode `0xA`).

The Web browser can ping the server when required, but a pong may be sent at server’s discretion. If the endpoint receives a ping frame before responding the the previous one, the endpoint can elect to send just one pong frame for the most recently processed ping. The ping frame may contain the application data (can be up to 125 bytes) and pong must have identical data in its message body.

There is no JavaScript API to send pings or receive pong **frames**. Pings and pongs may or may not be supported by the user’s browser. **There is also no API** to enable, configure or detect whether the browser supports pings and pongs.

## Data Frames

Since the WebSocket protocol allows the data to be fragmented into multiple frames, the first frame that transmits the data will be prepended with one of the following op-codes indicating the type of data being transmitted:

- The opcode `0x01` indicates the UTF-8 encoded text data.
- The opcode `0x02` indicates the binary data.

When your application transmits JSON over the wire the opcode is set to be `0x01`. When your code emits binary data it will be represented in a browser specific `Blob` object or an `ArrayBuffer` object and sent wrapped into a frame with the opcode `0x02`.



You must choose the type for the incoming binary data on the client using `webSocket.binaryType = "blob"` or `webSocket.binaryType = "arraybuffer"` before reading the data. It’s a good idea to check the type of the incoming data because the opcodes are not exposed to the client.

```
webSocket.onmessage = function(messageEvent) {  
    if (typeof messageEvent.data === "string"){  
        console.log("received text data from the server: " + messageEvent.data);  
    }  
}
```

```
    } else if (messageEvent.data instanceof Blob){
        console.log("Blob data received")
    }
};
```

## Closing The Connection

The connection is terminated by sending the frame with the close opcode 0x08.

There is the pattern to exchange close opcodes first and then let the server to shut down. The client is supposed to give the server some time to close the connection before attempting to do that on its own. The close event can also signal why it has terminated the connection.

A `CloseEvent` is sent to clients using `WebSocket` when the connection is closed. This is delivered to the listener indicated by the `WebSocket` object's `onclose` handler. `CloseEvent` has 3 properties - `code`, `reason`, `wasClean`.

- `code` This property represents the close code provided by the server.
- `reason` A string indicating the reason of why the server closed the connection.
- `wasClean` The property indicates if the connection was cleanly closed.

```
webSocket.onclose = function(closeEvent) {
    console.log("reason " + closeEvent.reason + "code " + closeEvent.code);
};
```

# WebSocket Frameworks

Working with the vanilla `WebSocket` API requires you to do some additional “house-keeping” coding on your own. For example, if the client’s browser doesn’t support `WebSocket` natively, you need to make sure that your code falls back to the legacy `HTTP` protocol. The good news is that there are frameworks that can help you with this task. Such frameworks lower the development time, allowing you to do more with less code. We’ve included brief reviews of two frameworks that can streamline your Web application development with `WebSocket`.

The frameworks mentioned below try to utilize the best supported transport by the current Web browser and server while sparing the developer from knowing internals of the used mechanism. The developer can concentrate on programming the application logic making calls to frameworks API when the data transfer is needed. The rest will be done by framework.

## The Portal

The [Portal](#) is a server agnostic JavaScript library. It aims to utilize a `WebSocket` protocol and provides a unified API for various transports (long polling, `HTTP` streaming, `Web-`

Socket). Currently, once you've decided to use WebSocket API for your next project, you need to remember about those users who still use old browsers like Internet Explorer 9 or older, which don't natively support WebSockets. In this case, your application should gracefully fall back to the best available networking alternative. Manually writing code to support all possible browsers and versions requires lots of time especially for testing and maintaining the code for different platforms. The `Portal` library could help as illustrated in the following brief code sample.

*Example 8-1. Simple asynchronous web application client with Portal*

```
portal.defaults.transports = ["ws", "sse", "stream", "longpoll"]; // ❶

portal.open("child-auction/auction").on{ // ❷
    connecting: function() {
        console.log("The connection has been tried by '" + this.data("transport") + "'");
    },
    open: function() { // ❸
        console.log("The connection has been opened");
    },
    close: function(reason) {
        console.log("The connection has been closed due to '" + reason + "'");
    },
    message: function(data) {
        handleIncommingData(data);
    },
    waiting: function(delay, attempts) {
        console.log("The socket will try to reconnect after " + delay + " ms");
        console.log("The total number of reconnection attempts is " + attempts);
    }
});
```

- ❶ The Portal framework supports different transports and can fall back from WebSocket connection to streaming or long polling. The server also has to support some fall-back strategy, but no additional code required on the client side.
- ❷ Connecting to the WebSocket endpoint.
- ❸ The Portal API is event-based similarly to W3C WebSocket API.

The Portal framework generalizes the client-side programming. With defining an array of transports, you don't have to worry about how to handle message sent by server with different transport. The Portal doesn't depend on any JavaScript library.

## Atmosphere

A Web application that has to be deployed on several different servers (e.g. WebSphere, JBoss, WebLogic) may need to support different WebSocket APIs. At the time of this writing, there is a plethora of different implementations of the server-side libraries sup-

porting WebSockets, and each of them uses their own proprietary APIs. The Java EE 7 specification intends to change the situation. But Atmosphere is a framework that allows you to write portable Web applications today.

**Atmosphere** is a **portable** WebSocket framework supporting Java, Groovy, and Scala. The Atmosphere Framework contains both client and server components for building Asynchronous Web Applications. The Atmosphere transparently supports WebSocket, Server Side Events, long-polling, HTTP streaming, and JSONP.

The client side component **Atmosphere.js** uses The Portal framework internally and simplifies the development of Web applications that require a fall back from the WebSocket protocol to long polling or HTTP streaming. The Atmosphere Framework hides the complexity of the asynchronous APIs, which differ from server to server and makes your application portable among them. Treat Atmosphere as a compatibility layer that allows to select best available on server transport for all major Java application servers.

The Atmosphere framework supports wide range of Java based server-side technologies via a set of **extensions and plugins**. The Atmosphere **supports** Java API for WebSocket, so you can have best of two worlds - the standard API and application portability.



WebSockets can be used not only in for the Web, but in any applications that use networking. If you're developing native iOS or OSX applications check **the SocketRocket** library developed by **Square Engineering Team**.

Square uses SocketRocket in their mobile payments application. If you're developing native Android applications and want to use WebSocket protocol goodies in Android-powered devices check **the AsyncHttpClient** framework.

## Application-level Messages Format Considerations

While WebSocket is a great solution for a real-time data transmission over the web, it has the downside too: the WebSocket specification defines only the protocol for transporting frames, but it doesn't include the application-level protocol. Developers need to invent the application-specific text or binary protocols. For example, the auction bid has to be presented in a form agreed upon by all application modules. Let's discuss our options from protocol modeling perspective.

Selecting a message format for your application's data communications is important. The most common text formats are CSV, XML, and JSON. They are easy to generate, parse, and are widely supported by many frameworks in the most development platforms. While XML and JSON allow you to represent the data in a hierarchical and easily readable by humans form, they create a lot of overhead by wrapping up each data element into additional text identifiers. Sending such additional textual information re-

quires extra bandwidth and may need additional string-to-type conversion on both the client and server's application code. Let's discuss the pros and cons of these message formats.

## CSV

CSV stands for Comma Separated Values although the delimiter can be any character, not only the comma. This depends on the parser design and implementation. Another popular type of delimiter is | - "pipe".

Pros:

- This format is very compact. The overhead of the separator symbol is minimal.
- It's simple to create and parse. The CSV message can be turned into array of values by using the standard JavaScript `String.split()`.

Cons:

- It's not suitable for storing complex data structures and hierarchies. In case of an auction application, we need transfer to client auction items' attributes for each auction. In this case we can't simply use `String.split()` and have to design and implement more complex parser.

## XML

XML nicely represents any hierachal data structures.

Pros:

- It's a human-readable format.
- Most browsers have build-in XML readers and parsers.
- XML data can be validated against XSD or DTD schema.

XML schema is very useful language feature as it defines the structure, content and semantic of XML document. Because of its human-readability the XML schema can be used by people who are not software developers and can be applied for integrating systems written in different programming languages.

Cons:

- XML is very verbose. To send a name of a customer you'd need something like this:  
`<cust_name>Mary</cust_name>`

- The XML validation on the client is a complex task. As for now, there is no platform independent solutions or the API to perform validation programmatically based on XSD or DTD.

The book *XML in a Nutshell, 3rd Edition* by Elliotte Rusty Harold and W. Scott Means is a well-written book describing the full spectrum of XML features and tools.

## JSON

As explained in Chapter 2, JSON stands for JavaScript Object Notation, and it's a way of representing structured data, which can be encoded and decoded by all Web browsers. JSON is widely accepted by the Web community as a popular way of the data serialization. As stated earlier, it's a more compact than XML way to represent the data and all modern Web Browsers understand and can parse JSON data.

## Google Protocol Buffers

Google Protocol Buffers is a language and platform-neutral extensible mechanism for structured data serialization. Once defined how you want your data to be structured, you can use special generated source code to easily write and read your structured data to and from a variety of data streams. Developers can use the same schemas across **diverse environments**.

A developer needs to specify how the serializable information has to be structured by defining the protocol buffer message types in the .proto files. Each protocol buffer message is a small logical record of information, containing a series of the name/value pairs. This protocol buffer message file is language agnostic. The protoc utility compiles proto files and produces language specific artifacts, e.g. .java, .js, etc files.

For example, you can create a protocol buffer proto file for our Save The Child to represent the information about donors.

*Example 8-2. Protocol Buffer for Donation message ( donation.proto )*

```
package savesickchild; // ①

option java_package = "org.savesickchild.web.donation"; // ②

message Donor{ // ③
    required string fullname = 1;
    required string email = 2; // ④
    required string address = 3;
    required string city = 4;
    required string state = 5;
    required int32 zip = 6;
    required string country = 7;

    message Donation{ // ⑤
```

```

    required Donor donor = 1;
    required double amount = 2;
    optional bool receipt_needed = 3;
}
}

```

- ❶ The protobuf supports packages to prevent naming conflicts among messages from different projects.
- ❷ Here we're using Java specific protobuf option to define in what package the generated code will reside.
- ❸ Start defining our custom message with the `message` keyword.
- ❹ Each message field can be `required`, `optional` or `repeated`. The `required` and `optional` modifiers are self explanatory. During serialization-deserialization process the protobuf framework will check the message for existence of fields, and if a required property is missing it will throw a runtime exception. The `repeated` modifier is used to create dynamically sized arrays.
- ❺ The protobuf supports nested messages.
- ❻ There are many standard field types available in protobuf: `string`, `int32`, `float`, `double`, and `bool`. You can also define a custom type and use it as a field type.

After creating the `donation.proto` file, you can use `protoc` compiler to generate Java classes according to this file's definitions.

```
protoc -I=. --java_out=src donation.proto # ❶
```

```
.
├── donation.proto
└── src
    └── org
        └── savesickchild
            └── web
                └── donation
                    └── Donation.java # ❷
```

- ❶ The Java code will be generated in `src` directory.
- ❷ All required code for serialization-deserialization of `Donation` message will be included in `Donation.java`. We're not going to publish the generated code here, but you can generate this code by yourself from the provided above message declaration.

Check the availability of the protobuf compiler for your preferred language at the [protobuf wiki page](#). To make yourself familiar with protobuf technology check [documentation](#) and [tutorials](#). Here some protobuf pros and cons:

Pros:

- The message is encoded into a compact and optimized binary format. You can find the details of the encoding format at [Protocol Buffers documentation website](#).
- Google supports Protocol Buffers for a wide range of programming languages (Java, C++, Python). The [developer's community](#) supports it too.
- The use of Protocol Buffers is well documented.

Cons:

- The binary format is not human readable.
- Although Protobuf is compact, especially when a lot of numeric values are transferred [by an encoding algorithm](#), the JSON is natively supported by the JavaScript and doesn't require any additional parser implementation.
- Protobuf requires the Web browser to support the binary format, but not all of them do it just yet. You can find which browser support raw binary data at <http://caniuse.com/#search=binary>.

## WebSockets and Proxies

The WebSocket protocol itself is unaware of intermediaries such as proxy servers, firewalls, content filters. The proxy servers are commonly used for content caching, security and enterprise content filtering.

HTTP always supported protocol upgrades, but many proxy servers seem to have ignored that part of the specification. Until the WebSocket came around the `Upgrade` attribute was not used. The problem with Web applications that use long-lived connection like WebSocket is that the proxy servers may choose to close streaming or idle WebSocket connections, because they appear to be trying to connect to the unresponsive HTTP server. Additionally, proxy servers may buffer unencrypted HTTP responses assuming that the browser needs to receive the HTTP response in its entirety.

If you want to get more details on how a WebSocket-enabled application has to deal with proxies there is comprehensive research paper by Google's Peter Lubbers [WebSocket and proxy servers](#).



The author of this book use **NGINX**, a hugely popular load balancer and proxy and HTTP server to serve static resources (e.g. images and text files), balance the load between Java servers, and for SSL offloading (turning Web browser's HTTPS requests into HTTP). NGINX uses a very small number threads to support thousands concurrent users as opposed to traditional Web servers that use one worker thread per connection. Recently NGINX started supporting WebSockets protocol.

## Adding an Auction to Save The Child

We gave you just enough of a theory to whet your appetite to start implementing WebSocket in our Save The Child application. The goal is to create an auction where people can bid and purchase various goods so the proceeds would go to The Children. Auctions require real-time communications - everyone interested in the particular auction item must be immediately notified if she was overbid or won. So we'll use WebSocket as a means for bidding and notifications of the changes in the auction.

To start the auction, the user has to select the *Auction* option under the menu *Way To Give* (see [Figure 8-4](#)). We realize that only a small number of users will decide to participate in the auction, which from the architectural point of view means that the code supporting the auction should be loaded *on demand* only if the user chooses to visit the auction. This why we need to write this code as a loadable module, and the reader will get a practical example of how a Web application can be modularized.

In this chapter we continue to use RequireJS (see Chapter 7) as a framework for modularization. Using RequireJS, we're going to lazy load some modules if and only if they get requested by the user.

This book is about development of the user interface and client side of the Web applications hence we're not going to cover all the details of server side implementation, but will make our server side code available for download. We'll keep our server up and running so you can test the UI by visiting <http://savesickchild.org:8080/websocket-auction/>, but our main goal in this section is to show you how you can exchange the auction data with the server and process them on the client side using WebSocket. We'll use the Java application server GlassFish 4, which is a reference implementation of Java EE 7 specification.



Authors of this book are Java developers and we have recorded a screencast (see the readme.asciidoc at <https://github.com/Farata/EnterpriseWebBook> for the URL) highlighting WebSocket server API. If you are not a Java developer, you may want to learn on your own which WebSocket servers exist for your favorite programming language or platform.

In Chapter 7 you had a chance to see how a Web application can be sliced into several modules using Require.js framework. We'll take this project as a base and will create a new one: project-16-websocket-auction adding to it the new modules supporting the auction.

*Example 8-3. Way To Give module (js/modules/way-to-give.js)*

```
define([], function() {
    var WayToGive;
    console.log("way-to-give module is loaded");
    WayToGive = function() {
        return {
            render: function() { // ①
                // rendering code is omitted
                console.log("way-to-give module is rendered");
                rendered = true;
                return
            },
            startAuction: function(){ // ②
                },
                rendered: false // ③
            };
    };
    return WayToGive;
});
```

- ① This function will lazy load the auction application content and render it to the top main section of the Web page.
- ② The function `startAuction()` will start the auction.
- ③ The module stores the rendering state in the property `rendered`.

Once the application starts, the require.js loads only essential modules - login and donation [Figure 8-4](#).

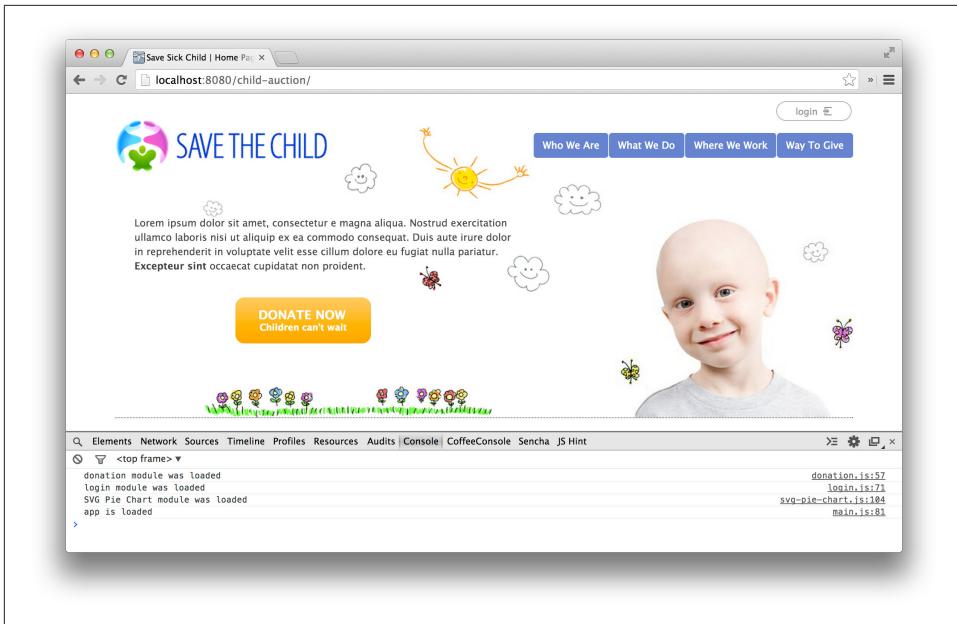


Figure 8-4. Initially only two modules are loaded

In Google Chrome Developer Tools console you can see that login and donation modules are reporting about successful loading. [Figure 8-5](#) confirms that these modules perform fine - clicking on the button Donate reveals the form and clicking on the Login button makes the id/password fields visible.

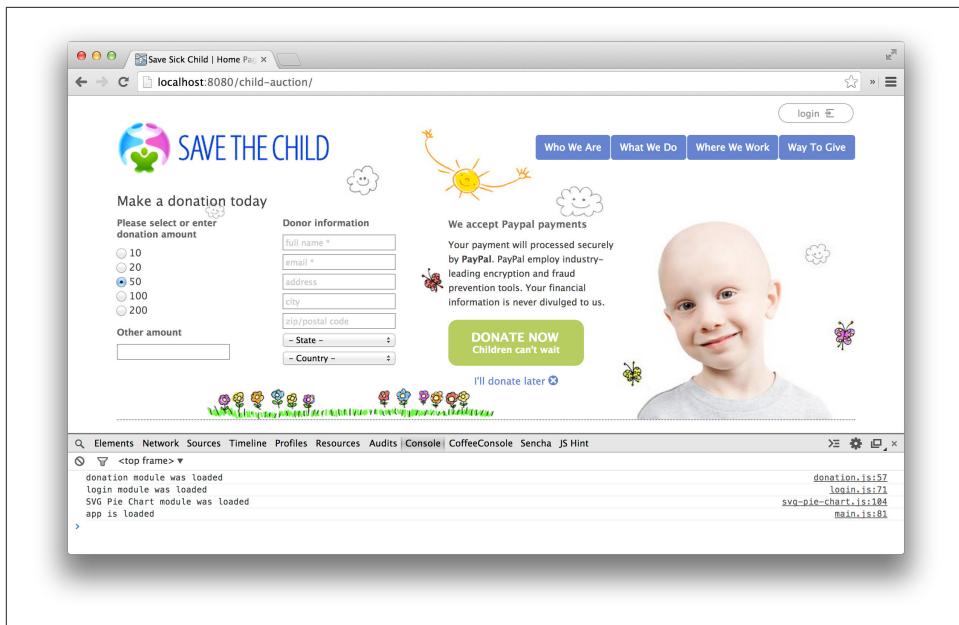


Figure 8-5. Two modules are loaded during the Save The Child application startup

Now click the Way To Give menu item and keep an eye on the Developer Tools console **Figure 8-6**. You will see the way-to-give module reporting about its loading and rendering.

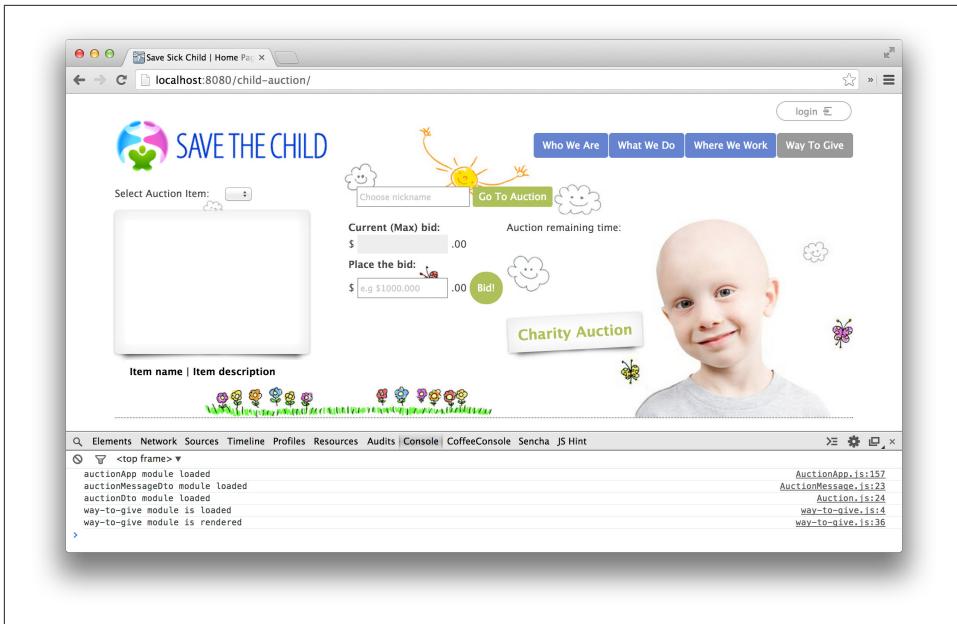


Figure 8-6. The auction controls are loaded and rendered

Once the user clicks the menu Way to Give the framework require.js has to load code of the WebSocket-based auction module. Here is the code snippet from the JavaScript file `app.js` - the entry point of our Save The Child application. This is how it loads the module *on demand* (see Chapter 7 for the Require.js refresher).

```
require([], function() { // ①
  'use strict';
  return (function() {
    var lazyLoadingEventHandlerFactory, wayToGiveHandleClick, wayToGiveModule, way_to_give;
    way_to_give = document.getElementById('way-to-give');

    wayToGiveModule = null; // ②

    lazyLoadingEventHandlerFactory = function(module, modulePath) {
      var clickEventHandler;
      clickEventHandler = function(event) {
        console.log(event.target);
        if (module === 'loading') { // ③
          return;
        }
        if (module !== null) {
          return module.startAuction(); // ④
        } else {
          module = 'loading'; // ⑤
          return require([modulePath], function(ModuleObject) { // ⑥
            module = new ModuleObject();
          });
        }
      };
      clickEventHandler();
    };
  });
});
```

```

        return module.render(); // ⑦
    });
}
return clickEventHandler;
};
wayToGiveHandleClick = lazyLoadingEventHandlerFactory(wayToGiveModule, 'modules/way-to-give');

way_to_give.addEventListener('click', wayToGiveHandleClick, false); // ⑧
})();
);

```

- ➊ This anonymous function will be lazy-loaded only if the user clicks on Way To Give menu.
- ➋ The variable `wayToGiveModule` will have a value of `null` until loaded.
- ➌ If the user keeps clicking on the menu while the `way-to-give` module is still being loaded, simply ignore these clicks.
- ➍ If the module has been loaded and UI has been rendered start Auction application.
- ➎ Set an intermediary value to the `way-to-give` module so that subsequent requests don't try to launch the module more than once.
- ➏ Load module asynchronously and instantiate it.
- ➐ Render UI component to the screen for first time.
- ➑ Register the click event listener for the Way To Give menu.

After the UI elements have rendered the client can connect to the WebSocket server and request the list of all available auction items.

```

if (window.WebSocket) {
    webSocket = new WebSocket("ws://localhost:8080/child-auction/auction");
    webSocket.onopen = function() {
        console.log("connection open..."); //①
        getAuctionsList();
    };
    webSocket.onclose = function(closeEvent) {
        // notify user that connection was closed
        console.log("close code " + closeEvent.code);
    };
    webSocket.onmessage = function(messageEvent) {
        console.log("data from server: " + messageEvent.data);
        if (typeof messageEvent.data === "string") {
            handleMessage(messageEvent.data);
        }
    };
    webSocket.onerror = function() {
        // notify user about connection error
        console.log("websocket error");
    };
}

```

```
        };
    }
}
```

- ❶ After establishing the connection the code requests the list of available auctions. We'll see details of `getAuctionsList()` method in next snipped.

```
var getAuctionsList = function() {
  'use strict';
  var auctionListMessage = {
    type: 'AUCTIONS_LIST',
    data: 'gime',
    auctionId: '-1'
  }; // ❶
  if (webSocket.readyState === 1) { // ❷
    webSocket.send(JSON.stringify(auctionListMessage));
  } else {
    console.log('offline');
  }
};
```

- ❶ Forming the request message. The details of the message format could be found in “Save The Child Auction Protocol” on page 338 section.
- ❷ Checking the WebSocket object state. If WebSocket is open (`readyState==1`) the application can send a message. If not, this code just simply logs the “offline” message on the console. In the real world you should always display this message on the user’s UI. Also, if your users work on unstable networks such as cellular or 3G you definitely don’t want to lose any bits of data. It’s a good idea to use the local storage API (see Chapter 1) to persist the data locally until the application gets back online and resubmit the data.

The user can select the auction lot from the combo box and see its images. [Figure 8-7](#) shows what’s displayed on the console, while [Figure 8-8](#) and [Figure 8-9](#) show the content of the Network tab for both images.

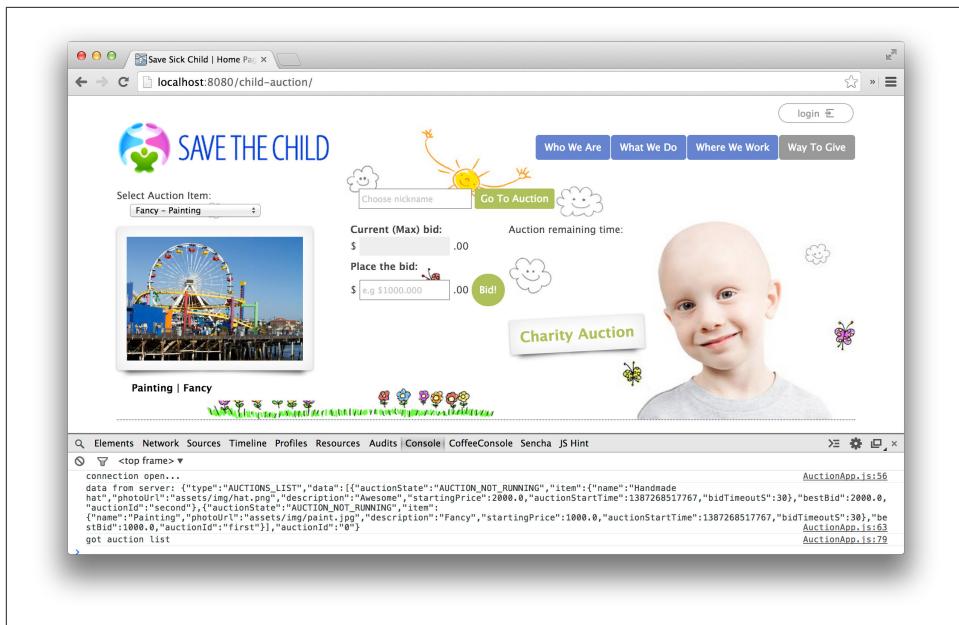


Figure 8-7. The console logs incoming message contains list of auction items.

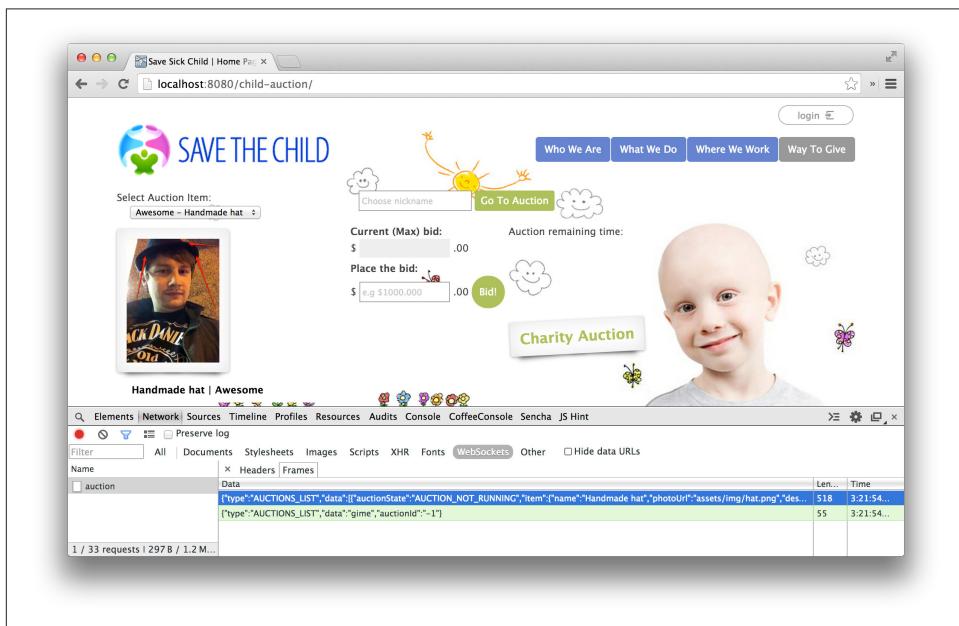


Figure 8-8. Using Network feature of Dev Tools we can monitor WebSocket frames.

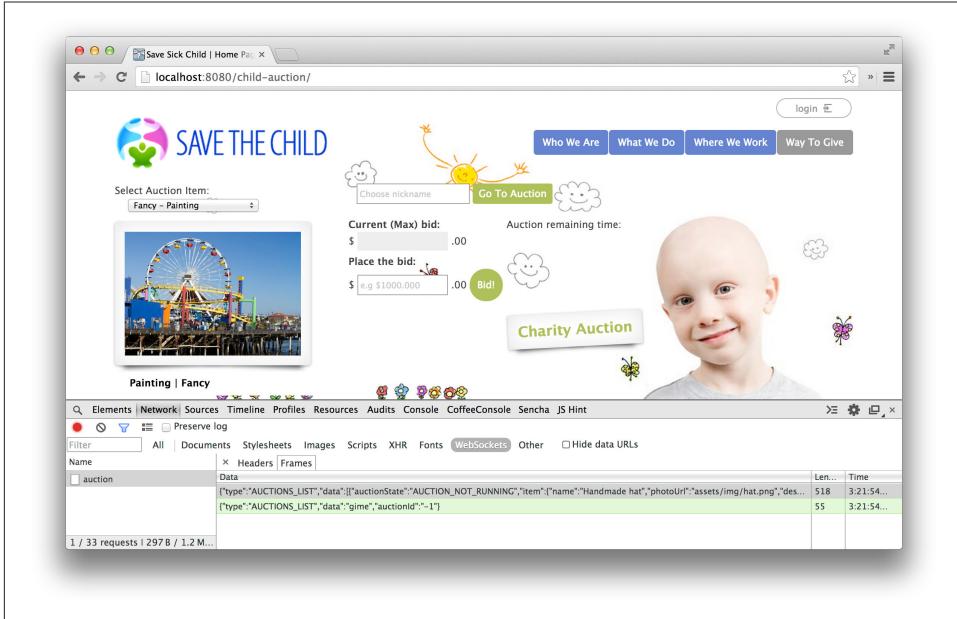


Figure 8-9. The buyer can choose another item to bid for.

## Monitor the WebSocket traffic with Chrome Developers Tools

Let's review the practical use of the theory described in the “[WebSocket Handshake](#)” on [page 309](#) section earlier. With the help of Chrome Developer Tools you can monitor the information about the initial handshake [Figure 8-10](#). Monitoring WebSocket traffic in Chrome Developers Tools is, in some ways, not that different from monitoring HTTP requests. This can be viewed in the Network tab after selecting the path of the WebSocket endpoint on the left panel.

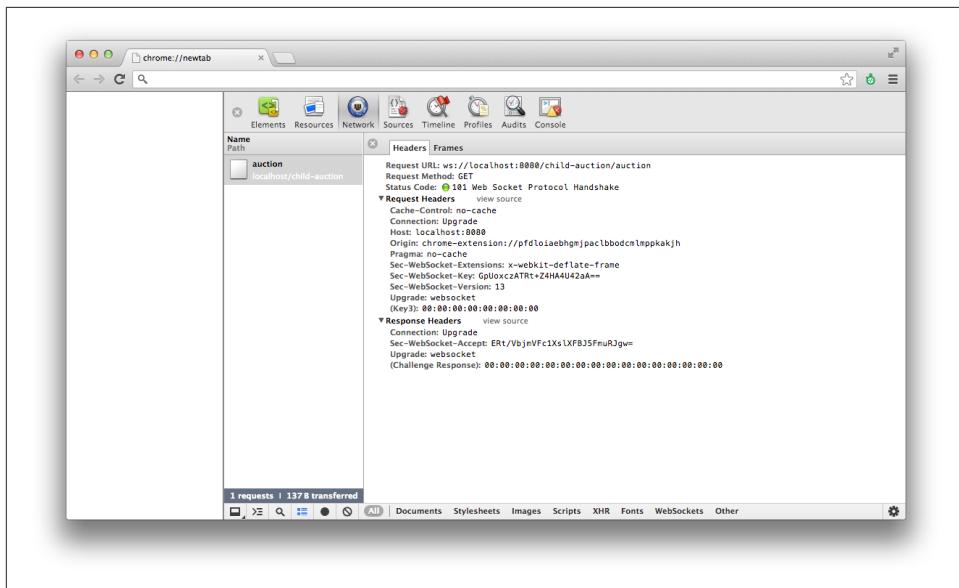


Figure 8-10. Initial WebSocket handshake in Chrome DevTools

You can also click on **WebSocket** at the bottom-right to show only the **WebSocket** endpoints. Click on **Frames** in the right panel to view the actual frames being exchanged between the client and server. **Figure 8-11**. The white colored rows represent incoming data, and the green (or gray on paper) rows - outgoing.

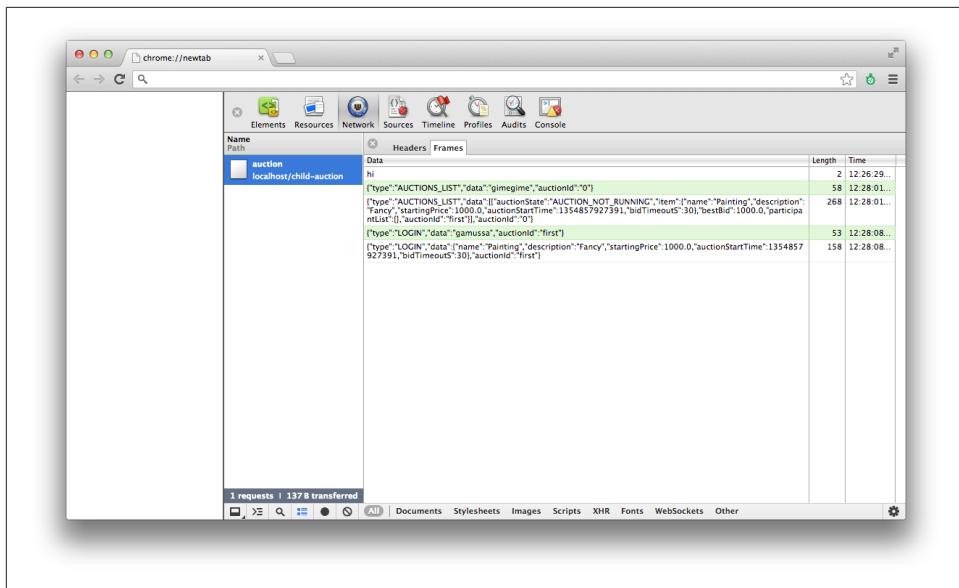


Figure 8-11. Monitoring WebSocket frames in Chrome Developer Tools

For more gore details you can navigate your Google Chrome to the secret URL - `chrome://net-internals`, which is a very useful URL showing a lot of additional information (see [Figure 8-12](#) and [Figure 8-13](#) ). You can find documentation about net-internal in [Chromium Design Documents](#)

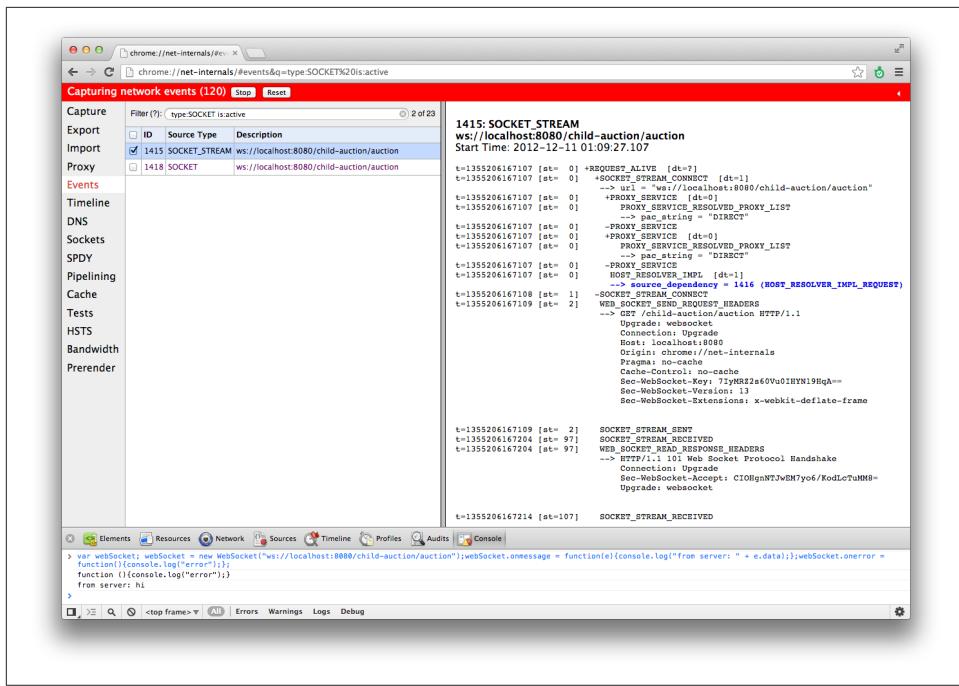


Figure 8-12. Details of initial handshake in Chrome Net Internals

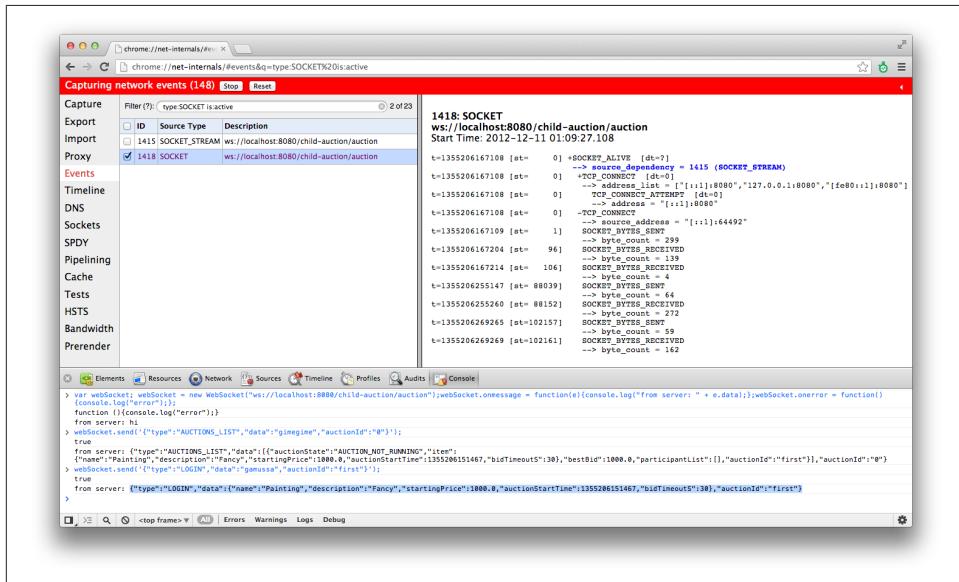
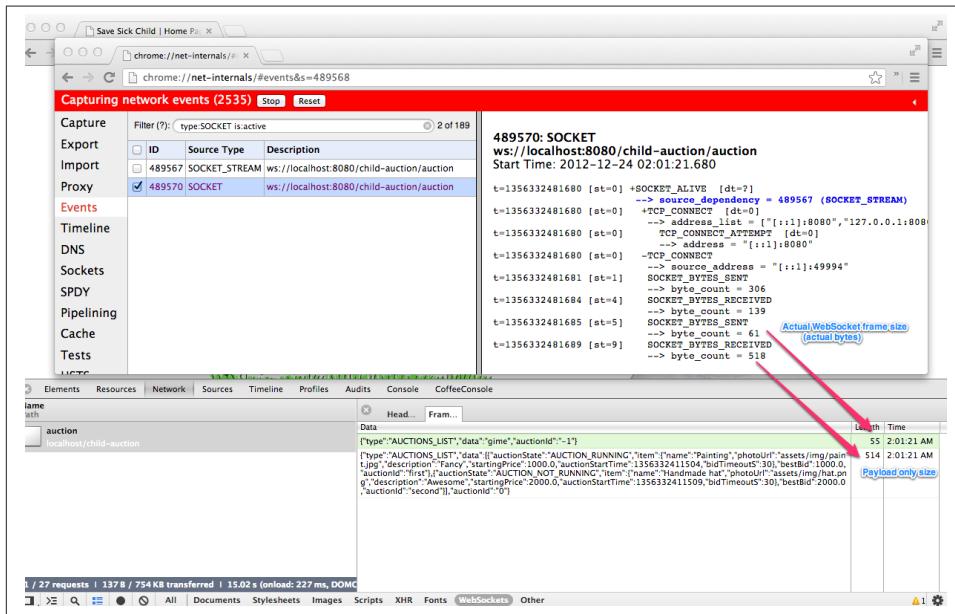


Figure 8-13. Details of the socket connection

Google Developer Tools show just the length of the data. But `chrome://net-internals` shows the actual size of the WebSocket frame too. [Figure 8-14](#) compares the views of net-internals and Developer Tools. As we learned earlier in this chapter, the total size of the frame is slightly different from the size of the payload. There are few more bytes for the frame header. Moreover, all outgoing messages will be masked by the browser (see the note in the “[The WebSocket Frame Anatomy](#)” on page 311 section). This frame’s mask is going to be transferred to the server as a part of the frame itself, which creates additional 32 bits (4 bytes) overhead.



*Figure 8-14. Dev tools and net-internals side by side*

## Sniffing WebSocket frames with Wireshark

Wireshark is a powerful and comprehensive monitoring tool for analyzing the network traffic. You can download it from [their web site](#). To start capturing the WebSocket traffic on `localhost` select the `Loopback` network interface from the left panel and click “Start” (see [Figure 8-15](#)).

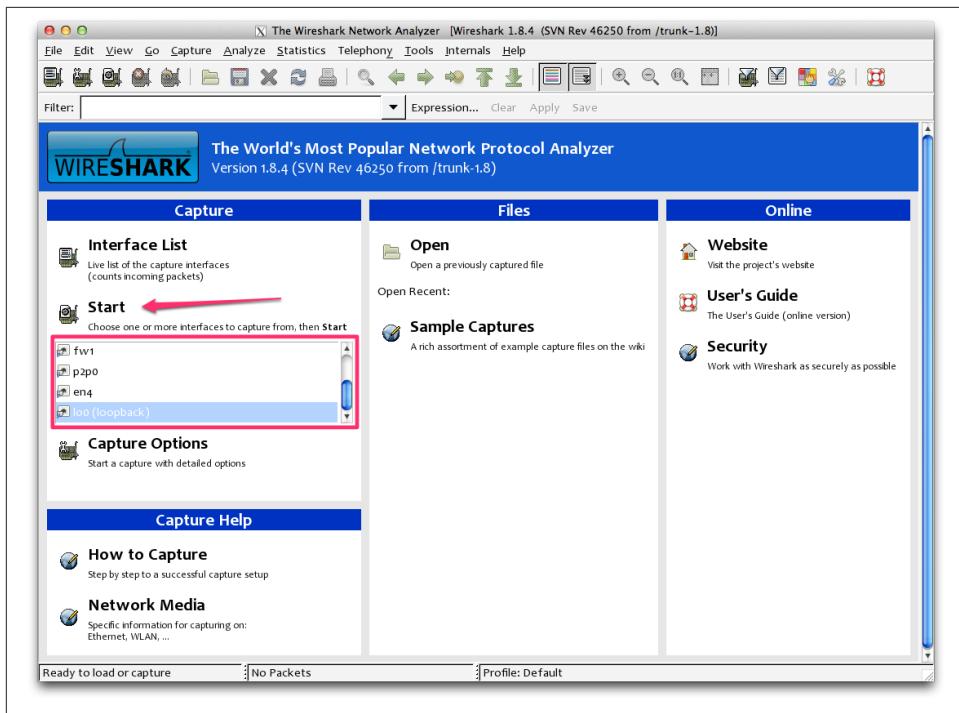


Figure 8-15. Wireshark application main view

Wireshark captures all network activity. Set up the filter to see only the data you are interested in. We want to capture HTTP and TCP traffic on the port 8080 because our WebSocket server (Oracle's GlassFish) runs on this port [Figure 8-16](#). Enter `http && (tcp.dstport==8080)` in the filter text box and click **Apply**.

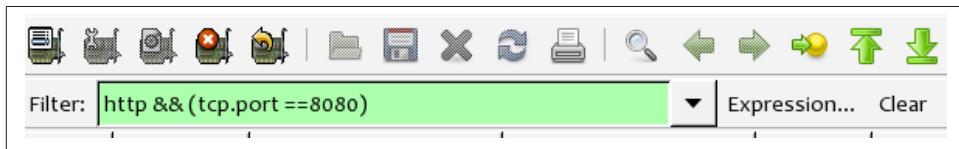


Figure 8-16. Filter setup

Now the Wireshark is ready to sniff at the traffic of our application. You can start the auction session and place bids. After you're done with the auction you can return to the Wireshark window and analyze what we got. You can see the initial handshake (GET request on [Figure 8-17](#) and the Upgrade response on [Figure 8-18](#)).

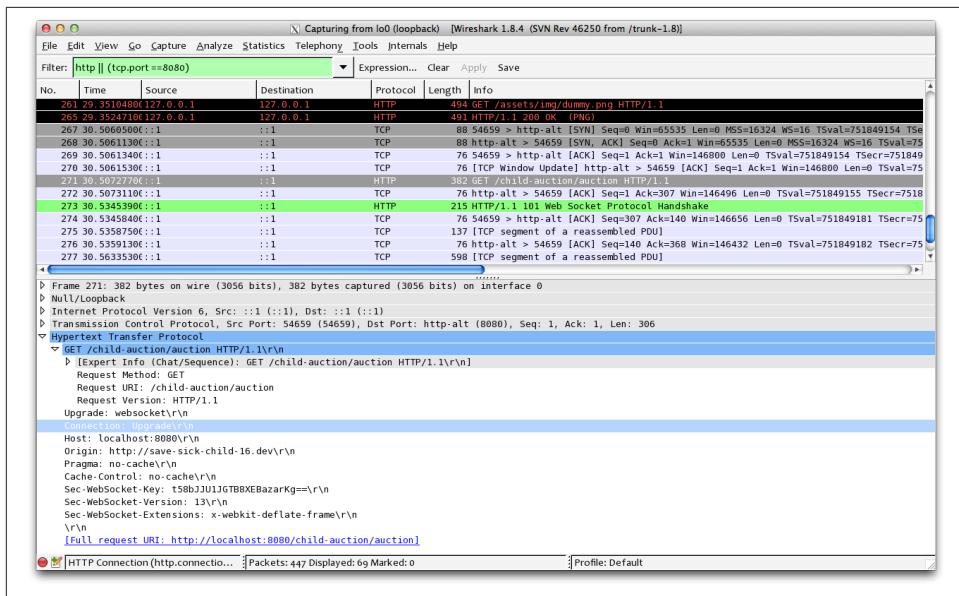


Figure 8-17. The GET request for protocol upgrade

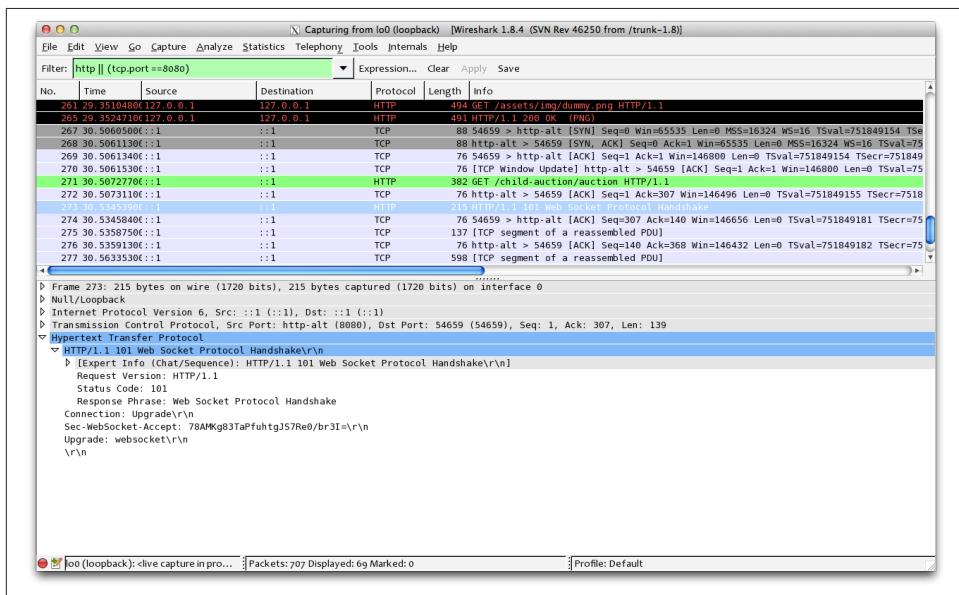
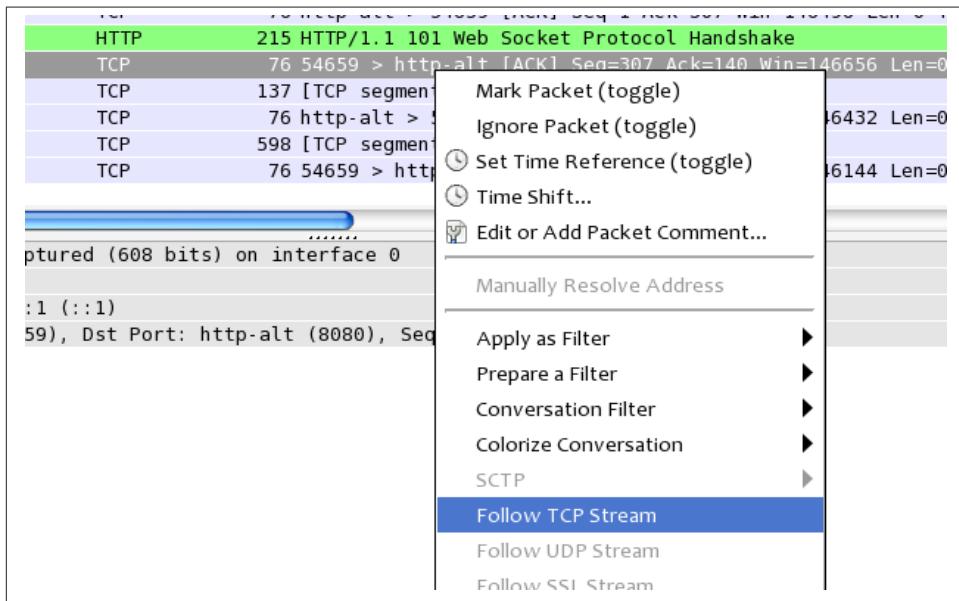


Figure 8-18. The GET response with protocol upgrade

After the successful connection upgrade, Wireshark captured the http-alt stream (this is how it reports the WebSocket's traffic) on the 8080 port. Right click on this row and select Follow TCP Stream [Figure 8-19](#).



*Figure 8-19. The GET response with protocol upgrade*

On the next screen you can see the details of WebSocket frame [Figure 8-20](#). We took this screenshot right after the auction application started. You can see here the data with the list of available auctions. The outgoing data is marked with *red* color and the incoming data is shown in *blue*.

The screenshot shows the NetworkMiner tool interface with the "Follow TCP Stream" option selected. The "Stream Content" pane displays a series of HTTP headers and a JSON payload. The JSON payload represents an auction list with two items: a handmade hat and a painting.

```
GET /child-auction/auction HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: localhost:8080
Origin: http://save-sick-child-16.dev
Pragma: no-cache
Cache-Control: no-cache
Sec-WebSocket-Key: t58bjJU1JGTB8XEBazarKg==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: x-webkit-deflate-frame

HTTP/1.1 101 Web Socket Protocol Handshake
Connection: Upgrade
Sec-WebSocket-Accept: 78AMKg83TaPfuhtgJS7Re0;br3I=
Upgrade: websocket

...eH.yG<dr.j' $.^V,.SQ:.TQ1j1 .)icGr?e.%x Ij|w.<tm..y _j03G5.~..
{"type": "AUCTIONS_LIST", "data": [{"auctionState": "AUCTION_NOT_RUNNING", "item": {"name": "Handmade_hat", "photoUrl": "assets/img/hat.png", "description": "Awesome", "startingPrice": 2000.0, "auctionStartTime": 1356387499577, "bidTimeouts": 30}, {"bestBid": 2000.0, "auctionId": "second"}, {"auctionState": "AUCTION_NOT_RUNNING", "item": {"name": "Painting", "photoUrl": "assets/img/paint.jpg", "description": "Fancy", "startingPrice": 1000.0, "auctionStartTime": 1356387499574, "bidTimeouts": 30}, {"bestBid": 1000.0, "auctionId": "first"}]}, "auctionId": "0"}
```

Below the content pane, there is a dropdown menu labeled "Entire conversation (1028 bytes)". At the bottom, there are several buttons: Find, Save As, Print, ASCII, EBCDIC, Hex Dump, C Arrays, Raw (selected), Help, Filter Out This Stream, and Close.

Figure 8-20. WebSocket frame

The screenshot [Figure 8-21](#) was taken after the auction has been closed. You can see here the entire data that were sent over the WebSocket connection.

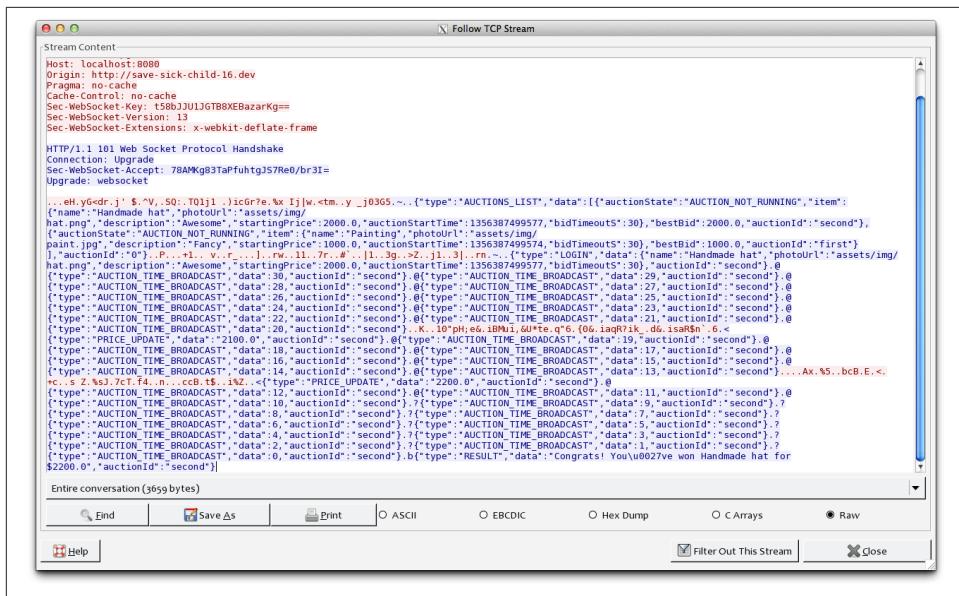


Figure 8-21. Entire auction conversation

## Save The Child Auction Protocol

Since WebSocket is just a transport protocol, we need to come up with the application-level protocol of how the auction messages should be formatted in the client-server interaction. This is how we decided to do it:

1. The client's code connects to the WebSocket endpoint on the server.
2. The client's code sends the AUCTION\_LIST message to retrieve the list of currently running auctions.

*Example 8-4. The AUCTION\_LIST Request*

```
{
  "type": "AUCTIONS_LIST", // ❶
  "data": "empty", // ❷
  "auctionId": "-1" // ❸
}
```

- ❶ The type of the message is AUCTION\_LIST
- ❷ This message doesn't send any data
- ❸ This message doesn't request for any specific auction id, so we just send -1.

Let's review the JSON object that will be arriving from the server as the auction's response.

*Example 8-5. Response with auction list data*

```
{  
    "type": "AUCTIONS_LIST",    // ①  
    "data": [                  // ②  
        {  
            "auctionState": "AUCTION_NOT_RUNNING",  
            "item": {                      // ③  
                "name": "Painting",  
                "description": "Fancy",  
                "startingPrice": 1000.0,  
                "auctionStartTime": 6000,  
                "bidTimeoutS": 30  
            },  
            "bestBid": 1000.0,  
            "participantList": [],  
            "auctionId": "first" // ④  
        },  
        {  
            "auctionState": "AUCTION_RUNNING",  
            "item": {  
                "name": "Handmade hat",  
                "description": "Awesome",  
                "startingPrice": 2000.0,  
                "auctionStartTime": 6000,  
                "bidTimeoutS": 30  
            },  
            "bestBid": 2000.0,  
            "participantList": [],  
            "auctionId": "second"  
        }  
    ],  
    "auctionId": "0"  
}
```

- ① The message type is AUCTION\_LIST
- ② The data property of the response object contains the list of all running auctions. An auction can be in one of three states: *not running*, *running*, or *finished*.
- ③ The item property of the response object is a nested object that represents the auction item.
- ④ The auctionId property contains a unique identifier of the selected auction.

3. The user picks the auction from the list, enters a desired nick name, and joins to the auction. The client-side application sends the following login message.

```
{  
    "type": "LOGIN", // ①  
    "data": "gamussa", // ②
```

```
        "auctionId": "second" //❸  
    }
```

- ❶ The message type is LOGIN
- ❷ The data property of the request contains the user's nickname.
- ❸ The auctionId property helps the server-side code to route the message to the correct auction.



As soon as the handshake completes successfully, the server side code that implements WebSocket protocol exposes the WebSocket Session object. The Session object encapsulates the conversation between the WebSocket endpoint (server - side) and remote endpoint (browser). Check the documentation for your server side framework for details about how it handles and exposes the remote endpoints in API.

4. Each time when the user enters the bid price the client's code sends following bid message:

*Example 8-6. Bit message*

```
{  
    "type": "BID",  
    "data": "1100.0",  
    "auctionId": "second"  
}
```

Above is the outgoing message. When user clicks on the “Bid!” button the value from the Bid text box is wrapped into the BID message. On the server, when the new higher BID message arrives, the message PRICE\_UPDATE has to be broadcast to all active clients.

5. The PRICE\_UPDATE message

*Example 8-7. The PRICE\_UPDATE Message*

```
{  
    "type": "PRICE_UPDATE", //❶  
    "data": "1300.0", //❷  
    "auctionId": "second"  
}
```

- ❶ If some auction participant outbids others, the rest of the participants will receive an update.
- ❷ Such an update will contain the current highest bid.

6. The RESULT message

*Example 8-8. The RESULT Message*

```
{  
    "type": "RESULT",  
    "data": "Congrats! You\u0027ve won Painting for $1300.0",  
    "auctionId": "first"  
}
```

After the auction ends, the server broadcasts the message with the final auction results. If the winning user is online and connected to the auction server, she'll receive the message with congratulations. Other participants will get the "Sorry, you didn't win" notification.

This is pretty much it. The amount of code to implement the client's side of the auction is minimal. After the connection and upgrade are done, most of the processing is done in the message handler of the WebSocket object's `onmessage`.

## Summary

After reading this chapter you should see the benefits of using WebSocket protocol in Web applications. In many cases WebSocket is an ultimate means for improving the application performance by reducing the network latency and removing the HTTP-headers overhead. You've learned how to integrate the WebSocket-based functionality into the existing HTTP-based application Save The Child. There is no need to make the entire communication of the Web application over WebSocket. Use this powerful protocol when it's improves the performance and responsiveness of your application.

As a side benefit, you've learned how to use the network monitoring capabilities of Google Chrome Developer Tools and Wireshark application by sniffing the WebSocket traffic. You can't underestimate the importance of monitoring tools, which are the best friends of Web developers.

## References

[igrorik] Ilya Grigorik. *High Performance Browser Networking* 2013 p.13



# Introduction to Web Application Security

Every newly deployed Web application creates a new security hole in accessing of your organization's data. The hackers get access to the data by sneaking through the ports that are supposedly hidden behind the firewalls. There is no way to guarantee that your Web application is 100% secure. If it was never attacked by hackers most likely it's too small and is of no interest to them.

This chapter is a brief overview of major security vulnerabilities that Web application developers need to be aware of. We'll also cover delegated authorization with OAuth, and possible authentication and authorization scenarios for our Save The Child application.

There are plenty of books and online articles that cover security, and enterprises usually have dedicated teams handling security for the entire organization. Dealing with security threats is their bread and butter, and this chapter won't have revelations for security professionals. But a typical enterprise application developer just knows that each person in the organization has an account in some kind of a naming server that's stores IDs, passwords and roles, which takes care of authentication and authorization flows. Application developers should find useful information in this chapter.

If an enterprise developer needs an access to some internal application, opening an issue with technical support team grants the required access privileges. But software developers should have at least a broad understanding of what makes a Web application more or less secure and which threats Web applications face - this is what this chapter is about. To implement any of the security mechanisms mentioned in this chapter you'll need to do additional research.



A good starting point for establishing security process for your enterprise project is Microsoft's Web site [Security Development Lifecycle](#). It contains a number of documents describing the software development process that helps developers build more secure software and address security compliance requirements while reducing development cost.

## HTTP vs HTTPS

Imagine a popular night club with a tall fence and two entry doors. People are waiting in lines to get in. The door number 80 is not guarded in any way - a college student checks the tickets and let people in. The other door has the number 443 on it, and it's protected by an armed bully letting only qualified people in. The chances of unwanted people getting into the club through the door 443 are pretty slim (unless the bully is corrupted), which is not the case with door 80 - once in a while people that have no right to be there get inside.

On a similar note, your organization created *network security* with a firewall (the fence) with only two ports (the doors) opened: 80 for HTTP requests and 443 for HTTPS. One door's not secure, the other one is.



Don't assume that your Web application is secure if it's deployed behind the firewall. As long as there are opened ports that allow external users accessing your Web application, you need to invest into the *application security* too.

The letter "s" in *https* means *secure*. Technically, https creates a secure channel over the insecure Internet connection. In the past, only the Web pages that dealt with logins, payments or other sensitive data would use URLs starting with https. Today, more and more Web pages use https and rightly so, because it forces Web browsers to use Secure Socket Layer (SSL) or its successor Transport Layer Security (TLS) protocol for encrypting all the data (including request and response headers) that travel between connected Internet resources. The O'Reilly book "High Performance Browser Networking" by Ilya Grigorik contains [a chapter](#) with detailed coverage of the TLS protocol.

The organization that runs the Web server creates the public key certificate that has to be signed by a trusted certificate authority (otherwise browsers will display invalid certificate warnings). The authority certifies that the holder of the certificate is a valid operator of this Web server. SSL/TLS layers authenticate the servers using these certificates to ensure that the browser's request is being processed by the proper server and not by some hacker's site.

When the client connects to the server via https, the client offers to the server a list of supported ciphers (authentication-encryption-decryption algorithms). The server replies with the cipher they both support.



The annual **Black Hat** computer security conference is dedicated to the information security. This conference is attended by both hackers and security professionals.

If https is clearly more secure than http, why not every Web site uses only https communications? Since https encrypts all the messages that travel between the client's browser and the server, such communications are a slower and need more CPU power comparing to http-based data exchanges. But this little slowness won't be noticeable in most Web applications (unless thousands of concurrent users hit the Web server), while the benefits of using https are huge.

When entering any sensitive or private information in someone's Web application always pay attention to the URL to make sure that it uses https protocol.

As a Web developer, you should always use the https protocol to prevent an attacker from stealing the user's session id. The fact that the National Security Agency has broken SSL encryption algorithm is not the reason for your application for not using https.

## Authentication and Passwords

*Authentication* is an ability to confirm that the user is who he claims to be. The fact the the user has provided a valid id/password combination just proves that he is known to the Web application. That's all.

Specifying the correct user id/password combination may not be enough for some Web applications. Banks often ask for additional information like "What's your pet's name?" or "What's your favorite movie?"

Large corporations often use the RSA Secure ID (a.k.a. RSA hard token), which is a physical device with random-generated combination of digits. This combination changes every minute or so and has to be entered as a part of the authentication process. Beside physical devices there are programs (soft tokens) that perform user authentication in a similar way. Many financial institutions, social networks, and large Web portals support 2-factor verification - in addition to asking for a user ID and password, they send you either an email, voice mail, or text message with a code that you'll need to use after entering the right id/password combination.

To make the authentication process more secure some systems check the biometrics of the user. For example, in the USA the Global Entry system is implemented in many

international airports. People who successfully passed a special background check are entered into the system deployed in the passport control checkpoints. These applications deployed in a special kiosks scan the users passports, check the face topography and fingerprints. The process takes several seconds and the *authenticated* person can pass the border without waiting in long lines.

Biometrics devices become more common these days, and the fingerprint scanners that can be connected to the user's computer are very inexpensive. Apple's iPhone 5S unlocks the phone based on the fingerprint of its owner - no need to enter the passcode. In some places you can enter the gym only after your fingerprints are scanned and matched. National Institute of Standards and Technology hosts a discussion about using biometrics web services, and you can participate by sending an email to [bws-request@nist.gov](mailto:bws-request@nist.gov) with *subscribe* as the subject.

## Basic and Digest Authentication

HTTP protocol defines two types of authentication: **Basic** and **Digest**. All modern web browsers support them, but basic authentication uses base64 encoding and no encryption, which means it should be used only with https protocol.

A Web server administrator can configure certain resources to require basic user authentication. If a Web browser requests such protected resource while the user didn't login to the site, the Web server (not your application) sends the HTTP response containing HTTP status code 401 - *Unauthorized* and *WWW-Authenticate: Basic*. The browser pops up the login dialog box. The user enters the ID/password, which are turned into an encoded *userID:password* string and sent to the server as a part of HTTP header. Basic authentication provides no confidentiality because it doesn't encrypt the transmitted credentials. Cookies are not used here.

With digest authentication, the server also responds with 401 (*WWW-Authenticate: Digest*). However, it also sends along additional data which allows the Web Browser to apply a hash function to the password. Then the browser sends encrypted password to the server. Digest authentication is more secure than the basic one, but it's still less secure than authentication that uses public keys or Kerberos authentication protocol.



The HTTP status code 403 (*Forbidden*) differs from 401. While 401 means that the user needs to login to access the resource, 403 means that the user is authenticated, but his security level is not high enough to see the data. For example, not every user role is authorized to see the Web page with salary report.

In application security the term *man in the middle attack* refers to the case when an attacker intercepts and modifies the data transmitted between two parties (usually the

client and the server). Digest authentication protects the Web application from losing the clear text password to an attacker, but doesn't prevent man in the middle attacks.

While digest authentication only encrypts the user's and password, using HTTPS protocol encrypts everything that goes between the Web browser and the server.

## Single Sign-on

Pretty often an enterprise user has to work with more than one corporate Web applications, and maintaining, remembering, supporting multiple passwords should be avoided. Many enterprises implement internally so-called Single Sign-On (SSO) mechanism to eliminate the need for the user to enter his login credential more than once even if the user works with multiple applications. Accordingly, signing out from one of these applications terminates the user's access to all of them. SSO solutions make authentication totally transparent to your application.

With SSO, when the user logs on to your application, the logon request is intercepted and handled by pre-configured SSO software (e.g. Oracle Enterprise Single Sign-On, CA SiteMinder, IBM Security Access Manager for Enterprise SSO, or Evidian Enterprise SSO). The SSO infrastructure verifies user's credentials by making a call to a corporate LDAP server and creates a user's session. Usually a Web server is configured with some Web agent, which will add the user's credential to the HTTP header, which your application can fetch.

The future access to the protected Web application is handled automatically by the SSO server without even displaying a logon window as long as the user's session is active. SSO servers also log all login attempts in a central place, which can be very important to meet the enterprise regulatory requirements (e.g. Sarbanes-Oxley in financial industry or medical confidentiality in the insurance business).

In the consumer-oriented Internet space single (or reduced) sign-on solutions become more and more popular. For example, some Web applications allow reusing your Twitter or Facebook credentials (provided that you've logged on to one of these applications) without the need to go through additional authentication procedures. Basically, your application can delegate authentication procedures to Facebook, Twitter, Google and other authorization services, which we'll discuss later in the section on OAuth.

Back in 2010, Facebook has introduced their **SSO solution** that still helps millions of people to log on to other applications. This is especially important in the mobile world, where users' typing should be minimized. Instead of asking the user to enter credentials, your application can show the button "Login with Facebook".

Facebook has published JavaScript API that allows implementing Facebook Login in your Web applications (they also offer native API for iOS and Android apps). For more details visit online documentation on [FaceBook Login API](#).

Besides Facebook other popular social networks offer authentication across the applications:

- If you want your application to have a button “Login with Twitter”, refer to the Sign in with Twitter API [documentation](#).
- LinkedIn is a popular social network for professionals. It also offers API to create the button “Sign in with LinkedIn”. For details visit LinkedIn [online documentation](#) for developers.
- Google also offers the OAuth-based authentication API. Details about their client library for JavaScript are published [online](#). For implementing SAML-based SSO with Google, visit [this Web page](#).
- Mozilla offers a new way to sign-in using any of your existing email addresses using [Persona](#).
- Several large organizations (e.g. Google, Yahoo!, Microsoft, Facebook) either issue or accept [OpenID](#), which allows to sign in to more than 50000 Web sites.

Typically, large enterprises would not want users to use logins from social networks. But some organizations started integrating their applications with social networks. Especially now, with the spread of mobile devices, the users may need to be able to get authenticated and authorized while being outside of the enterprise perimeter. We'll discuss it in more detail in the section on OAuth.

### Save The Child and SSO

Is there a use of SSO for our charity application Save The Child? Certainly. In this book we're mostly concerned about developing the UI for the consumer-facing part of this application. But there is also a back office team that is involved with the content management that produces the information for the consumer.

For example, the employees of our charity organization create fund-raising campaigns in different cities. If an employee of this firm logged in to his desktop, our Save The Child Web application shouldn't ask him to login. SSO can be a solution here.

## Dealing With Passwords

It might sound obvious, but we'll still remind you that the Web client should never send passwords in clear text. Use [Secure Hash Algorithms](#) (SHA). Longer passwords are more secure, because if an attacker will try to guess the password by using dictionaries to generate every possible combination of characters (a [brute-force attack](#)), it'll take a lot more time with long passwords. Periodic changing of the passwords makes the hacker's work more difficult too. Typically, after successful authentication the server creates and sends to the Web client the session ID, which is stored as a cookie on the client's computer. Then, on each subsequent request to the server the Web browser will place the

session id in the HTTP request object and send it along with each request. Technically, the user's identity is always known at the server side, so the server-side code can re-authenticate the user more than once (without the user even knowing it) whenever the Web client requests the protected resource.



Salted hashes increase security by adding *salt* - a randomly generated data that's concatenated with the password and then processed by a hash function.

Have you ever wondered why Automated Teller Machines (ATM) often ask you to enter PIN more than once? Say, you've deposited a check and then want to see the balance on your account. After the check deposit has been completed your ATM session was invalidated to protect the careless users who may rush out from the bank in a hurry as soon as the transaction is finished. Otherwise the next person by the ATM could have requested a cash withdrawal from your bank account.

On the same note, if the Web application's session is idling more than allowed time interval, the session should be automatically invalidated. For example, if a trader in a brokerage house is not interacting with the Web trading application for some time, invalidate the session programmatically to exclude the situation when the trader stepped out, and someone else is buying financial products on his behalf.

## Authorization

*Authorization* is a way to determine which operations the user can perform and what data he can access. For example, the owner of the company can perform money withdrawals and transfers from the online business bank account, while the company accountant is provided with the read-only access.



Similarly to authentication the user's authorization can be checked more than once during the user's session. As a matter of fact, authorization can even change during the session (e.g. a financial application can allow trades only during business hours of the stock exchange).

Users of the application are grouped by roles, and each role comes with a set of privileges. The user can be given a privilege to read and modify certain data, while other can be hidden. In the relational DBMS realm there is a term *row-level security*, which means that the same query can produce different results to different users. Such security policies are implemented at the data source level.

A simple use case where row-level security is really useful is a salary report. While the employee can see only his salary report, the head of department can see the data of all subordinates.

Authorization is usually linked with the user's session. HTTP is stateless protocol, so if a user retrieves a Web page from a Web server, and then goes to another Web page, this second page does not know what has been shown or selected on the first one. For example, in case of an online store the user adds an item to the shopping cart and moves to another page to continue shopping. To preserve the data reused in more than one Web page (e.g. the content of the shopping cart) the server-side code must implement *session tracking*. The session information can be passed all the way down to the database level when need be.



Session tracking is usually controlled on the server side. If you'd like to get familiar with session tracking options in greater details, consult the product documentation for the server or technology being used with your Web application. For example, if you use Java, you can read Oracle's documentation for their **WebLogic server** that describes the options for session management.

## OAuth-Based Authentication and Authorization

To put it simple, OAuth is a mechanism of delegated authorization. OpenID Connect is a OAuth-based mechanism for authentication.

Most likely you ran into Web applications that offer you to share your actions via social networks. For example, you just made a donation and want to share this information via social networks.

If our charity application needed to access the user's Facebook account for authentication, the charity app could have asked the user Facebook's ID and password. This wouldn't be the right approach, because the charity application would get the user's Facebook id/password in clear text along with the full access to the user's Facebook account. But the charity app only needed to authenticate the Facebook user. Hence there is a need for a mechanism to give a *limited access* to Facebook to the third party applications.

**OAuth** became one of the mechanisms for providing a limited access to an authorizing facility. OAuth is "An open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications". Its **current draft specification** provides the following definition:

*The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating*

*an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.*

Aron Parecki writes in his book “OAuth 2.0: The Definite Guide: *Many cars today come with a valet key. It is a special key you give a parking attendant. Unlike your regular key, the valet key can only turn on the engine but will not open the trunk or glove compartment, or may not let the car drive more than a mile or two. Regardless of what restrictions the valet key imposes, the idea is very clever. You give someone limited access to your car with a special key, while using your regular key to unlock everything.*” This is a good example of a limited access to a resource in a real life. The OAuth 2.0 authorization server gives the requesting application an *access token* (think valet key) so it can access, say the charity application.

OAuth allows users to give limited access to third-party applications without giving away their passwords. The access permission is given to the user in a form of access token with limited privileges and for a limited time. Coming back to our example of communication between the charity app and Facebook (unless we have our own enterprise authentication server), the former would get a limited access to the user’s Facebook account (just the valet key, not the master key).

OAuth becomes a standard protocol for developing the applications that require authorization. With OAuth application developers won’t need to use proprietary protocols if they need to add an ability to identify the user via multiple authorization servers.

## Federated Identity with OpenID Connect and JSON Web Tokens

There is a term **federated identity**, which Wikipedia defines as the means of linking a person’s electronic identity and attributes, stored across multiple distinct identity management systems. This is similar to the enterprise single sign-on, but is wider because the authentication token with the information about the user’s identity can be passed across multiple departments or organizations and software systems.

Microsoft’s publication called **“A Guide to Claims-Based Identity and Access Control”** includes a section on **federated identity for Web applications** with greater details on this subject.

In the past, the markup language **SAML** was the most popular open standard data format for exchanging authentication and authorization data. **OpenID Connect** is a newer open standard. It’s a layer on top of OAuth 2.0 that simply verifies the identity of the user. **OpenID providers** that can confirm the user’s identity include such companies as Google, Yahoo!, IBM, VeriSign and more. Typically OpenID Connect uses so-called **JSON Web Tokens(JWT)**, which should eventually replace popular XML-based SAML tokens. JSON Web Token is base64 encoded and signed JSON data structure. While OAuth 2.0 spec doesn’t mandate using JSON Web Tokens, they became a de-facto standard token format.

To have a better understanding of how the JSON Web tokens are encoded, visit the [Federation Lab](#), which is a Web site with a set of tools for testing and verification of various identity protocols. In particular, you can enter a JWT in a clear text, select a secret signature and encode the token using the HS256 algorithm as shown on the diagram [Figure 9-1](#).

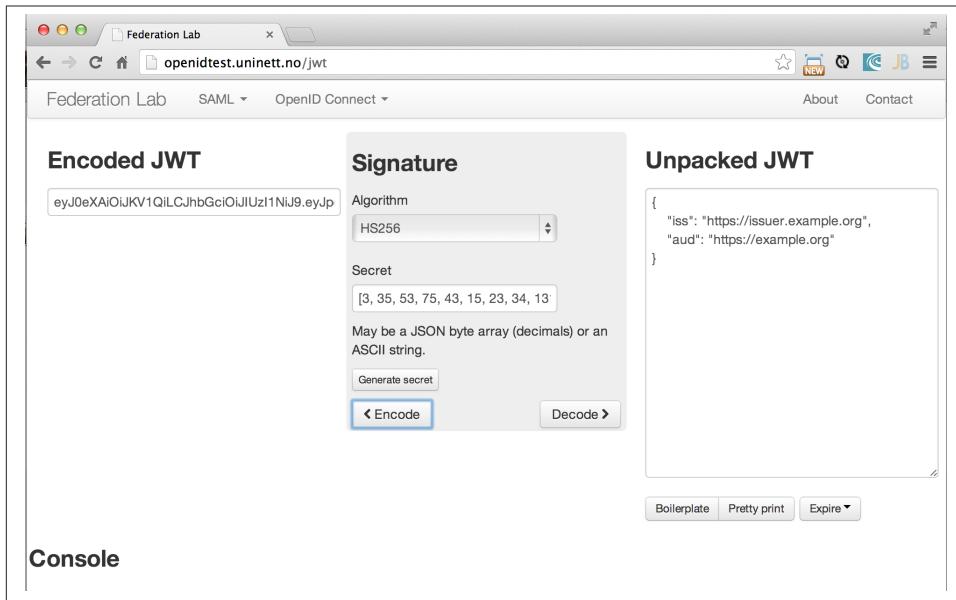


Figure 9-1. Encoding JSON Web Token

## Using Facebook API

Facebook is one of the authorization servers that offer OAuth-based authentication and authorization API. The online document "[Quickstart: Facebook SDK for JavaScript](#)" is a good starting point.

Before using the SDK you need to register your application with Facebook by creating a client ID and obtaining the client secret (the password). Then use the JavaScript SDK code (provided by Facebook) in your Web application. Include the newly created app id there. During this registration stage you'll need to specify the URI where the user should be redirected in case of successful login. Then add a JavaScript code to support required Facebook API (e.g. for Login) to your application. You can find a sample JavaScript code that uses Facebook Login API in [this guide](#).

Facebook Login API communicates with your application by sending events as soon as the login status changes. Facebook will send the authorization token to your application's code. As we mentioned earlier, authorization token is a secure encoded string that iden-

tifies the user and the app, contains the information about permissions and has the expiration time. Your application's JavaScript code makes calls to Facebook SDK API, and each of these calls will include the token as a parameter or inside the HTTP request header.

## OAuth 2.0 Main Actors

Any communications with OAuth 2.0 servers are made through https connections. Below are the main actors of the OAuth flows:

- The user who owns the account with some of the resource servers (e.g. account at Facebook, Google et al.) is called *resource owner*.
- The application that tries to authenticate the resource owner is called *the client application*. This is an application that offers the buttons like "Login with Facebook", "Login with Twitter" and the likes. The client application
- The *resource server* is a place where the resource owner stores his data (e.g. Facebook, Google et al.)
- The *authorization server* checks the credentials of the resource owner and returns an authorization token with limited information about the user. It can be the same as resource server, but not necessarily. Facebook, Google, Windows Live, Twitter, GitHub are some of the examples of authorization servers. For the current list of OAuth 2.0 implementations visit [oauth.net/2](#).

To implement OAuth in your client application, you need to pick the a resource/authorization server and study their API documentation. Keep in mind that OAuth defines two types of clients - public and confidential. Public clients use embedded password while communicating with the authorization server. If you're going to keep the password inside your JavaScript code, it won't be safe. To be considered a confidential client, a Web application should store its password on the server side.

OAuth has provisions for creating authorization tokens to browser-only applications, for mobile applications, and for the server-to-server communications. For the in-depth coverage get the O'Reilly book by Aaron Parecki "[OAuth 2.0: The Definite Guide](#)".

## Save The Child and OAuth

We can distinguish two major scenarios of a third party application working with the OAuth server. In one scenario OAuth authorization servers are publicly available, in the other - privately own by the enterprise. Let's consider these scenarios in the context of our charity non-profit organization.

## Public Authorization Servers

A Facebook account owner works with *the client* (the Save The Child application). The client uses the external *authorization server* (Facebook) to request the authorization of the user's work with the charity application. The client has to be registered (has assigned clientID, secret, and redirect URL) with the authorization server to be able to participate in such OAuth flow. The authorization server returns a token offering a limited access (e.g. to Facebook's account) to the Save The Child application. The diagram [Figure 9-2](#) shows a use case where Save The Child uses Facebook for authentication and authorization.

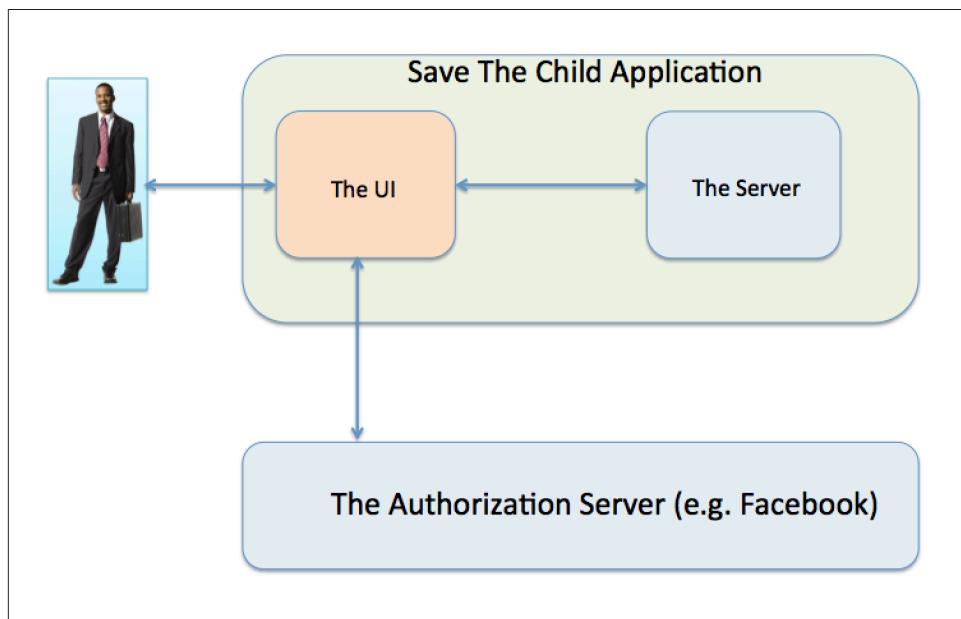


Figure 9-2. Save The Child and OAuth

While the client application tries to get an authorization from the authorization server, it can open a so-called *consent window* that should warn the user that the Save The Child application tries to access certain information from your Facebook or Google's account. In this scenario the user still has a chance to deny such access. It's a good idea to display a message that the user's password (e.g. to Facebook or Google) will not be given to the client application.

Your application should request only minimum access to the user's resource server. For example, if the Save The Child application just needs to offer an easy authentication for all Facebook users, then do not request the write access to the user's Facebook account. On the other hand, if a kid was cured with the involvement of our charity application,

and he wants to share the good news with his Facebook friends, the Save The Child application needs a write permission to the user's Facebook account.

The UI code of the Save The Child application doesn't have to know how to parse the token returned by the authorization server. It can simply pass it to the Save The Child's server software (e.g. via the HTTP request header). The server has to know how to read and decipher the information from the token. The client application sends to the authorization server only the client id, and not the *client secret* needed for deciphering the user's information from the token.

### Private Authorization Servers

The OAuth authorization server may be configured inside the enterprise, but may serve not only internal employees, but external partners too. Say, one of the upcoming charity events is a marathon to fight cancer. To prepare such a marathon our charity organization needs to use help of a partner company named Global Marathon Suppliers, which will take care of the logistics (providing banners, water, food, rain ponchos, blankets, branded tents et al.)

It would be nice if our supplier would know up-to-date information about the number of participants in this event. If our charity firm will set them up with the access to our internal authorization server, the employees of the Global Marathon Suppliers can have limited access to the marathon participants. On the other hand, if the suppliers would open a limited access to their data, this could increase the productivity of the charity company employees. This is practical and cost-saving setup.



The authors of this book were helping [Leukemia and Lymphoma Society \(LLS\)](#) with developing both front and back end software. LLS ran a number of successful marathons as well as many other campaigns for charity causes. We also used [OAuth solution from Intuit QuickBooks](#) in the billing workflows for our software product for insurance industry at [SuranceBay](#). Our partner companies get limited access to our billing systems and to our software can access theirs.

## Top Security Risks

[Open Web Application Security Project \(OWASP\)](#) is an open source project focused on improving security of Web applications. OWASP is a collection of guides and tools for increasing security of Web applications. OWASP publishes and maintains the list of [top 10 security risks](#). Figure [Figure 9-3](#) shows how this list looked in 2013:

## Top 10 2013-Top 10

2013 Table of Contents	
<a href="#">← Risk</a>	<a href="#">2013 Top 10 List</a>
<a href="#">A1-Injection</a>	Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.
<a href="#">A2-Broken Authentication and Session Management</a>	Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities.
<a href="#">A3-Cross-Site Scripting (XSS)</a>	XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.
<a href="#">A4-Insecure Direct Object References</a>	A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data.
<a href="#">A5-Security Misconfiguration</a>	Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.
<a href="#">A6-Sensitive Data Exposure</a>	Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser.
<a href="#">A7-Missing Function Level Access Control</a>	Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.
<a href="#">A8-Cross-Site Request Forgery (CSRF)</a>	A CSRF attack forces a logged-on victim's browser to send a forged HTTP request, including the victim's session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests the vulnerable application thinks are legitimate requests from the victim.
<a href="#">A9-Using Components with Known Vulnerabilities</a>	Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover. Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.
<a href="#">A10-Unvalidated Redirects and Forwards</a>	Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages. Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

Figure 9-3. Top 10 security risks circa 2013

This Web site allows you to drill down on each of the items from this list, see the illustration of the selected security vulnerability and recommendations on how to prevent it. You can also download this list as a [PDF document](#). Let's review a couple of the top 10 security threats: injection and cross-site scripting.

## Injection

If a bad guy will be able to *inject* a piece of code that will run inside your Web application, such code can steal or damage the data from this application. In the world of compiled libraries and executables injecting malicious code would be a rather difficult task. But if an application uses interpreted languages (e.g. JavaScript or clear text SQL) the task of injecting malicious code becomes a lot easier than you might think. Let's look at a typical example of SQL injection.

Say your application can search for data based on some keywords the user enters into a text input field. For example, to find all donors in the city of New York the user will enter the following:

“New York”; *delete from donors;*

If the server side code of your application would be simply attaching the entered text to the SQL statement, this could result in execution of the following command:

*Select \* from donors where city="New York"; delete from donors;*

This command doesn't require any additional comments, does it? Is there a way to prevent the users of your Web application from entering something like this? The first thing that comes to mind is to not allow the user to enter the city, but force him to select it from the list. But such a list of possible values might be huge. Besides, the hacker can modify the HTTP request after the browser sends it to the server.



Always use pre-compiled SQL statements that use parameters to pass the user's input into the database query (e.g. the *PreparedStatement* in Java).

The importance of the server-side validation shouldn't be underestimated. In some scenarios you can come up with a regular expression that checks for the matching patterns in the data received from the clients. In other cases you can write a regular expression that invalidate the data if it contains SQL (or other) keywords that leads to modifications of the data on the server.



Always minimize the interval between validating and using the data.

In the ideal world the client-side code should not even send the non-validated data to the server. But in real-world you'll end up with duplicating some of the validation code in both the client and server.

## Cross-Site Scripting

Cross-site scripting (XSS) is when an attacker injects the malicious code into a browser-side script of your Web application. The user was accessing a trusted Web site, but got an injection from a malicious server that reached the user via the trusted server (hence cross-site). Single-page AJAX-based applications make lots of under-the-hood requests

to the servers, which increases the attack surface comparing to traditional legacy Web sites that would be downloading Web pages a lot less frequently. XSS can happen in three ways:

- Reflected (a.k.a. phishing) - the Web page contains a link that seems valid, but when the user clicks on it, the user's browser receives and executes the script created by the attacker.
- Stored - the external attacker managed to store the malicious script on the server that hosts someone's Web application so every user will get it as a part of the Web page and their Web browser will execute it. For example, if a user's forum allows posting texts that include JavaScript code, a malicious code typed by a "bad guy" can be saved in the server's database and executed by users' browsers visited this forum afterward.
- Local - no server is involved. Web page A opens Web page B with malicious code, which in turn modifies the code of the page A. If your application uses a hash-tag (#) in URLs (e.g. <http://savesickchild.org#something>), make sure that before processing this *something* doesn't contain anything like "javascript:somecode", which may have been attached to the URL by an attacker.

W3C has published the draft of the **Content Security Policy** document - "a mechanism web applications can use to mitigate a broad class of content injection vulnerabilities, such as cross-site scripting".

## STRIDE - Classification of Security Threats

Microsoft has published a **classification** that divides security threats into six categories (hence six letters in the acronym STRIDE):

- Spoofing - an attacker pretends to be a legitimate user of some application, e.g. a banking system. This may be implemented using XSS.
- Tampering - modifying the data that were not supposed to be modified (e.g. via SQL injection).
- Repudiation - the user denies that he sent the data (e.g. made an online transaction like purchase or sale) by modifying application's log files.
- Information disclosure - an attacker gets an access to the classified information
- Denial of Service (a.k.a. DoS) - make a server unavailable for the legitimate users, which often is implemented by generating a large number of simultaneous requests to saturate the server.
- Elevation of privilege - gaining an elevated access to the data, e.g. by obtaining administrative rights.



While we've been working on the section describing Apple's developers certificates (Chapter 14) their Web site was hacked and was not available for about two weeks.

One of the OWASP guides is titled [Web Application Penetration Testing](#). In about 350 pages it explains the methodology of testing a Web application for each vulnerability. OWASP defines *penetration test* as a method of evaluating the security of a computer systems by simulating an attack. Hundreds of security experts from around the world have contributed to this guide. Running penetration tests should become a part of your development process, and the sooner you start running them the better.

For example, the Payment Card Industry published a Data Security Standard, which includes a [Requirement 11.3](#) of penetration testing.

## Regulatory Compliance and Enterprise Security

So far in this chapter we've been discussing security vulnerabilities from the technical perspective. But there is another aspect that can't be ignored - the regulatory compliance of the business you automate.

During the last four years the authors of this book develop, deploy, support, and market the software that automates certain workflows for insurance agents. We serve more several hundreds of insurance agencies and more than 100K agents. In this section we'll share with you our real-world experience of dealing with security while running our company, which sells software as service. In addition to developing the application we had to set up the data centers and take care of security issues too.

Our customers are insurance agencies and carriers. We charge for our services, and our customers pay using credit cards using our application. This opens up a totally different category of security concerns:

- Where the credit card numbers are stored?
- What if they get stolen?
- How secure is the payment portion of your application?
- How the card holder's data is protected?
- Is there a firewall protecting customer's data?
- How the data is encrypted?

One of the first questions our perspective customers ask if our application is *PCI compliant*. They won't work with us until they review the *application-level security* implemented in our system. As per the [PCI Compliance Guide](#), "the Payment Card Industry Data Security Standard is used by all card brands to assure the security of the data gathered while an employee is making a transaction at a bank or participating vendor".

If your application stores PCI data, authenticating via Facebook, Google or a similar OAuth service won't be an option. The users will be required to authenticate themselves by entering long passwords containing combinations of letters, numbers and special characters.

Even if you are not dealing with the credit card information, there are other areas where the application data must be protected. Take the human resources application - social security numbers (unique ID's of the USA residents) of employees must be encrypted.

Some of our perspective customers send us a questionnaire to see if our security measures are compliant with their requirements. In some cases this document can include as many as 300 questions.

You may want to implement different levels of security depending on what devices is being used to access your application - a public computer, an internal corporate computer, iPad or an Android tablet. If a desktop user forgot his password, you may implement a recovery mechanism that send an email to the user and expects to receive a certain response from him. If the user holds a smartphone, the application can send a text message to his device.

If the user's record contains both his email and the cell phone number, the application should ask where to send the password recovery instructions. If a mobile device runs the hybrid or native version of the application, the user can be automatically switched to a messaging app of the device so he can read the text message while the main application remains at the view where authentication was required.

In the enterprise Web applications more than one layer of security must be implemented: at the communication protocol level, at the session level, and at the application level. The HTTP server [Nginx](#) besides being a high-performance proxy server and load balancer can serve as a security layer too. Your Web application can offload authentication tasks and validation of SSL certificates to Nginx.

Most of the enterprise Web applications are deployed on the cluster of servers, which adds another task to your project plan: how to manage sessions in a cluster. The user's session has to be shared between all servers in a cluster. High-end application servers may implement this feature out of the box. For example, IBM WebSphere server has an option to tightly integrate HTTP sessions with its application security module. Another example is Terracotta Cluster, which has the Terracotta Web Sessions module that allows session to survive the nod hops and failures. But small or mid-size applications may require some custom solutions for distributed sessions.



Minimize the amount of data stored in the user's session to simplify session replication. Store the data in the application cache, that can be replicated quickly and efficiently using open source or commercial products (e.g. JGroups, Terracotta et al).

Here's another topic to consider: multiple data centers when each one runs a cluster of servers. To speed up the disaster recovery process, your Web application has to be deployed in more than one data centers located in different geographical regions. The user authentication must work even if one of the data centers becomes not operational.

An external computer (e.g. Nginx server) can perform token-based authentication, but inside the system the token is used only when the access to protected resources is required. For example, when the application need to process a payment, it doesn't need to know any credit card details - it just uses the token to authorize the transaction of the previously authenticated user.

This grab bag of security considerations mentioned in this section is not a complete list of security-related issues that your IT organization needs to take care of. If you work for a large enterprise on the Intranet applications, these security issues may not sound as overly important. But as soon as your Web application starts serving external Internet users, someone has to worry about potential security holes that were not in the picture for internal applications. Our message to you is simple: "Take security very seriously if you are planning to develop deploy, and run a production-grade enterprise Web application".

## Summary

Every enterprise Web application has to run in a secure environment. The mere fact that the application runs inside the firewall doesn't make it secure. First, if you're opening at least one port to the outside world, a malicious code can sneak in. Second, there can be an "angry employee" or just a "curious programmer" inside the organization who can inject the unwanted code.

The proper validation of the received data is very important. Ideally, use the *white list* validation to compare the user's input against the list of allowed values. Otherwise do a *black list* validation to compare against the keywords that are not allowed in the data entered by the user.

There is no way to guarantee that your application is 100% protected from security breaches. But you should ensure that your application runs in the environment with the latest available patches for known security vulnerabilities. For example, if your application includes components written in Java programming language, install **critical security patches** as soon as they become available.

With proliferation of clouds, social networks, and sites that offer free or cheap storage people lose control over security hoping that Amazon, Google or Dropbox will take care of it. Besides software solutions, software-as-a-service providers deploy specialized hardware - security appliances that serve as firewalls, perform content filtering, virus and intrusion detection. Interestingly enough, hardware security appliances are also vulnerable.

In any case, the end users upload their personal files without thinking twice. Enterprises are more cautious and prefer private clouds installed on their own servers, where they administer and protect data themselves. The users who access Internet from their mobile devices have little or no control of how secure their devices are. So the person in charge of the Web application has to make sure that it's as secure as possible.

## PART III

# Responsive Web Design and Mobile

BYOD stands for Bring Your Own Device. It became a new trend - many enterprises started allowing their employees to access corporate applications from personal tablets or smartphones.

CYOD stands for Choose Your Own Device - corporations let their employees to choose from a set of devices that belong to the enterprise. It's about selecting a strategy that organizations should employ while bringing new devices.

Developers of new Web applications should always think of the users that will try to run this application on a mobile device. This part of the book is about various strategies of developing Web applications that look and perform well not only on the desktop computers but on a smaller screens too.

Today most of the enterprise applications are still being developed for desktop computers. The situation is changing, but it's a slow process. If five years ago it would be close to impossible to get a permission to bring your own computer to work and use it as for work related activities, the situation is a lot better now with BYOD and COYD.

Sales people want to use tablets while dealing with prospective clients. Business analysts want to be able to run familiar Web applications on their smartphones. Enterprises want to offer access to their valuable data to external clients from a variety of different devices.

In Chapter 10 we'll explain what the *Responsive Web Design* is and how you can build an HTML5 application that will have a single code base for desktops, tablets and smartphones. We'll apply the Responsive Design principles and re-design our Save The Child application to have **fluid layout** so it'll remain usable on smaller screens too.

Another approach is to have separate versions of the application for the desktops and mobile devices. Chapter 11 and 12 will demonstrate how to create dedicated mobile versions of Web application with jQuery Mobile library and Sencha Touch framework respectively. And the Save The Child application will get re-written in each of these chapter.

But if using Responsive Web Design (RWD) allows to have a single code base for all devices, you may be wondering, why not just build every Web application this way? RWD works fine for site that mainly publish information. But if the users are expected not just read, but also input some data on the small-screen devices, the UI and the navigation may need to be custom-designed to include only partial functionality where each page view is carefully designed to provide the best user experience. Besides, with responsive design the entire code and CSS for all devices is loaded to the user's smartphone making the application unnecessary heavy and slow when the connection speed is not great.

With small screens you have to re-think carefully what are the *must have* widgets and what functionality is crucial for the business you're creating a Web application for. If it's a restaurant, you need to provide an easy way to find the menu, phone, address, and directions to your place. If it's a site to collect donations like our Save The Child the design should provide an easy way to donate, while the rest of the information should be hidden by simple navigational menus.

In rare occasions an enterprise application is created solely for the mobile platform. More often the task is to migrate an existing application to mobile platform or develop of separate versions of the same application for desktops and mobile devices. If a decision is made to develop native mobile applications then the choice of the programming languages is dictated by the mobile hardware.

If it's a Web application then using the same library or a framework for desktop and mobile platforms may shorten the development cycle. That's why we decided to cover such pairs in this book, namely:

- jQuery and jQuery Mobile
- Ext JS and Sencha Touch

But even though each of these pairs shares the same code for the core components, do not expect to be able to kill two birds with one stone. You are still going to use different version of the code, for example, jQuery 2.0 and jQuery Mobile 1.3.1. This means that you may have to deal with separate bugs that might have sneaked into the desktop and mobile version of the frameworks.

What's better jQuery Mobile or Sencha Touch? There is no general answer to this question. It all depends on the application you're building. If you need a simple mobile application for displaying various information (a publishing type of application), then

jQuery Mobile will do the job with least efforts. If you are building an application that requires some serious data processing Sencha Touch is a better choice. Of course, there are lots of other frameworks and libraries that can help you with developing a mobile Web application. Do your homework and pick the one that fits your needs best.

There's a [Web site](#) that compares mobile frameworks. It even has a little wizard application that allows you to pick a framework that meets your needs and is supported on required devices. [Figure III.1](#) is a fragment snapshot from this site. As you can see, jQuery Mobile supports the largest number of different platforms.

Framework	Platform (Rendering Engine)											
	iOS (Webkit)	Android (Webkit)	Windows Mobile (Trident)	Windows Phone (Trident)	Blackberry OS (Webkit)	Symbian (Webkit/Gecko)	MeeGo (Gecko)	Maemo (Gecko)	WebOS (Webkit)	Bada (Webkit)	Java ME	
Apache Flex	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	
Appcelerator Titanium Mobile*	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	
iUI*	✓	✓	✗	✗	✓	✗	✓	✗	✓	✓	✗	
iPhone Universal*	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	
CNET iPhone UI*	✓	✓	✗	✗	✓	✗	✓	✗	✓	✓	✗	
GWT mobile webkit + gwt mobile ui*	✓	✓	✗	✗	✗	✗	✗	✗	✓	✗	✗	
Jo HTML5 Mobile App Framework*	✓	✓	✗	✗	✗	✓	✗	✗	✓	✗	✗	
JQ Touch*	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	
Phone Gap*	✓	✓	✗	✗	✓	✓	✗	✗	✗	✓	✗	
iQuery Mobile*	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	
Quick Connect*	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	
mobione*	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	
Rhodes*	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	
Sencha Touch*	✓	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	

*Figure III.1. A fragment of the Mobile Frameworks Matrix*



There is a framework called **Zepto**, which is a minimalist JavaScript library with the API compatible to jQuery. Zepto supports both desktop and mobile browsers.

Finally, in Chapter 13 will talk about yet another approach for developing HTML5 applications for mobile devices - so called *hybrid* applications. These are the applications that are written in JavaScript, but are packaged as native apps. You'll learn how Adobe's PhoneGap can package an HTML5 application to be accepted in online stores where native applications are being offered. To illustrate accessing hardware features of a mobile device we'll show you how to access a photo camera of the mobile device - this can be a useful feature for the Save The Child application.

# Responsive Design: One Site Fits All

Up till now we've been writing and re-writing the desktop version of the Save The Child application. Will it look good on the small screen of a mobile device? Starting from this chapter we'll deal with the mobile devices too.

Let's discuss different approaches to developing a Web application that can work on both desktop and mobile devices. There are three choices:

1. In addition to your Web application that works on desktops develop a separate version of the native applications for multiple mobile devices. Development of native mobile applications is not covered in this book.
2. Develop a single HTML5 Web application, but create several different UI layouts that will be applied automatically based on the screen size of the user's device.
3. In addition to your Web application that works on desktops develop a *hybrid application*, which is a Web application on steroids - it works inside the mobile browser, but is packaged as native app and can invoke native API of the mobile device too. Chapter 13 is dedicated to hybrid applications.

This chapter is about the second approach called *Responsive Design*. This term was coined by Ethan Marcotte in the article [Responsive Web Design](#). The design of the Web page changes responding (reacting) to the user's display size. We'll modify the design of the Save The Child site to introduce different layouts for the desktop, tablet, and smart phones. By the end of this chapter the site Save The Child will automatically change its layout (without losing functionality) based on the screen size of the user's device.

## One or Two Versions of Code?

Run any of the versions of our Save The Child from the first chapters of the book on your desktop and start dragging the right border of the browser's window to make it

narrower. At some point you'll see only a part of the content - those layouts of the Save The Child was not meant to be responsive. It defined fixed sizes for the page sections, which didn't change even if the display area shrinks.

Enter the address of the book supporting site **Save The Child** in your mobile phone's browser. Select the version titled HTML/AJAX. Either you'll see a partial content of the page or the entire page with illegible small fonts as in **Figure 10-1**. Such design of the Save The Child application doesn't look good on all devices.

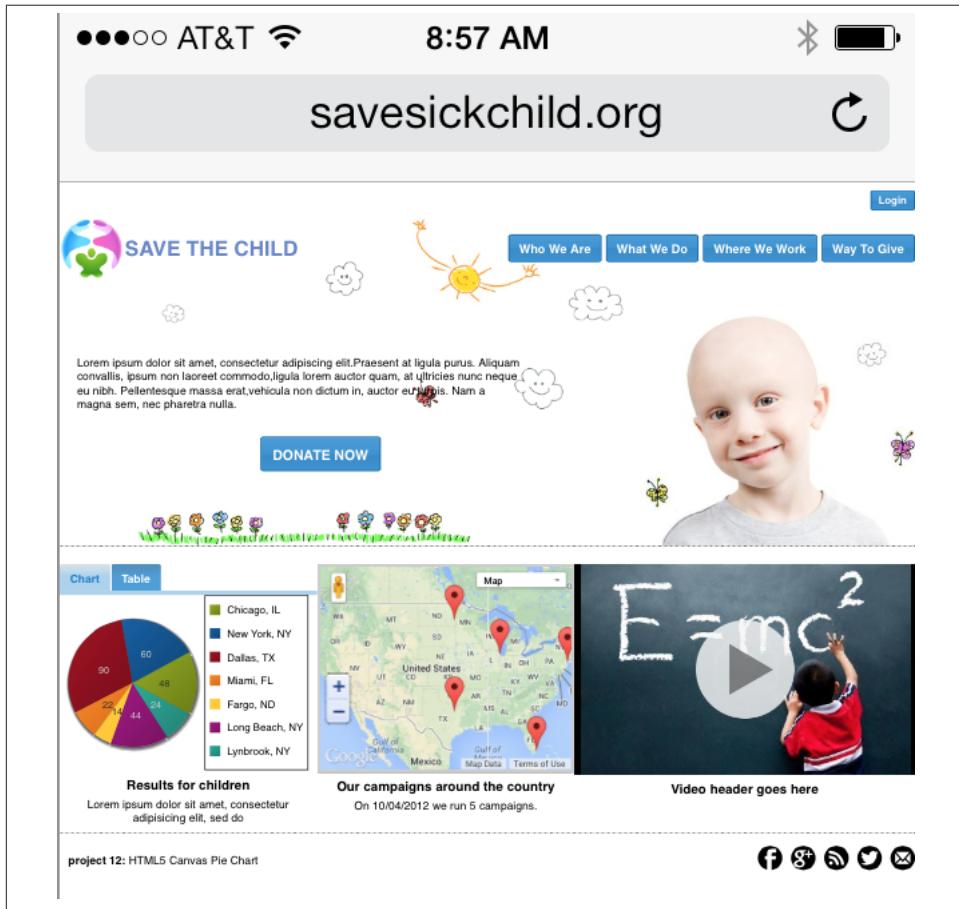


Figure 10-1. Non-responsive version of the app on iPhone 5

Now try the version titled Responsive Design - looks more usable on a small screen. How many versions of the UI do we need to create then? People responsible for developing a Web applications that can run on both desktop and mobile platforms usually start with making an important decision: HTML5 or native? But even if a decision was

made in the favor of the Web platform, the next question is if the desktop and mobile clients will use the same code or not.

If a decision is made to go with separate versions of the Web applications, the Web server can be configured to perform the redirection to the appropriate code depending on the type of the user's device. Web servers can do it based on the value of the `User-Agent` attribute of the HTTP request header. For example, the mobile Web browsers trying to access of [BBC](#) (or any other Web page) report their `User-Agent` to the server differently from desktop computers hence they receive different content delivered from a different URL. The snapshots [Figure 10-2](#) and [Figure 10-3](#) of the BBC main page were taken at the same time, but [Figure 10-2](#) shows how the page looks on the desktop computer, while [Figure 10-3](#) was taken from the iPhone.



The Safari Browser has a menu Develop, where you can select different User Agents to see how the current Web page will look on different Web browsers. You can also copy/paste a User Agent string from the site [useragentstring.com](http://useragentstring.com) to have an idea how the Web page will look like in hundreds of devices if the Web site is user-agent driven.

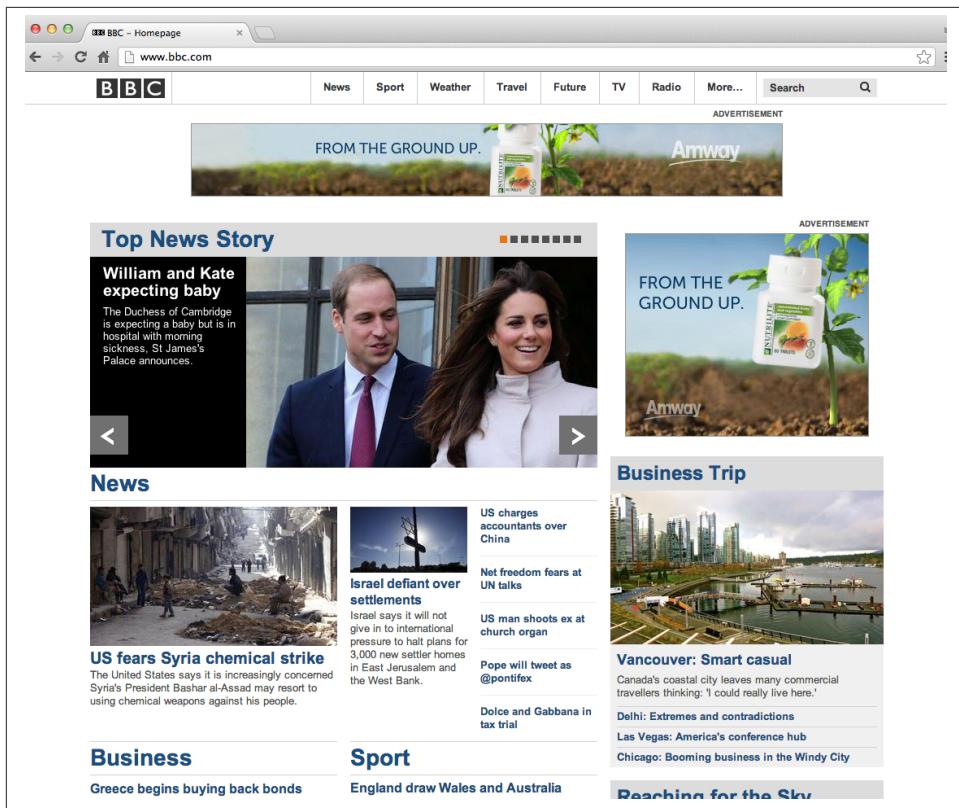


Figure 10-2. The desktop version of [bbc.com](http://bbc.com)

The Figure 10-2 page layout delivers more content as it can be allocated nicely on the large desktop monitor or a tablet. But the mobile version on Figure 10-3 substantially limits what's delivered to the client, which is done not only because the screen is small, but the user may be accessing the page over the slower network.

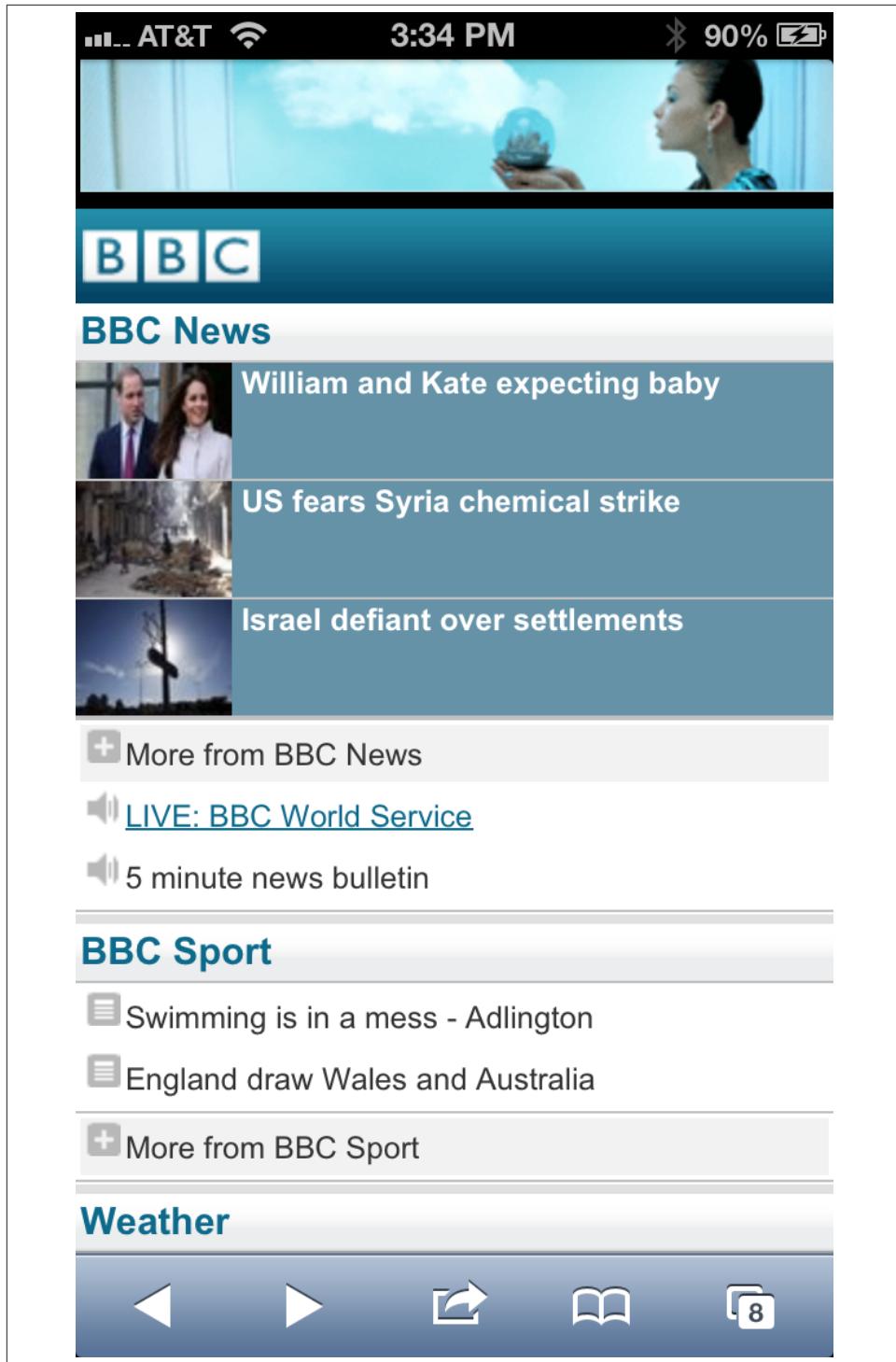


Figure 10-3. The mobile version of [bbc.com](http://bbc.com)

Have you ever tried to share the link of a Web site a specifically designed for smart-phones? It's so easy! Just press the button and enter the email of the person to share the site with. Many mobile Web sites shared this way won't look pretty on the large screen. It may just show the wider version of what you see in your mobile screen.

Maintaining two different versions of the application code requires more efforts than maintaining one: you need to have two sets of HTML, CSS, JavaScript, and images. Besides, most likely your Web application will use a third-party JavaScript framework. At some point you may run into a bug and will need to upgrade the mobile version to use the latest version of, say jQuery framework. But the desktop version works just fine. In case of having two separate versions of the application you'll have to either upgrade jQuery and thoroughly test both mobile and desktop versions of Save The Child, or live with two different versions of the framework.

Responsive design allows you to create one version of the Web application, which includes multiple sections of CSS controlling page layouts for different screen sizes. In this chapter we'll create yet another version of the Save The Child application that will render UI differently on desktop and mobile devices. All these version will share the same HTML and JavaScript code, but will include several versions of styling using CSS *media queries*.

There is a number of Web sites that were built using responsive design. Visit the following Web sites first from the desktop computer and then from smart phones (or just lower the width of the desktop browser window) to experience such fluid responsive design:

1. [Boston Globe](#)
2. [Mashable](#)
3. [Cafe Evoke](#)
4. [Fork CMS](#)
5. [a lot more examples](#)

Note that each of these Web pages displays the content on the desktop in three different layouts (often in three, four, six, or twelve imaginary columns). As you make the window narrower, the layout will automatically switch to the tablet or a large smartphone mode (usually two columns layout), and then to the phone mode layout (the one column layout).

This sounds like a great solution, but if you put all your media queries in the same CSS files, your users will be downloading unnecessary bytes - the entire CSS file that includes all versions of screen layouts. This is not the case in the BBC example, which has different versions of the code that load only what's necessary for a particular device category.

You can have several CSS files for different devices. Include these files using the media attributes. But Web browsers were not designed to selectively download only those CSS that are needed. For example, the following HTML will load both CSS files (**without blocking rendering**) on any user's device:

```
<link media="only screen and (max-width: 480px)" href="css/mobile.css" rel="stylesheet">  
<link media="only screen and (max-width: 768px)" href="css/tablet.css" rel="stylesheet">
```



Using the `Window.matchMedia` attribute may allow you to **conditionally load** CSS in JavaScript. There JavaScript utility `eCSSential` can help Web browser to download CSS faster.



Consider combining Responsive Design on the client with some device-specific components (a.k.a. **RESS**) optimization on the server.

While Responsive Design allows to re-arrange the content based on the screen size, it may not be a good idea to show the same amount of content on desktop and smartphones. Making a Web application look good on mobile devices must involve not only Web designers and developers, but also people who are responsible for content management.

Now comes the million dollar question, “Do we need to create two different versions of the Web application or twenty two? Why not two hundred and twenty two?” How many different mobile devices are there today and will be there tomorrow?

## How Many User Agents Are There

The HTTP header's attribute `User-Agent` contains information about the user agent originating request. Should you decide to create several versions of the UI based on the value in the `User-Agent` field, you can refer to the Web site <http://useragentstring.com>. It lists not two, but hundreds of strings representing possible content of the `User-Agent` attribute for a variety of desktop and mobile devices. For example, **Figure 10-4** shows how the `User-Agent` string from iPhone 5 is reported and explained by [useragentstring.com](http://useragentstring.com). But this information might become unreliable after iOS upgrades.

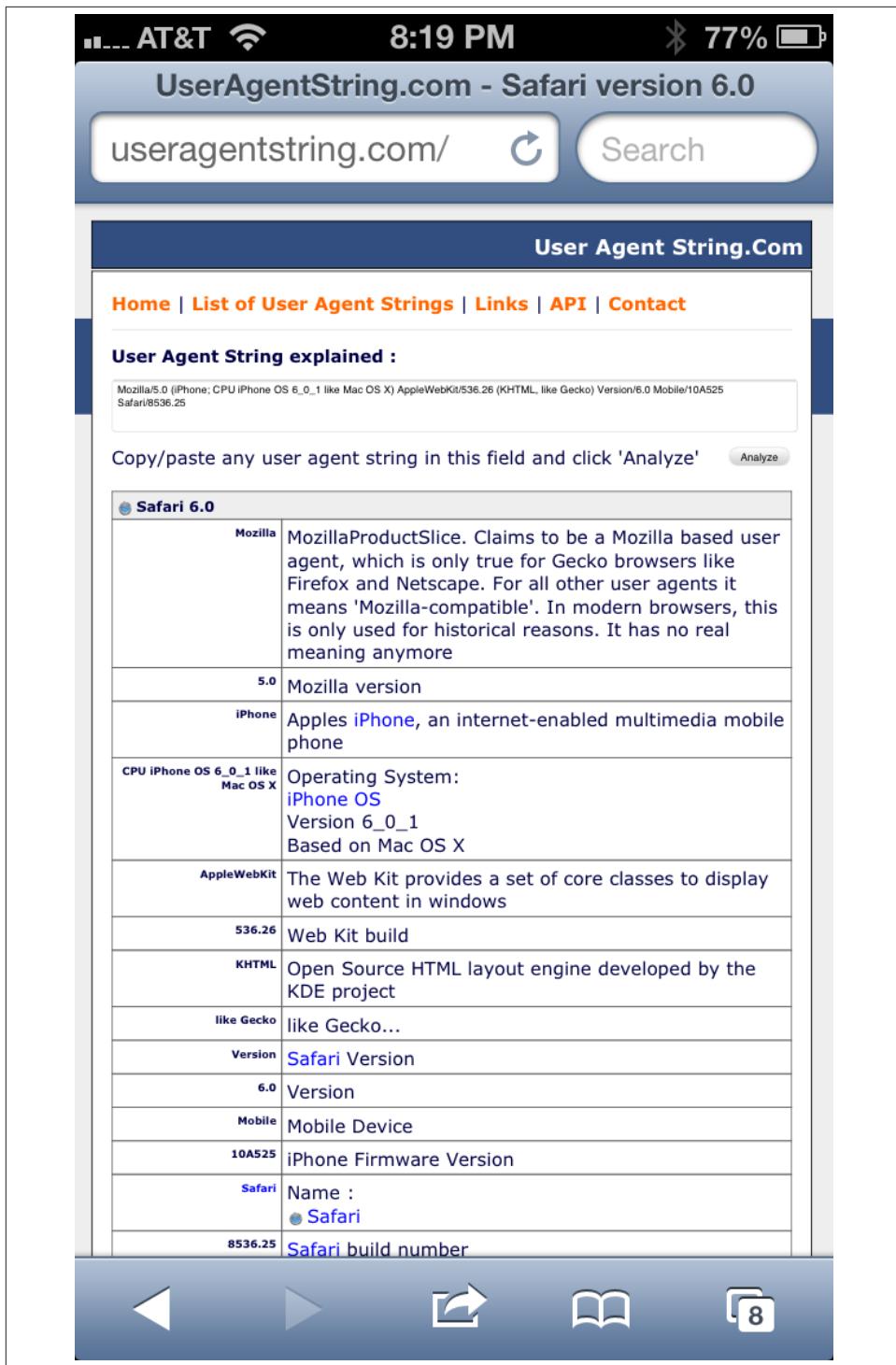


Figure 10-1 The Responsive Design Site for All iPhone 5

There is an easier way to detect on the server that the request came from a mobile device. **Wireless Universal Resource File (WURF)** is a database of thousands of supported devices and their properties. Such Internet giants as Facebook and Google rely on this service and your application could too, if need be. WURF offers APIs from several programming languages to detect specific capabilities of the user devices. For example, the following code snippet is how you could access the WURF data from a Java servlet.

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    WURFLHolder wurfl = (WURFLHolder)getServletContext()
        .getAttribute(WURFLHolder.class.getName());

    WURFLManager manager = wurfl.getWURFLManager();

    Device device = manager.getDeviceForRequest(request);

    log.debug("Device: " + device.getId());
    log.debug("Capability: " + device.getCapability("preferred_markup"));
}
```

It's impossible to create different layouts of a Web application for thousands of user agents. Market fragmentation in the mobile world is a challenge. People are using 2500 different devices to connect to Facebook. Android market is extremely fragmented. The Figure [Figure 10-5](#) is taken from the report [Android Fragmentation Visualized \(July 2013\)](#) by Open Signal.

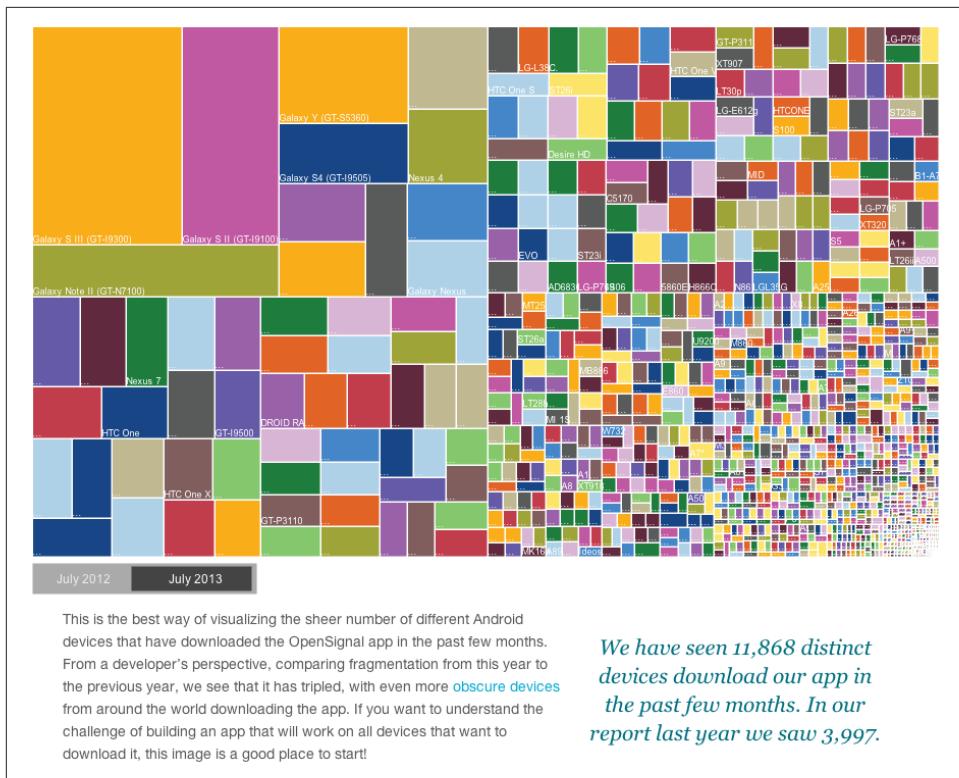


Figure 10-5. Android Device Fragmentation

Of course, device fragmentation doesn't equal Android OS version fragmentation, but this situation is similar to the challenge that Microsoft was always facing - making sure that Windows works fine on thousands different types of hardware. It's not an easy job to do. In this regard Apple is in much better position because they are the only hardware and software vendor of all devices running iOS.

It's great for the consumers that Android can be used on thousand devices, but what about us, the developers? Grouping devices by screen sizes may be a more practical approach for lowering the number of UI layouts supported by your application. The responsive design is a collection of techniques based upon these main pillars:

1. CSS *media queries*
2. *Fluid grids* or fluid layouts
3. Fluid media



Typography can be also considered as one of the pillars of the responsive design. This subject belongs to publications written for Web designers and will not be covered in this book. Oliver Reichenstein's article [Responsive Typography: The Basics](#) is a good introduction to this topic.

**Media query** is a CSS element. It consists of a media type (e.g. `@media (min-width: 700px)` and `(orientation: landscape)`) followed by the styles applicable to this media. Media queries allow to rearrange the sections (like `<div>`, `<section>`, `<article>` et al.) of the page based on the screen size, fluid grids allows to properly align and scale the content of these sections, and the fluid media is about resizing images or videos.

Data grid components are often included in enterprise applications. Fluid grids are designed using relative positioning and can scale based on the screen sizes. Fluid media is about creating videos and images that react to the screen sizes. We'll talk about the above pillars in greater details later in this chapter. But before going into technical details, let's get back to the mockups to see how the UI should look like on different devices.

## Back to Mockups

Jerry, our Web designer came up with another set of Balsamiq mockups for the Save The Child application. This time he had four different versions: desktop, tablet, large phone, and small phone. As a matter of fact, Jerry provided more mockups - the user can hold both smartphones and tablets either in portrait or landscape mode. [Figure 10-6](#), [Figure 10-7](#), and [Figure 10-8](#), [Figure 10-9](#) show the screenshots taken from Balsamiq Mockups for desktop, tablet, large, and small phone layouts. [Figure 10-6](#) shows the desktop mockup.

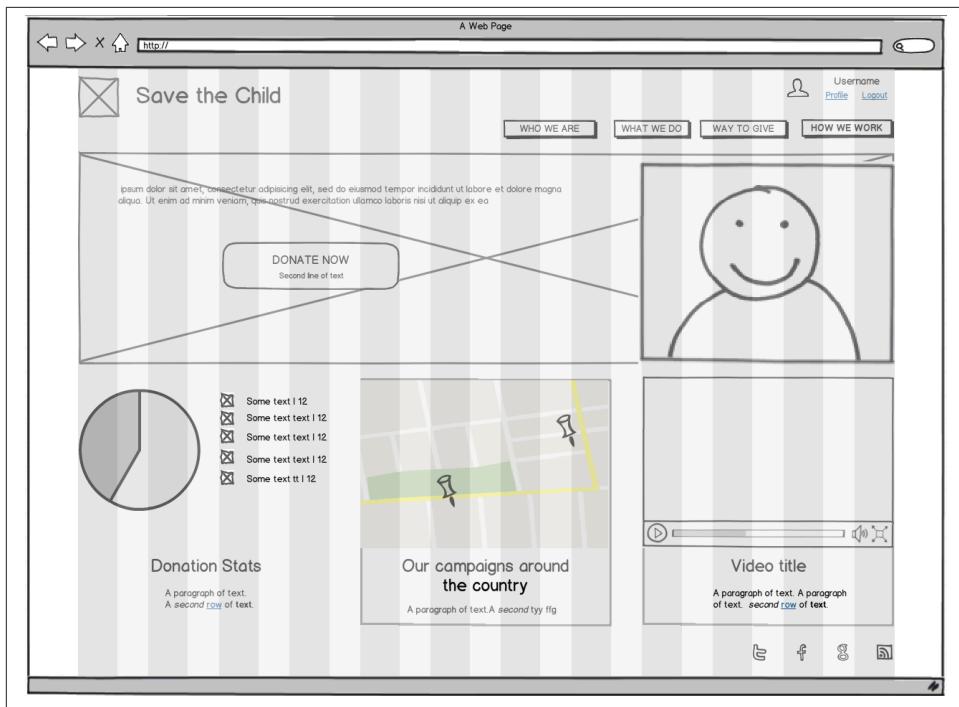


Figure 10-6. The Desktop layout

Jerry gave us several versions of the images - with and without the grid background. The use of the grid will be explained later in the section “Fluid Grids”. **Figure 10-7** depicts the rendering on tablet devices that fall in a category of under 768px width screen in the portrait mode.

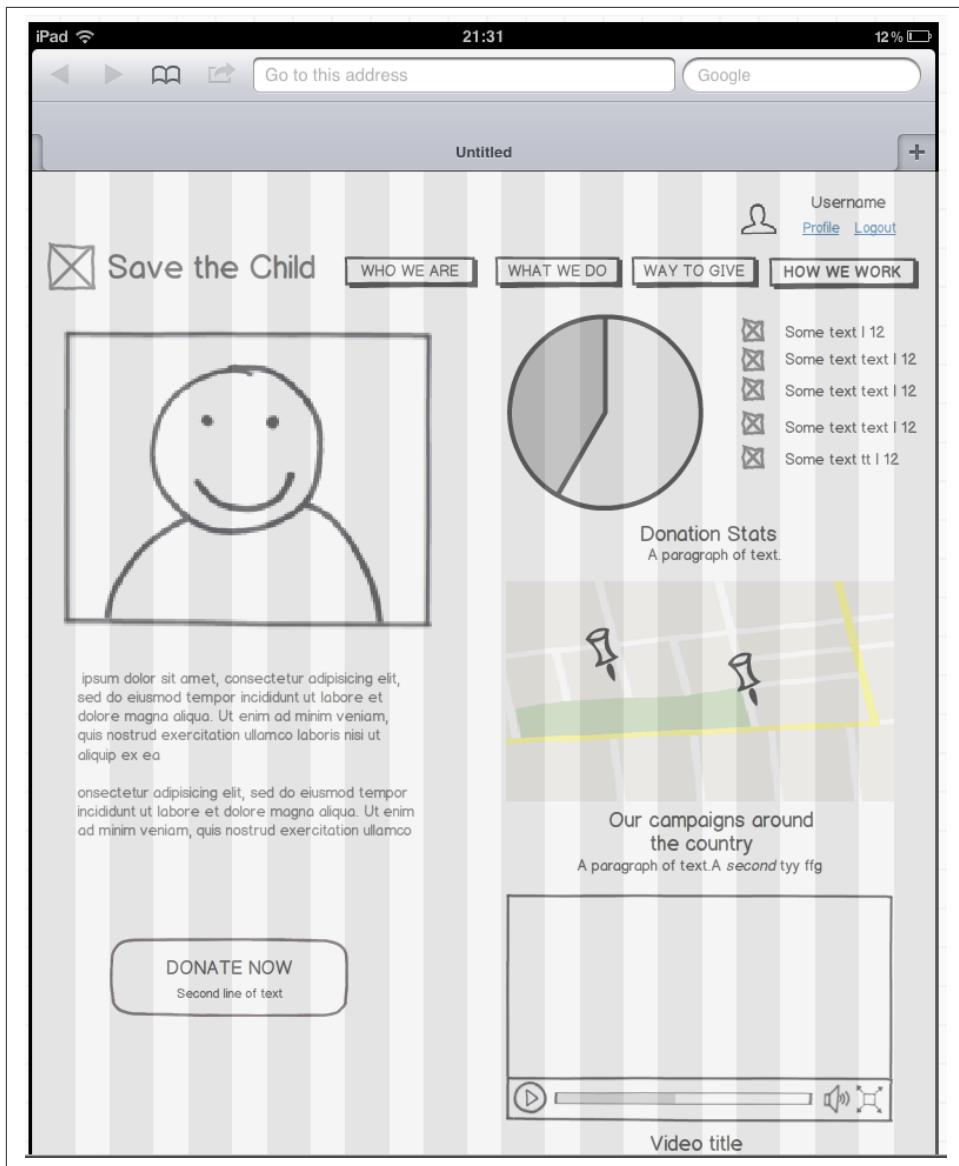


Figure 10-7. The tablet layout (portrait)

Next comes the mockup for the large smart phones having the width of up to 640 pixels. **Figure 10-8** shows two different images of the screen next to each other (the user would need to scroll to see the second image).

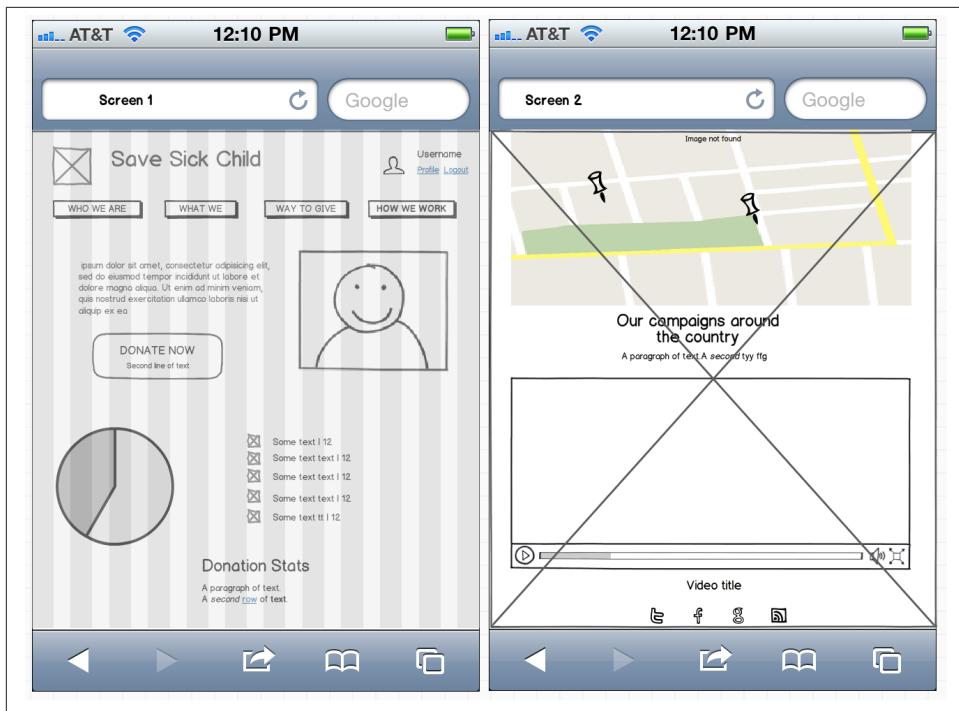


Figure 10-8. The large phone layout (portrait)

The mockup for the smaller phones with the width of under 480 pixels is shown on [Figure 10-9](#). The mockup looks wide, but it actually shows three views of the phone screen next to each other. The user would need to scroll vertically to see the middle or the right view. iPhone 3 falls into this category.

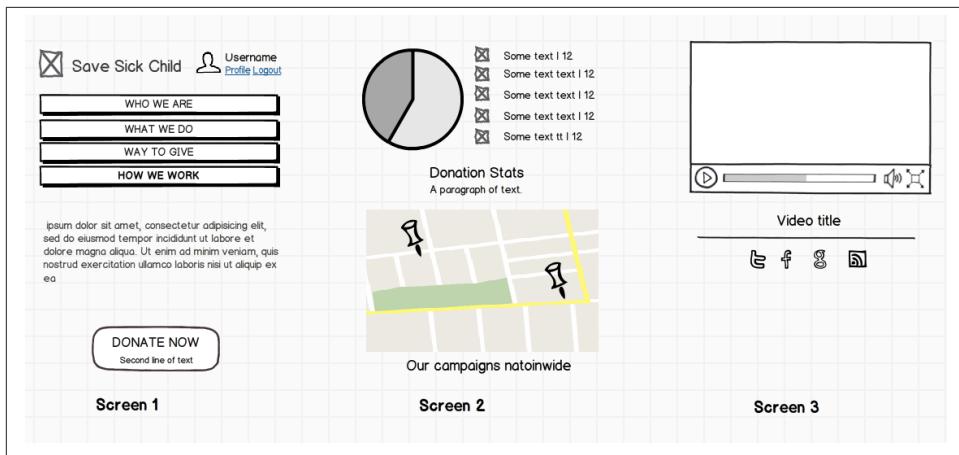


Figure 10-9. The small phone layout (portrait)

If need be, you can ask Jerry to create mockups for the real devices with the width under 320 pixels, but we won't even try it here. Now we need to translate these mockups into working code. The first subject to learn is CSS media queries.

## CSS Media Queries

First, let's see the CSS media queries in action, and then we'll explain how this magic was done. Run the project titled `Responsive_basic_media_queries`, and it'll look as in **Figure 10-10**. This is a version for the desktops (or some tablets in the landscape mode). The section chart, map, and video divide the window into three imaginary columns.

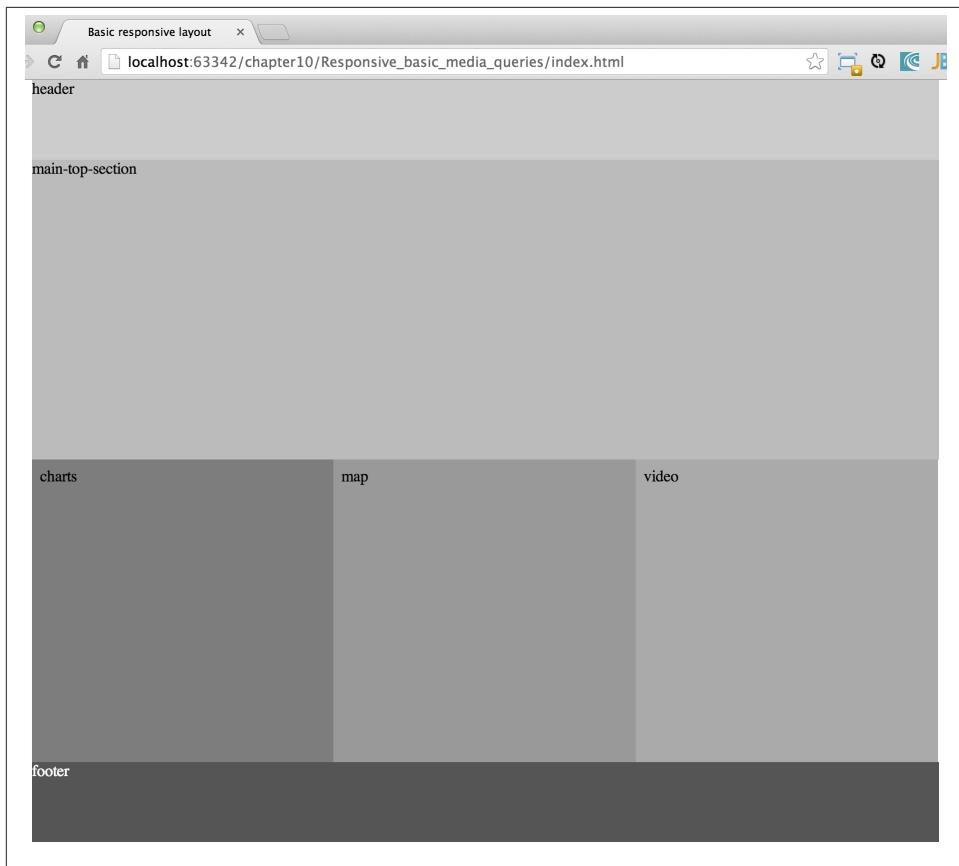


Figure 10-10. The desktop layout implemented

Drag the right border of your desktop Web browser's window to the left to make it narrower. After reaching certain *breakpoint width* (in our project it's 768 pixels) you'll see how the `<div>`s reallocate themselves into the two-column window shown on [Figure 10-11](#).

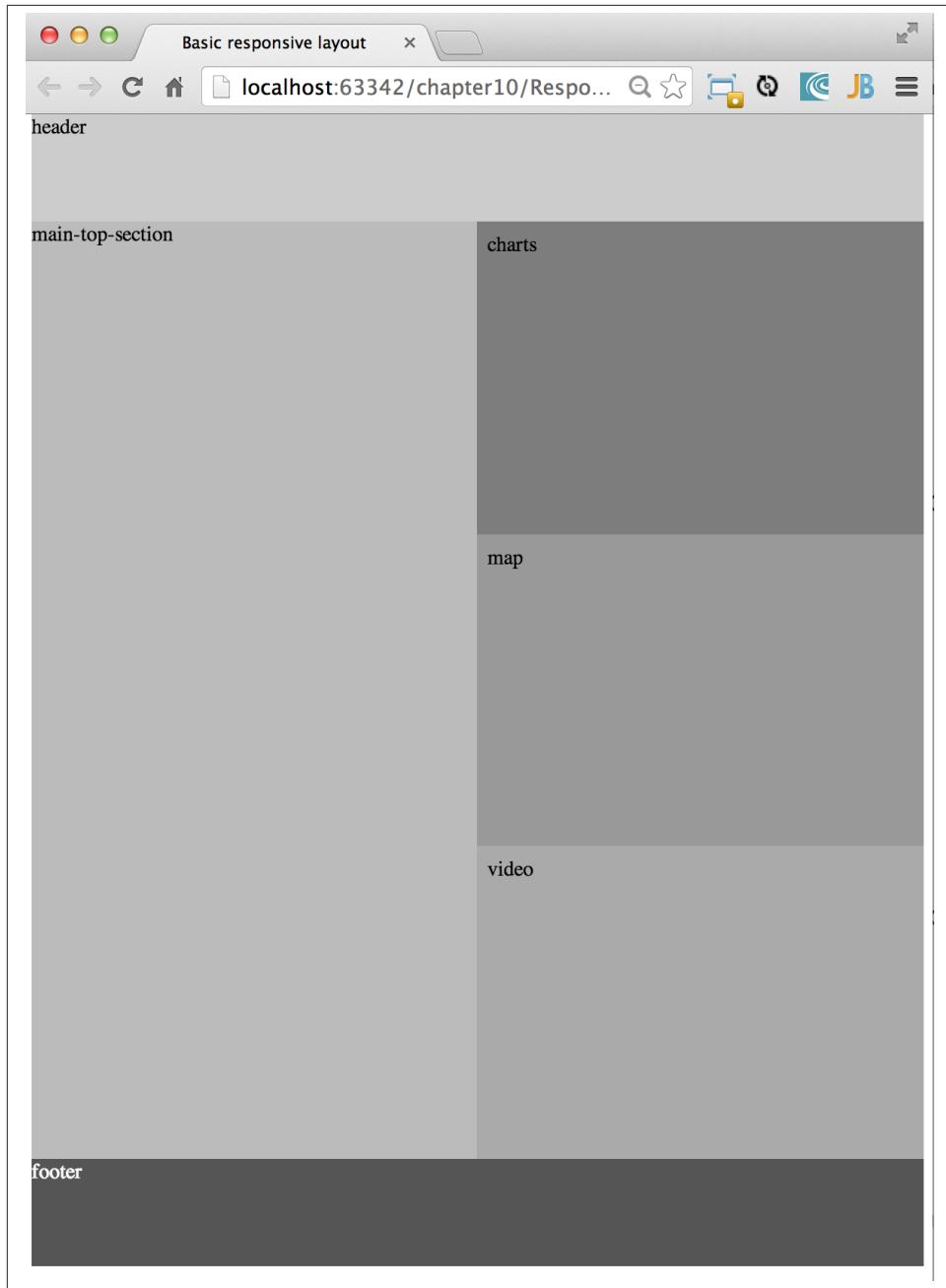


Figure 10-11. The tablet layout (portrait) implemented

Keep making the browser's window narrower, and when the width will pass another breakpoint (becomes less than 640 pixels), the window will re-arrange itself into one long column as in [Figure 10-12](#). The users will have to use scrolling to see the lower portion of this window, but they don't lose any content.

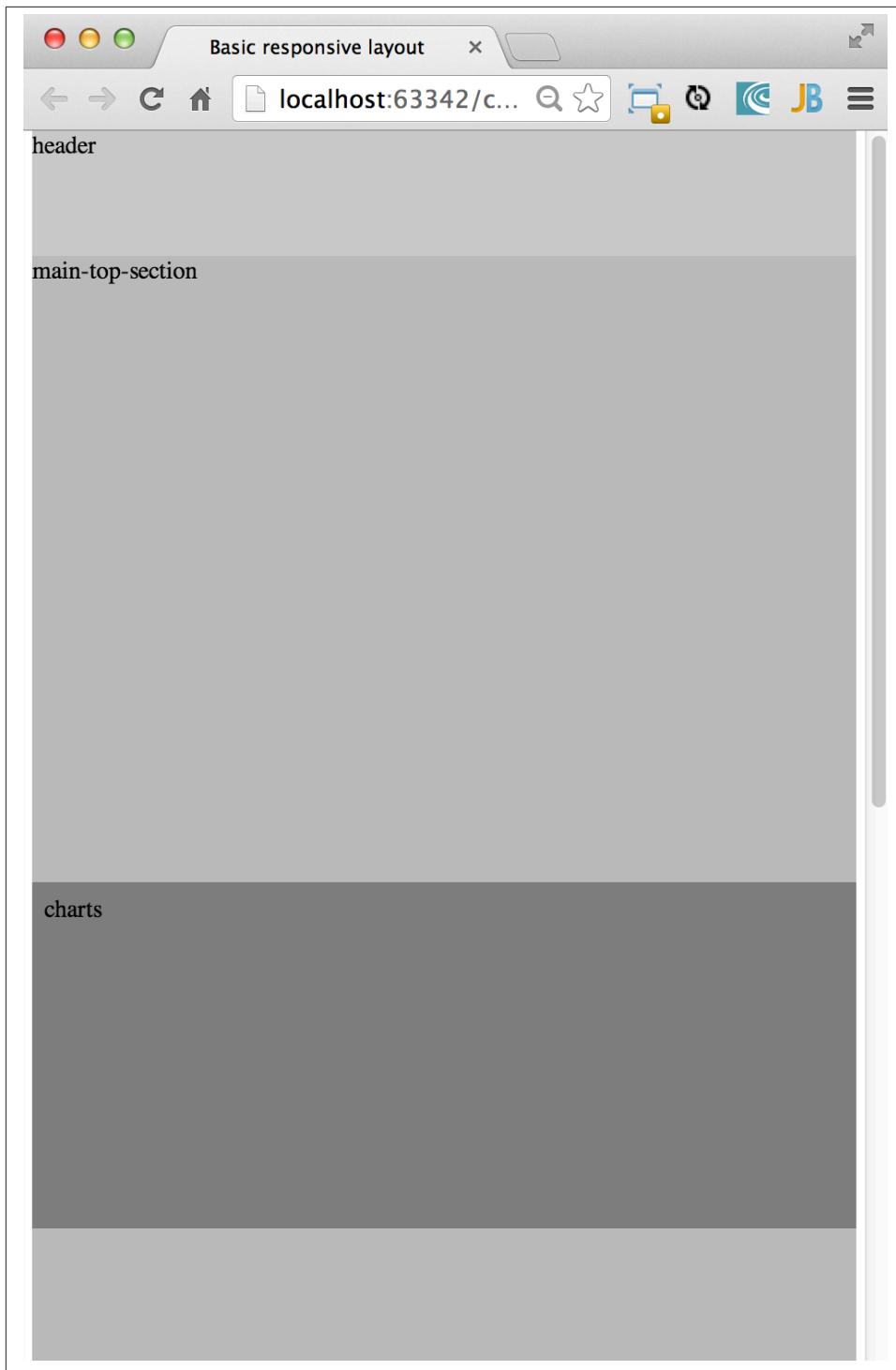


Figure 10-12. The smaller phone layout (portrait) implemented CSS Media Queries | 385

The W3C Recommendation titled [Media Queries](#) has been introduced in CSS2 and HTML4. The idea was to provide different stylesheets for different media. For example, you can specify different stylesheets in HTML using the `media` attribute for the screens that are less than 640 pixel in width.

```
<link rel="stylesheet" href="assets/css/style.css" media="screen">

<link rel="stylesheet" href="assets/css/style_small.css"
      media="only screen and (max-width: 640px)">
```

You may have several of such `<link>` - tags for different screen widths. But all of them will be loaded regardless of the actual size of the user's display area. The modern browser may defer loading of the CSS files that don't match the current display size.

The other choice is to specify a section in a CSS file using one or more `@media` rules. For example, the following style will be applied to the HTML element with the `id=main-top-section` if the width of the display area (screen) is less than 640 pixels. Screen is not the only media type that you can use with media queries. For example, you can use `print` for printed documents or `tv` for TV devices. For the up to date list of media types see the document [Media Queries W3C Recommendation](#).

```
@media only screen and (max-width: 640px) {

    #main-top-section {
        width: 100%;
        float: none;
    }
}
```

Two fragments of the CSS file styles.css from the project Responsive\_basic\_media\_queries are shown next. The first one starts with defining styles for windows having 1280px width (we use 1140 pixels to leave some space for padding and browser's chrome). Here's the first fragment:

```
/* The main container width should to be 90% of viewport width but not wider than 1140px */
#main-container {
    width: 90%;
    max-width: 1140px;           // ①
    margin: 0 auto;
}

/* Background color of all elements was set just as an example */
header {
    background: #ccc;
    width: 100%;
    height: 80px;
}

#main-top-section {
    background: #bbb;
```

```

        width: 100%;
        height: 300px;
        position: relative;
    }

#main-bottom-section {
    width: 100%;
}

#video-container, #map-container, #charts-container {
    width: 33.333%; // ②
    padding-bottom: 33.333%; // ③
    float: left;
    position: relative;
}

#video, #map, #charts {
    background: #aaa;
    width: 100%;
    height: 100%;
    position: absolute;
    padding: 0.5em;
}

#map {
    background: #999;
}

#charts {
    background: #7d7d7d;
}

footer {
    background: #555;
    width: 100%;
    height: 80px;
    color: #fff;
}

```

- ➊ Set the maximum width of the window on a desktop to 1140 pixels. It's safe to assume that any modern monitor supports the resolution of 1280px width (minus about 10% for padding and chrome).
- ➋ Allocate one third of the width for video, charts, and maps each.
- ➌ Float left instructs the browser to render each of these divs starting from the left and adding the next one to the right.

This CSS mandates to change the page layouts if the screen size is or becomes below 768 or 640 pixels. Based on your Web designer's recommendations you can specify as many breakout sizes as needed. Say, in the future, everyone will have at least 1900px

wide monitor - you can provide a layout that would use five imaginary columns. This can be a good idea for online newspapers or magazines, but Save The Child is not a publication so we keep its maximum width within 1140px. Or you may decide to make a version of Save The Child available for LCDs of only 320px in width - create a new media query section in your CSS and apply fluid grids to make the content readable. Here's the second fragment of the CSS file that defines media queries.

```
/* media queries */

@media only screen and (max-width: 768px) {    // ①
    #main-container {
        width: 98%
    }

    #main {
        background: #bbb;
    }

    #main-top-section, #main-bottom-section {
        width: 50%;                      // ②
        float: left;                     // ③
    }

    #main-top-section {
        height: 100%;
    }

    #video-container, #map-container, #charts-container {
        float: none;                    // ④
        width: 100%;
        padding-bottom: 70%;
    }
}

@media only screen and (max-width: 640px) {    // ⑤

    #main-top-section, #main-bottom-section {
        width: 100%;                  // ⑥
        float: none;
    }

    #main-top-section {
        height: 400px;
    }

    #video, #map, #charts {
        height: 60%;
    }
}
```

- ❶ This media query controls layouts for devices with viewports having the max width of 768px.
- ❷ Split the width fifty-fifty between the HTML elements with ID's `main-top-section` and `main-bottom-section`.
- ❸ Allocate main-top-section and main-bottom-section next to each other (`float: left;`) as in [Figure 10-11](#). To better understand how the CSS `float` property works, visualize a book page having an small image on the left with the text floating on the right (a text wrap) - this is what `float: left;` can do on a Web page.
- ❹ Turn the floating off so the charts, maps, and video containers will start one under another as in [Figure 10-11](#).
- ❺ The media query controlling layouts for devices with viewports with the max width of 640px starts here.
- ❻ Let the containers main-top-section, main-bottom-section take the entire width and be displayed one under another (`float: none;`) as in [Figure 10-12](#).



Internet Explorer 8 and older don't natively support media queries. Consider using Modernizr to detect support of this feature, and load the [Media Queries Polyfill](#), if needed.

## The Viewport Concept

Mobile browsers use a concept of *viewport*, which is a virtual window where they render the Web page content. This virtual window can be wider than the actual width of the display of the user's mobile device. For example, by default iOS Safari and Opera Mobile render the page to the width of 980px, and then shrinks it down to the actual width (320px on old iPhones and 640px on iPhone 4 and 5). That's why your iPhone renders the entire Web page of, say New York Times (yes, the fonts are tiny), and not just its top left section.

By using the meta tag `viewport` your Web page overrides this default and renders itself according to the actual device size. All code samples in this chapter include the `viewport` meta tag in index.html. All mobile browsers support it even though it's not a part of the HTML standard yet. Desktop browsers ignore the tag `viewport`.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

This meta tag tells the browser that the width of the virtual viewport should be the same as the width of the display. This setting will produce good results if your responsive Web design includes a version of the page layout optimized for the width of the current user's

device. But if you'd be rendering a page with a fixed width, which is narrower than the default width of the display (e.g. 500 pixels) setting the attribute `content="width=500"` would instruct the mobile Web browser to scale the page to occupy the entire display real estate. In other words, setting a fixed width is like saying, "Dear mobile browser, I don't have a special layout for this device width - do the best you can and scale the content".

Setting the initial scaling to 1.0 ensures that the page will be rendered as close to the physical device size as possible. If you don't want to allow the user scale the Web page, add the attribute `user-scalable=no` to the meta tag `viewport`.



If you'll apply the initial scale to be 1.0, but to a Web page that was not build using responsive design principles, users will need to zoom or pan to see the entire page.

For details about configuring the viewport refer to [Apple's](#) or [Opera's](#) documentation.

Some of the important concepts to take away from this example are to switch from pixels to percentages when specifying width. In the next examples you'll see how to switch from using rigid `px` to more flexible `em` units. The CSS `float` property you can control relative (not absolute) positioning of your page components. There are also such CSS measure units as `vw` and `vh`, which represent percentages of the viewport width and height respectively. But the best practice here is to use [`rem` units](#). The app can set the font size on `BODY` and then specify everything in relative-ems that scale only from that number. `Em`s cascade their scale down from their parent, meaning lots of extra math the developer and the browser has to do.



Install an Add-On for Google Chrome called [Window Resizer](#). It'll add an icon to the toolbar for easy switch between the browser screen sizes. This way you can quickly test how your Web page looks on different viewports. There is another handy Add-On for Chrome called [Responsive Inspector](#), which allows you to see the various media queries for a page and automatically resize to them.



Google Chrome Developer Tools offer you a way to test a Web page on various emulators of mobile devices. you just need to check off the "Show Emulation view in console drawer" in Settings, and then you'll see the Emulation tab under the Elements menu (hit the Esc key if it's not shown).

## How Many Breakpoints?

How many media queries is too many? It all depends on the Web page you're designing. In the sample CSS shown in this section above we've used the breakpoint of 768px to represent the width of the tablet in the portrait mode, and this is fine for the iPad. But several tablets (e.g. 10.1" Samsung Galaxy) have 800px-wide viewport while Microsoft Surface Pro is 1080px wide.

There is no general rule as to how many breakpoint is needed for a typical Web page. Let the content of your page (and where it breaks) dictate where you add breakpoints. Just create a simple Lorem Ipsum prototype of your Web site and start changing its size. At a certain point (viewport size) your design starts to break. This is where you need to put your breakpoint and define a media query for it. It is recommended to start with designing for the smallest viewports (the "Mobile First" principle). As the viewport width increases you may decide to render more content hence define a new breakpoint. Technically this means that the content of your CSS should default to the smaller viewports and only if the screen is larger, apply media queries. Such approach will reduce the CSS handling by the browser of the mobile device (no need to switch from large to smaller layouts).



Use Google Chrome Developer Tools to find out the current width of the viewport. Just type in the console `window.innerWidth` and you'll see the width in pixels.

Don't try to create a pixel perfect layout using responsive design. Use common sense and remember, the more different media queries you provide the heavier your CSS file will become. But in mobile world you should try to create Web applications as slim as possible.

Warning: Be prepared to see inconsistencies among the desktop browsers in measuring the width of the viewport. Our tests showed that WebKit-based browsers add about 15px to the width, supposedly accounting for the width of the scrollbar. So if you have a media query that has to change the layout at 768px, it'll change it at about 783px. Do more testing on different viewports and adjust your CSS as needed.

## Fluid Grids

Fluid grids is a very important technique in the responsive design. Grids were used by Web designers for ages - a web page was divided by a number of imaginary rows and columns. But the fluid grid, as the name implies, is flexible and can scale based on the screen sizes.

## Moving Away From Absolute Sizing

When a browser displays text it uses its default font size unless it was overruled by the `font-size` property. Typically, the default font size is 16 pixels. But instead of using the absolute font size, you can use the relative one by using so called *em* units. The default browser's font size can be represented by 1em. Since the font size happens to be 16px then 1em is 16 px.

The absolute sizes are enemies of the responsive design Web sites, and specifying the sizes in em unit allows you to create Web pages with the pretty flexible and fluid content. The size can be calculated based on a formula offered by Ethan Markotte in his [article on fluid grids](#): `target/context=result`, which in case of fonts becomes `size-in-pixels/16 = size-in-em`.

For example, instead of specifying the size as 24px, you can set it to 1.5em:  $24/16 = 1.5\text{em}$ . In your CSS file you can write something like `padding-bottom: 1.5em`. This may seem not a big deal, but it is, because if everything is done in relative sizing, your page will look good and proportional regardless of the screen size and regardless of how big or small 24px may look on a particular screen.

If we are talking about em units for representing font sizes, the font becomes *the context*, but what if you want to represent the width of an arbitrary HTML component in a browser's window or any other container? Then the width of your component becomes the `target` and the total width of the container becomes the `context`. We can still use the above formula, but will multiply the result by 100%. This way the width on an HTML component will be represented not in em, but in percentages relative to the total width of the container.

Let's say the total width of the browser's window is 768px, and we want to create a 120px-wide panel on the left, instead of specifying this width in pixels we'll use the formula and turn it into percentages. We want to calculate the target's width in percents of the available context (100%):

$$120 / 768 * 100\% = 15.625\%$$

Such approach makes the page design *fluid*. If someone decides to open this page on a 480px-wide screen, the panel will still take 15.625% of the screen rather than demanding 120 pixels, which would look substantially wider on a smaller viewport.

## Window as a Grid

While designing your page you can overlay any HTML container or the entire Web page real estate with imaginary grid with any number of columns. Make it flexible though - the width of each column has to be specified in percentages.

Adobe Dreamweaver CS6 automates creation of media queries and it introduced Fluid Grid layout (see [Figure 10-13](#)). It also allows you to quickly see how your design will look like on the tablet or phone (you can pick screen size too) with a click on the corresponding status bar button.

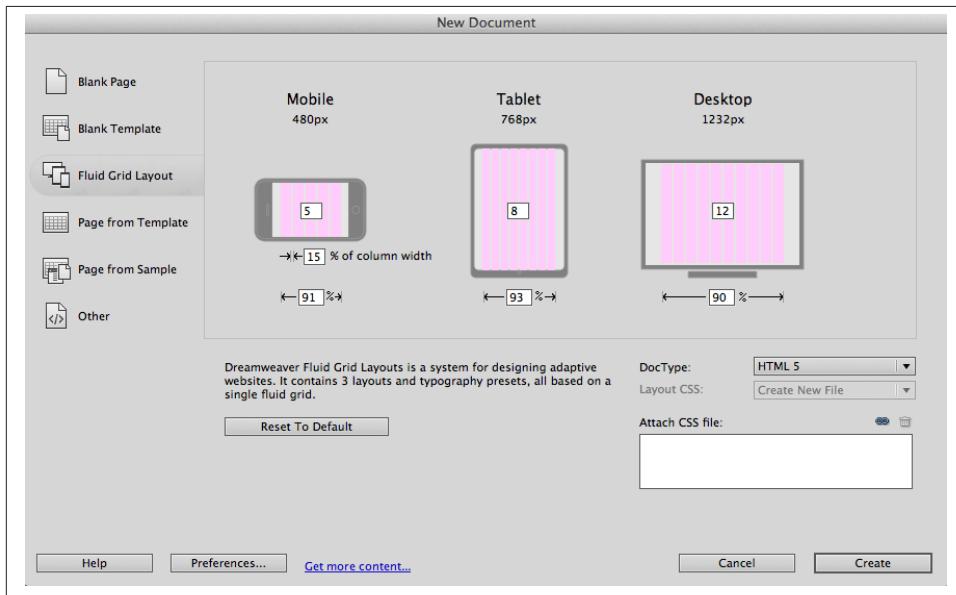


Figure 10-13. Creating a Fluid Grid Layout in Dreamweaver



Adobe's Creative Cloud includes a tool called [Edge Reflow](#), which will help designers in creation of responsive Web pages.

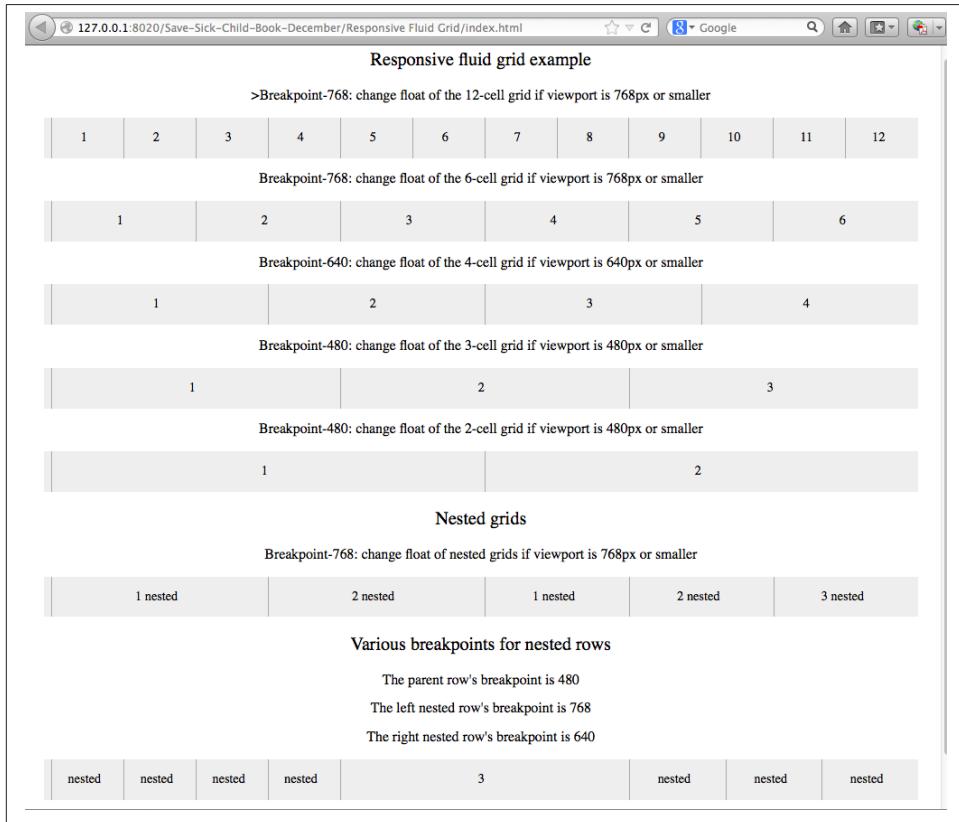
Web designers use different approaches in styling with fluid grids. When you design a new page with Dreamweaver's Fluid Grid Layout it suggests you to allocate different number of columns for desktop, tablet and mobile, for example, its default offer is to allocate 12 columns for the desktops, 8 for the tablets, and 5 for phones, which is perfectly solid approach. But our Web designer Jerry prefers using 12 columns for all screen sizes playing with the width percentages for different layouts - you'll see how he does it in the project Responsive Donation Section later in this chapter.

Now imagine that you'll overlay the entire window with an invisible grid containing twelve equally sized columns. In this case each column will occupy 8.333% of the total width. Now, if you'd need to allocate to some HTML component about 40% of the total

width, you could do this by allocating five grid columns ( $8.333\% \times 5 = 41.665\%$ ). Accordingly, your CSS file can contain 12 classes that you can use in your page:

```
.one-column {  
    width: 8.333%;  
}  
  
.two-column {  
    width: 16.666%;  
}  
  
.three-column {  
    width: 24.999%;  
}  
  
.four-column {  
    width: 33.332%;  
}  
  
.five-column {  
    width: 41.665%;  
}  
  
.six-column {  
    width: 49.998%;  
}  
  
.seven-column {  
    width: 58.331%;  
}  
  
.eight-column {  
    width: 66.664%;  
}  
  
.nine-column {  
    width: 74.997%;  
}  
  
.ten-column {  
    width: 83.33%;  
}  
  
.eleven-column {  
    width: 91.663%;  
}  
  
.twelve-column {  
    width: 100%;  
    float: left;  
}
```

Now let's see the fluid grid in action. Run the project Responsive Fluid Grid and you'll see the Web page that looks similar to [Figure 10-14](#). This example changes the grid layout if the viewport width falls under one of the following width breakpoints: 768px, 640px, and 480px. In this context the term *breakpoints* here has nothing to do with debugging - we just want the content of the Web page to be rearranged when the width of the viewport passes one of these values.



*Figure 10-14. Fluid Grid on the wide screen*

If you'll start lowering the width of the browser's window, you'll see how the grid cells start squeezing, but the layout remains the same until the page size will become lower than one of the predefined breakpoints. Then another media query kicks in and the layout changes. For example, [Figure 10-15](#) shows the fragment of the Web page when the width of the browser's window goes below 640px. The 12-, 6-, and 4-cell grids show all the cells vertically one under another. Only the 480px grids still have enough room to display their cells horizontally. But if you keep squeezing the window, all the grids will display their content in one column as long as the viewport width stays under 480px.





127.0.0.1:8020/Save-Sick-Child-B



Breakpoint-768: change float of the 6-cell grid if viewport is 768px or smaller



Breakpoint-640: change float of the 4-cell grid if viewport is 640px or smaller



Breakpoint-480: change float of the 3-cell grid if viewport is 480px or smaller



Breakpoint-480: change float of the 2-cell grid if viewport is 480px or smaller



### Nested grids

Breakpoint-768: change float of nested grids if viewport is 768px or smaller

The fragment of the index.html from the Responsive Fluid Grid project goes next. For brevity, we've removed some repetitive markup and marked such places with the comment "A fragment removed for brevity". This code fragment includes the 12-, 6-, and 4-column grids shown on top of Figure 10-14.

```
<head>
  <meta charset="utf-8">
  <title>Responsive fluid grid</title>
  <meta name="description" content="Responsive fluid grid example">
  <meta name="viewport" content="width=device-width,initial-scale=1">

  <link rel="stylesheet" href="css/style.css">
</head>

<body>
  <div id="wrapper-container">

    <h1 class="temp-heading">Responsive fluid grid example</h1>
    <h4 class="temp-heading">Breakpoint-768: change float of HTML elements
      if viewport is 768px or smaller</h4>
    <div class="row breakpoint-768">
      <div class="one-column cell">
        1
      </div>
      <div class="one-column cell">
        2
      </div>
      <div class="one-column cell">
        3
      </div>

      <!-- A fragment removed for brevity -->

      <div class="one-column cell last-cell" >
        12
      </div>
    </div>

    <h4 class="temp-heading">Breakpoint-768: change float of the 12-cell grid
      if viewport is 768px or smaller</h4>

    <div class="row breakpoint-768">
      <div class="two-column cell">
        1
      </div>
      <div class="two-column cell">
        2
      </div>

      <!-- A fragment removed for brevity -->

      <div class="two-column cell">
```

```

        </div>
    </div>

<h4 class="temp-heading">Breakpoint-768: change float of the 6-cell grid
    if viewport is 768px or smaller</h4>

<div class="row breakpoint-640">
    <div class="three-column cell">
        1
    </div>
    <div class="three-column cell">
        2
    </div>
    <div class="three-column cell">
        3
    </div>
    <div class="three-column cell">
        4
    </div>
</div>

```

Note that some of the above HTML elements are styled with more than one class selector, for example `class="one-column cell"`. The entire content of the file styles.css from Responsive Fluid Grids project is shown next, and you can find the declarations of the class selectors `one-column` and `cell` there.

```

* {
    margin: 0;
    padding: 0;
    border: 0;
    font-size: 100%;
    font: inherit;
    vertical-align: baseline;
    -webkit-box-sizing:border-box;
    -moz-box-sizing: border-box;
    box-sizing: border-box;
}

article, aside, details, figcaption, figure, footer, header, hgroup, menu, nav, section {
    display: block;
}

ul li {
    list-style: none;
}

.row:before, .row:after, .clearfix:before, .clearfix:after {
    content: "";
    display: table;
}

```

```
.row:after, .clearfix:after {  
    clear: both;  
}  
  
/* Start of fluid grid styles */  
  
.row {  
    padding: 0 0 0 0.5em; // ❶  
    background: #eee;  
}  
  
.breakpoint-480 .cell, .breakpoint-640 .cell, .breakpoint-768 .cell,  
    .breakpoint-960 .cell, .no-breakpoint .cell { //❷  
    float: left;  
    padding: 0 0.5em 0 0;  
}  
  
.one-column {  
    width: 8.333%; // ❸  
}  
  
.two-column {  
    width: 16.666%; // ❹  
}  
  
.three-column {  
    width: 24.999%; // ❺  
}  
  
.four-column {  
    width: 33.332%;  
}  
  
.five-column {  
    width: 41.665%;  
}  
  
.six-column {  
    width: 49.998%;  
}  
  
.seven-column {  
    width: 58.331%;  
}  
  
.eight-column {  
    width: 66.664%;  
}  
  
.nine-column {  
    width: 74.997%;  
}
```

```

.ten-column {
    width: 83.33%;
}

.eleven-column {
    width: 91.663%;
}

.twelve-column {
    width: 100%;
    float: left;
}

.right {
    float: right;
}

.row.nested {
    padding: 0;
    margin-right: -0.5em
}

```

- ❶ Styling grid rows, which are containers for cells.
- ❷ Defining common class selectors (floating and padding) for the cells located in the viewports of any width. Please note the property `float: left;` - it'll change in the media queries section.
- ❸ Dividing 100% of the container's width by 12 columns results in allocating 8.333% of width per column. Each cell in the 12-column table in our HTML has the `one-column` class selector.
- ❹ Check the HTML for the 6-column grid - each cell is styled as `two-column` and will occupy 16.666% of the container's width.
- ❺ The HTML for the 4-column grid uses the `three-column` style for each cell that will use 24.999% of the container's width.

Note the section with media queries in this file (below is just another fragment of the same CSS file).

```

/* ----- Media queries ----- */

@media only screen and (max-width: 768px) {
    .breakpoint-768 .cell {
        float: none; // ❶
        width: 100%;
        padding-bottom: 0.5em
    }
}

```

```

@media only screen and (max-width: 640px) {
    .breakpoint-640 .cell { // ②
        float: none;
        width: 100%;
        padding-bottom: 0.5em
    }
}

@media only screen and (max-width: 480px) {
    .breakpoint-480 .cell {
        float: none;
        width: 100%;
        padding-bottom: 0.5em
    }
}

/*End of fluid grid styles*/

#wrapper-container {
    width: 95%;
    max-width: 1140px;
    margin: 0 auto;
}

/* --- .cell visualisation --- */
.cell {
    min-height: 50px;
    text-align:center;
    border-left: 1px solid #aaa;
    vertical-align: middle;
    line-height: 50px;
}
.cell .cell:first-child{
    border-left:none;
}
/* --- .cell visualisation end --- */

h1.temp-heading, h2.temp-heading, h4.temp-heading {
    font-size: 1.4em;
    margin: 1em 0;
    text-align: center
}
h4.temp-heading {
    font-size: 1.1em;
}

p.temp-project-description {
    margin: 2em 0;
}

```

- ❶ This media query turns off the floating (`float:none`) if the viewport is 768px or less. This will reallocate the cells vertically. The `width:100%` forces the cell to occupy the entire width of the container as opposed to, say 8.333% in the 12-column grid.
- ❷ The media query for 640px won't kick in until the viewport width goes below 640px. If you'll resize the browser window to make it below 768px but larger than 640px, note that the 4-column grid (styled as `breakpoint-640`) has not changed its layout just yet.



In some cases you may need to use a mix of fluid and fixed layouts, for example, you may need to include an image of a fixed size on your fluid Web page. In such cases you can use a fixed width on some of the elements, and if needed, consider using CSS tables (not to be confused with HTML tables). CSS tables **are supported** by all current browsers.

Spend some time analyzing the content of `index.html` and `styles.css` files from the project named Responsive Fluid Grid. Try to modify the values in CSS and see how your changes affect the behavior of the fluid grid. In the next section we'll apply these techniques to our Save The Child application.

## Responsive CSS: The Good News

We were explaining how the fluid grid work under the hood, but calculating percentages is not the most exciting job for software developers. The good news is that there are several responsive frameworks that offer CSS, typography and some JavaScript to jump start the development of the UI of a Web application. They'll spare you for most of the mundane work with cascading style sheets. Here are some of them:

- Consider using Twitter's framework called **Bootstrap**, which has lots of greatly styled **components** and also supports fluid grid system.
- The **Foundation 4** framework promotes mobile first design and includes the flexible grid.
- **The Skeleton** is a collection of CSS files, which includes a scalable grid.
- **Semantic-UI** is a collection of styled UI components, which includes **responsive grid** too.



People who work with CSS a lot use an authoring framework **Compass** with CSS extension **SASS** or a CSS pre-processor **LESS**, which are systems that compile to CSS allowing for code including variables for tracking and calculating such numbers as column width and more. You can now modularize your CSS as well as your code. In Chapter 12 we'll use a SASS theme that comes with Sencha Touch framework.

## Making Save The Child Responsive

First, run any of the previous versions of the Save The Child application to make sure it was not responsive. Just make the browser window narrower, and you won't see some of the page content on the right. We'll make the page responsive gradually - the first version will make the header responsive, then the donation section, and, finally the entire page will become fluid. Open in the Web browser the file index.html from the project named Responsive Header and you'll see a page similar to [Figure 10-16](#).

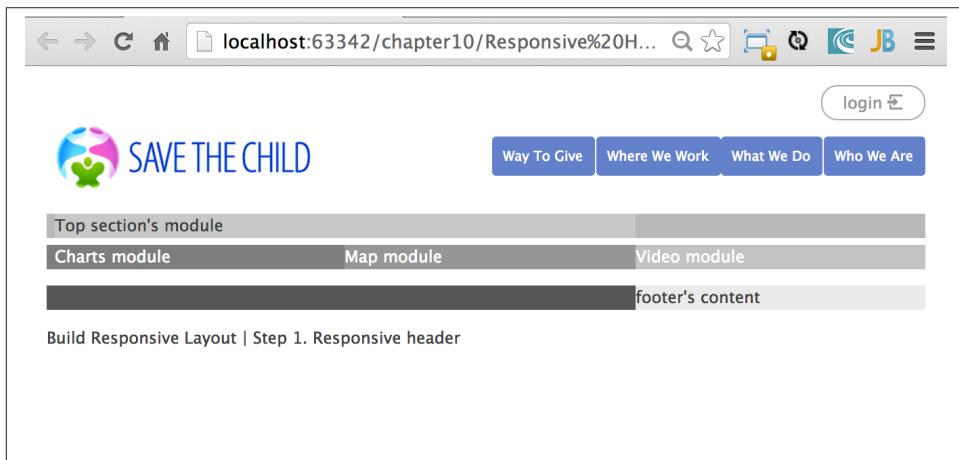


Figure 10-16. Responsive Header (width 580px+)

Below is the fragment from index.html that displays the logo image and the header's menus.

```
<div id="wrapper-container">
  <header class="row breakpoint-640">
    <h1 id="logo" class="four-column cell">
      </h1>
    <nav class="eight-column cell">
      <ul>
        <li>
          <a href="javascript:void(0)">Who We Are</a>
```

```

        </li>
        <li>
            <a href="javascript:void(0)">What We Do</a>
        </li>
        <li>
            <a href="javascript:void(0)">Where We Work</a>
        </li>
        <li>
            <a href="javascript:void(0)">Way To Give</a>
        </li>
    </ul>
</nav>
```

Initially, this code uses the `four-column` style (`width: 33.332%` of the container) for the logo and `eight-column` (`66.664%`) for the `<nav>` element. When the size of the viewport changes, the appropriate media query takes effect. Note the `breakpoint-640` class selector in the `<header>` tag above. Jerry, our Web designer, decided that 640 pixels is not enough to display the logo and the four links from the `<nav>` section in one row. Besides, he wanted to fine tune the width of other elements too. This is how the media query for the 640px viewport looks like this:

```

@media only screen and (max-width: 640px) {
    .breakpoint-640 .cell {
        float: none;
        width: 100%;
        padding-bottom: 0.5em
    }

    header {
        margin-top: 1em;
    }
    #login {
        top: 1em;
    }
    #logo.four-column {
        width: 40%;
    }
    nav {
        width: 100%;
        margin-top: 0.8em
    }
    nav ul li {
        width: 24.5%;
        margin-left: 0.5%
    }
    nav li a {
        text-align: center;
        font-size: 0.6em;
    }
    #login-link-text {
        display: none;
    }
}
```

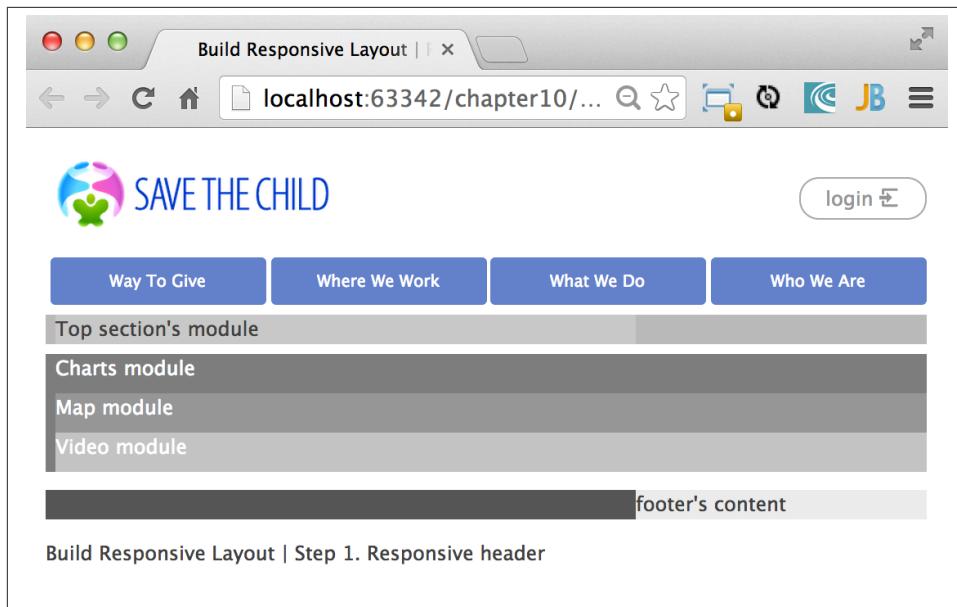
```

        }
        a#login-submit {
            padding: 0.2em 0.5em
        }
        #login input {
            width: 9em;
        }
    }
}

```

As you see, if the `cell` has to be styled inside `breakdown-640`, the float gets turned off (`float: none;`) and each of the navigation items has to take 100% of the container's width. The `logo`, `login`, and `nav` elements will change too. There is no exact science here - Jerry figured out all these values empirically.

Start slowly changing the width of the viewport, and you'll see how the layout changes. The styles.css of this project has media queries for different viewport sizes. For example, when the page width is below 580 pixels, but more than 480 pixels it'll look as in [Figure 10-17](#).



*Figure 10-17. Responsive Header 2 (width between 480 and 580px )*

When the width of the viewport shrinks below 480px, the header's content rearranges and looks as in [Figure 10-18](#). Once again, we are not tying the design to the specific device, but rather to a viewport width. The iPhone 4 will render this page using the layout shown at <>>FIG11-16>, but iPhone 5 will use the layout from [Figure 10-17](#). You can't go by a device type.

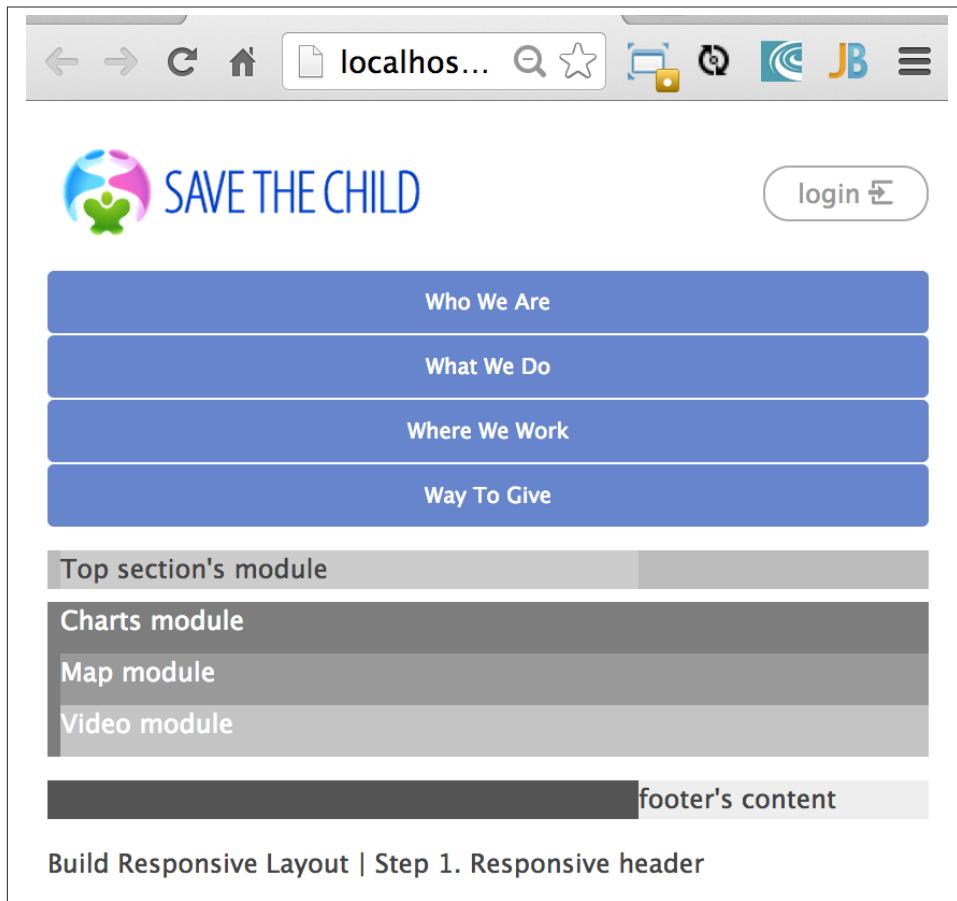


Figure 10-18. Responsive Header (viewport's width below 480px)

The next project to try is called Responsive Donation. This version make the donation section fluid. The donation section contains the Lorem Ipsum text and the form, which is revealed when the user clicks the button Donate. First, let's look at the HTML. The index.html contains the following fragment (some of the content that irrelevant for layout was removed for better readability):

```
<div id="main-content" role="main">
<section id="main-top-section" class="row breakpoint-480">
  <div id="donation-address" class="seven-column cell">
    <p class="donation-address">
      Lorem ipsum dolor sit amet
    </p>
    <button class="donate-button" id="donate-button">
      <span class="donate-button-header">Donate Now</span>
    </button>
```

```

</div>
<div id="donate-form-container">
    <h3>Make a donation today</h3>
    <form name="_xclick" action="https://www.paypal.com/cgi-bin/webscr"
        method="post">

        <div class="row nested breakpoint-960">
            <div class="six-column cell">
                <div class="row nested">
                    <div id="donation-amount" class="five-column left">
                        <label class="donation-heading">Donation amount</label>
                        <input type="radio" name="amount" id="d10" value="10"/>
                        <label for="d10">10</label>
                    </div>
                    <div id="donor-info" class="five-column left">

```

The donation section is located in the `main-top-section` of the page. Jerry wanted to keep the image of the boy visible for as long as possible on the narrower viewports. The top section of the Save The Child has two backgrounds: the flowers (`bg-2.png`) and the boy (`child-1.png`). This is how they are specified in the `style.css`:

```

#main-top-section {
    background: url(..../img/child-1.png) no-repeat right bottom,
                url(..../img/bg-2.png) no-repeat 20% bottom;
}

```

If the viewport is wide enough, both backgrounds will be shown. What's wide enough? Jerry figured it out after experimenting. The `seven-column` style prescribes to allocate more than a half (58.331%) of the viewport width for the `donation-address` section and `six-column` (49.998%) for the donation form. For example [Figure 10-19](#) shows how the donation section will look when the viewport width is 570px.

The screenshot shows a responsive web page for "SAVE THE CHILD". The header features a logo of three interlocking puzzle pieces in blue, green, and pink, followed by the text "SAVE THE CHILD". A "login" button is located in the top right corner. Below the header, there are four navigation tabs: "Way To Give", "Where We Work", "What We Do", and "Who We Are". The "Way To Give" tab is currently active.

The main content area is titled "Make a donation today". It includes a "Donation amount" section with radio buttons for 10, 20, 50, 100, or 200, and an "Other amount" input field. To the right of these fields is a "Donor information" section containing fields for full name, email, address, city, zip/postal code, state, and country. The "Donor information" section is decorated with small cartoon icons of a butterfly, a sun, and clouds.

A heading "We accept Paypal payments" is followed by a paragraph explaining secure processing. To the right of this text is a large, smiling photo of a young child.

A prominent yellow button labeled "DONATE NOW Children can't wait" is positioned below the donation form. Below this button is a link "I'll donate later" with a close button. The background features a decorative border of flowers and butterflies.

At the bottom of the page, there are three module links: "Charts module", "Map module", and "Video module". A footer bar at the very bottom contains the text "Build Responsive Layout | Step 2. Donation Section" and "footer's content".

Figure 10-19. Responsive Donate Section: 570px

But when the width become less then 480px, there is no room for two background images, and only the flowers will remain on the page background. The media query for 480px viewport is shown next - note that the background in the main top section has only one image now: bg2.png. Floating is off to show the navigation menu vertically as in [Figure 10-20](#).

```
@media only screen and (max-width: 480px) {  
    .breakpoint-480 .cell {  
        float: none;  
        width: 100%;  
        padding-bottom: 0.5em  
    }  
    #logo {  
        padding-bottom: 11em  
    }  
    nav ul li {  
        float: none;  
        width: 100%;  
        margin-left: 0;  
        margin-bottom: 0.5%;  
    }  
    #main-top-section {  
        background: url(..../img/bg-2.png) no-repeat 20% bottom;  
    }  
    .donate-button {  
        width: 14em;  
        margin-left: auto;  
        margin-right: auto;  
    }  
    .donate-button-header {  
        font-size: 1.1em;  
    }  
    .donate-2nd-line {  
        font-size: 0.9em;  
    }  
    #donate-later-link {  
        display: block;  
        width: 11em;  
        margin-left: auto;  
        margin-right: auto;  
    }  
    #make-payment p {  
        width: 100%;  
    }  
    #donation-amount.five-column {  
        width: 50%  
    }  
    #donor-info.six-column {  
        width: 50%  
    }  
    #donate-form-container select, input[type=text], input[type=email] {  
        width: 90%;  
    }
```

}  
}

The screenshot shows a mobile browser displaying the 'SAVE THE CHILD' website. The top navigation bar includes standard icons for back, forward, refresh, home, search, and other browser functions. The main header features the 'SAVE THE CHILD' logo with a stylized green and blue globe icon. To the right of the logo is a 'login' button with a small orange bird icon above it. Below the header is a vertical menu with four blue rectangular buttons: 'Who We Are', 'What We Do', 'Where We Work', and 'Way To Give'. The main content area is titled 'Make a donation today' and features a butterfly icon. It includes fields for 'Donation amount' (radio buttons for 10, 20, 50, 100, 200, and an 'Other amount' input field) and 'Donor information' (input fields for full name, email, address, city, zip/postal code, State, and Country). Below this is a section titled 'We accept Paypal payments' with a message about secure processing. A large yellow 'DONATE NOW' button with the subtext 'Children can't wait' is prominently displayed. At the bottom, there are links for 'Charts module', 'Map module', and 'Video module', followed by a dark footer bar labeled 'footer's content'. A link at the very bottom reads 'Build Responsive Layout | Step 2. Donation Section'.

Figure 1 Chapter 10 Responsive Design One Site fits All under 480px

The project Responsive Final includes the charts, maps, and video. Each of these sections uses four-column style, which is defined in styles.css as 33.332% of the container's width.

```
<section id="main-bottom-section" class="row breakpoint-768">

  <div id="charts-container" class="four-column cell">
    <svg id="svg-container" xmlns="http://www.w3.org/2000/svg">
      </svg>
      <h3>Donation Stats</h3>
      <h5>Lorem ipsum dolor sit amet, consectetur.</h5>
    </div>
    <div id="map-container" class="four-column cell">
      <div id="location-map"></div>
      <div id="location-ui"></div>
    </div>
    <div id="video-container" class="four-column cell last">
      <div id="video-wrapper">
        <video id="movie" controls="controls"
          poster="assets/media/intro.jpg" preload="metadata">
          <source src="assets/media/intro.mp4" type="video/mp4">
          <source src="assets/media/intro.webm" type="video/webm">
          <p>Sorry, your browser doesn't support the video element</p>
        </video>
      </div>
      <h3>Video header goes here</h3>
      <h5><a href="javascript:void(0);">More video link</a></h5>
    </div>
  </section>
```

The id of this section is still main-bottom-section, and it's shown at the bottom of the page on wide viewports. Now take another look at the image [Figure 10-11](#). Jerry wants to display these three sections at the right hand side for tablets in the portrait mode, and it's shown on [Figure 10-21](#).

The screenshot shows a responsive web design for a charity organization. At the top, there's a header bar with a back arrow, forward arrow, refresh button, and a home icon. The URL 'savesickchild.org:8080/ssc-responsive/' is displayed. To the right of the URL are several icons: a star, a 'JB' logo, a square with a diagonal line, a mail icon, a circular icon, and a three-line menu icon.

The main content area features a large, smiling child's face in the center-left. Above the child is the organization's logo, which consists of three stylized human figures in green, blue, and pink. To the right of the child are three small decorative icons: a white cloud-like shape, a yellow butterfly, and a purple flower.

Below the child's face is a block of placeholder text (Lorem ipsum) in a sans-serif font. To the right of this text is a pie chart titled 'Donation Stats' showing the distribution of donations by location:

Location	Count
Chicago, IL	48
New York, NY	60
Dallas, TX	90
Miami, FL	22
Fargo, ND	14
Long Beach, NY	44
Lynbrook, NY	24

Below the pie chart is a section titled 'Donation Stats' with a short description: 'Lorem ipsum dolor sit amet, consectetur.' To the right of this is a map of the United States and Mexico, showing donation locations with red pins. The map includes state and country labels like California, Texas, Mexico, and Guatemala. A 'Google' logo is visible at the bottom left of the map.

On the far left, below the child's face, is a yellow call-to-action button with the text 'DONATE NOW' and 'Children can't wait' underneath it. Below this button is a decorative horizontal bar featuring colorful flower icons.

To the right of the map is a video player window. The video frame shows a chalkboard with the equation 'E=mc<sup>2</sup>' written on it. A young child is seen from behind, reaching up towards the chalkboard. The video player includes controls for play, volume, and progress (0:10).

Below the video player, there's a section with the text 'Video header goes here' and a link 'More video link'.

Figure 10-21. The Portrait Mode on Tablets

The relevant code from the style.css is shown below. The top and bottom sections get about a half of the width each, and the floating is turned off so the browser would allocate charts, maps, and video vertically.

```
@media only screen and (max-width: 768px) {  
    .breakpoint-768 .cell {  
        float: none;  
        width: 100%;  
        padding-bottom: 0.5em;  
    }  
  
    #main-bottom-section, #main-top-section {  
        width: 49%;  
    }  
}
```



We've explained the use of media queries for applying different styles to the UI based on screen resolutions. But there is a twist to it. What device comes to mind if you hear about the screen with the resolution of 1920x1080 pixels? Most likely you got it wrong unless your answer was the smartphone Galaxy S4 or Sony Xperia Z. The resolution is high, but the screen size is 5 inches. What media query are you going to apply if the user has such a device? Even with such high resolution you'd rather not apply the desktop's CSS to such a mobile device. The CSS media query *device-pixel-ratio* may help you in telling apart high-resolution small devices from desktops.

## Fluid Media

If your responsive Web page contains images or videos, you want to make them fluid too - they should react to the current size of the containers they are in. Our page has a chart image and a video - both of them are made flexible, but we use different techniques.

If you'll keep narrowing the viewport, the project Responsive Final will show the page with the layout similar to [Figure 10-12](#). While reading the code of this project, visit the main.js file. There is some work done in the JavaScript too, which listens to the resize event for the charts container.

```
window.addEventListener("resize", windowResizeHandler);  
function windowResizeHandler() {  
    drawPieChart(document.getElementById('svg-container'),  
                donorsDataCache, labelsDataCache);  
}
```

Whenever the size changes, it invokes the function drawPieChart() that recalculates the width of the SVG container (it uses the clientWidth property of the HTMLElement) and re-draws the chart accordingly.



Consider storing images in the [WebP format](#), which is a lossless format, and WebP images are about 25% less in size than PNG or JPEG images. Your application needs to check first if the user's Web browser supports WebP format, otherwise images in more traditional formats should be rendered. The other choice is to use [Tumbor imaging service](#) that can automatically serve WebP images to the browsers that support this format.

The video is flexible too, and it's done a lot simpler. We do not specify the fixed size of the video, but use a CSS property `width` instructing the browser to allocate the 100% of the available container's width. The height of the video must be automatically calculated to keep the proportional size.

```
video {  
    width: 100% !important;  
    height: auto !important;  
}
```

The `!important` part disables regular cascading rules and ensures that these values will be applied overriding more specific width or height declarations, if any. If you prefer not always use the entire width of the container for the video, you can use the `max-width: 100%;`, which will display the video that fits in the container at its original size. If a video is larger than the container, the browser will resize it to fit inside the container.

While the landing Web page of your application simply includes links to the required images, the rest of the images should be loaded from the server by making AJAX requests with passing parameter regarding the viewport size. This way the server's software can either resize images dynamically and include them as base-64 encoded strings or use pre-created properly sized images depending on the viewport dimensions.



While using base-64 encoding increases the total size of the image in bytes, it allows you to group together multiple images to minimize the number of network calls the browser needs to make to retrieve these images separately. The other way to combine multiple images into one is CSS sprites.

Regardless of what the width and height of the image is, use tools to reduce image sizes in bytes. Some of such tools are [TinyPNG](#) or [Smush.it](#). If you use *lossy* tools, some of the image data will be lost during compression, but in many cases the difference between the original and compressed image is invisible.



**Sencha.io SRC** is a proxy server that allows you to dynamically resize images for various mobile screen sizes.

Besides making images responsive, keep in mind that some people have mobile devices with high resolution retina displays. The problem is that to make an image look good on such displays its size has to be large, which increases its loading time. There is no common recipe for doing the image size optimization properly - plan to spend an extra time just to preparing the images for your application.

There is a living W3C document titled [An HTML extension for adaptive images](#) that will provide developers with a means to declare multiple sources for an image. The proposed HTML element `<picture>` will allow to specify different images for different media (see [demos](#)), for example:

```
<picture width="500" height="500">
  <source media="(min-width: 45em)" src="large.jpg">
  <source media="(min-width: 18em)" src="med.jpg">
  <source src="small.jpg">
  
</picture>
```

Another technique is to have a CDN that caches and serves images of different sizes for different user agents. The very first time when a request is made from a device with an unknown user engine, this first “unlucky” user will get an image with a low resolution, and then the application makes an AJAX call passing the exact screen parameters for this device. The CDN server resizes the original high-resolution image for this particular user agent, and caches it, so any other users having the same device will get a perfectly-sized image from the get go.



[Imager.js](#) is an alternative solution to the issue of how to handle responsive image loading, created by developers at BBC News. Imager loages the most suitable sized image and does it once.

## Summary

Responsive Web design is not a silver bullet that allows using a single code base for all desktop and mobile versions of your HTML5 Web application. RWD can be the right approach for developing Web sites that mainly publish information. It's not likely that you can create a complex single-code-base Web application that works well on Android, iPhone, and desktop browsers.

Responsive design may result in unnecessary CSS loaded to the user's device. This consideration is especially important for mobile devices operating on 3G or slower networks (unless you'll find a way to lazy load them).

Responsive design can still be a practical business solution when the form factor is relatively low (which enterprise can mandate), e.g. if your target group of users operate specific models of iOS and Android devices.

If you'll take any JavaScript framework that works on both desktop and mobile devices, you'll get two sets of controls and will have to maintain two different source code repositories. Not using mobile JavaScript frameworks limits the number of user-friendly UI controls. Besides, frameworks spare you from dealing with browsers' incompatibilities.

In this chapter you've seen how the Save The Child application was built with responsive design principles. We have several areas (`<div>`'s) and one of them included a donation form (we could have added the responsive `<div>` with the online auction too). On the wide screen we displayed three of these `<div>`'s horizontally and two underneath, on the narrow screen each of these sections could scale down and displayed one under another.

But using responsive design for styling the application that must run on tablets or mobile devices will require Jerry-the-designer to work in tandem with the User Experience specialist so that UI will have larger controls and fonts while minimizing the need of manual data entry. And don't forget that the half of a mobile screen could be covered by a virtual keyboard, and if you ignore this, the user will look at your application's UI via a keyhole and even our fluid `<div>`'s may not fit.

In the next two chapters we'll be working on yet another version of the Save The Child application. First, we'll use the jQuery Mobile framework and then - Sencha Touch.

---

# CHAPTER 11

# jQuery Mobile

According to [jquerymobile.com](http://jquerymobile.com), *jQuery Mobile is a HTML5-based user interface system designed to make responsive web sites and apps that are accessible on all smartphone, tablet and desktop devices.* But jQuery Mobile was mainly created for developing Web applications for smaller screens.

To start learning jQuery Mobile you need to know HTML, JavaScript, CSS, and jQuery. In some publications you may see the statements that you could start using jQuery Mobile knowing only HTML. This is true till you'll run into the first unexpected behavior of your code, which will happen pretty soon in one of the Web browsers (take the statements about being a cross-browser framework with a grain of salt too). After that you need to add some event listeners, scripts, and start debugging.

## Where to get jQuery Mobile

The Web site of jQuery Mobile has all you need to start using this library. you can find lots of learning materials under the Demos section - they have tutorials, API reference, and samples of use. The Download section contains the links for the library itself.

There are two ways of including jQuery Mobile in the source code of your application: either download and uncompress the zip file in your local directory and specify this location in the source code of your application or include the URLs of the CDN-hosted files. Visit the [jQuery Mobile Download](#) page for the up-to-date URLs.

In our code samples we'll be adding the following code snippets, which in gzipped format will make our application only 90Kb "heavier":

```
<link rel="stylesheet" href="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.css" />
<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script src="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.js"></script>
```

# Organizing the Code

The UI of a jQuery Mobile application consists of a set of HTML documents where certain attributes are added to the regular HTML components. Your Web application will consist of *pages*, and the user's mobile device will show one page at a time. After the mockup of your application is ready (see section "Prototyping with Balsamiq Mockups" below), you know how many pages your application will have and how to navigate between the pages. Let's see how to define the content of each page in jQuery Mobile.

HTML5 specification includes an important feature - you can add to any HTML tag any number of **custom non-visible attributes** as long as they start with `data-` and have at least one character after the hyphen. In jQuery Mobile this feature is being used in a very smart way. For example, you can add an attribute `data-role` to the HTML tag `<div>` to specify that it's a page with id `Stats`:

```
<div data-role="page" id="Stats">
```

The UI of your application will consist of multiple pages, but what's important, jQuery Mobile will show them *one page at a time*. Let's say your application consists of two pages (Stats and Donate), then HTML may be structured as follows:

```
<body>
  <!-- Page 1 -->
    <div data-role="page" id="Donate">
      ...
    </div>

  <!-- Page 2 -->
    <div data-role="page" id="Stats">
      ...
    </div>
</body>
```

When this application starts, the user will see only the content of the page `Donate` since it was included in the code first. We'll talk about defining navigation a bit later.



The above code fragment is an example of a *multi-page template*, where a single HTML document contains multiple pages. An alternative way of organizing the code is to have the content of each page in a separate file or a *single-page template*, and you'll see the example later in this chapter.

Let's say you want a page to be divided into the header, content and the footer. Then you can specify the corresponding roles to each of these sections.

```
<body>
  <!-- Page 1 -->
    <div data-role="page" id="Donate">
```

```

<div data-role="header" >...</div>
<div data-role="content" >...</div>
<div data-role="footer" >...</div>

</div>

<!-- Page 2 -->
<div data-role="page" id="Stats">
    ...
</div>
</body>

```

It's not a must to split the page with the data roles header, content, and footer. But if you do, the code will be better structured and additional styling can be applied in the CSS based on these attributes.



It would be a good idea to replace three `<div>` tags inside the Donate page with HTML5 tags `<header>`, `<article>`, and `<footer>` but during the learning stage this could have confuse you mixing up HTML5 `<header>` and jQuery Mobile data role header (the footer line might have looked confusion too).

Let's say you want to add navigation controls to the header of the page. You can add to the header a container with a `data-role="navbar"`. In the following code sample we'll use the menus from the Save The Child application.

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.css"/>
</head>
<body>

<div data-role="page">
    <div data-role="header">
        <h1>Donate</h1>
        <div data-role="navbar">
            <ul>
                <li>
                    <a href="#Who-We-Are">Who We Are</a>
                </li>
                <li>
                    <a href="#What-We-Do">What We Do</a>
                </li>
                <li>
                    <a href="#Where-We-Work">Where We Work</a>
                </li>
            </ul>
        </div>
    </div>
    <div data-role="content">
        ...
    </div>
    <div data-role="footer">
        ...
    </div>
</div>

```

```

<li>
    <a href="#Way-To-Give">Way To Give</a>
</li>
</ul>
</div>
</div> <!-- header -->

<div data-role="content" >
    The content goes here
</div>

<div data-role="footer" >
    The footer goes here
</div>

<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script src="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.js"></script>
</body>
</html>

```

We'll explain the meaning of the HTML anchor tags in the section "Adding Page Navigation below". Note the The `<viewport>` tag in the above example. It instructs the browser of the mobile device to render the content to a virtual window that has to be the same as the width of the device's screen. Otherwise the mobile browser may assume that it's a Web site for desktop browsers and will minimize the content of the Web site so the user would need to zoom out. Read more about it in the sidebar titled "The Viewport Concept" in Chapter 10.



You can find the list of all available jQuery Mobile `data-` attributes in the [Data attribute reference](#) from the online documentation.

The above code sample is a complete HTML document that you can test in your browser. If you'll do it in your desktop Web browser, the Web page will look as in [Figure 11-1](#).

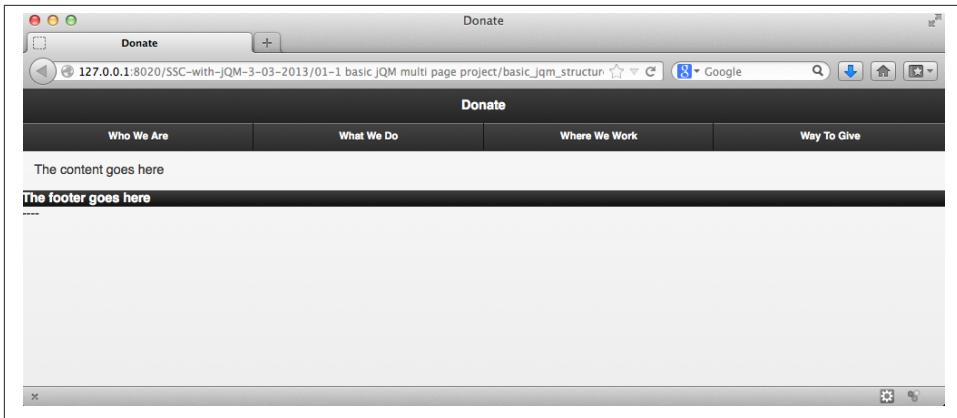


Figure 11-1. Viewing the document in Firefox

## How Will It Look on Mobile Devices?

Any mobile Web developer wants to see how his Web application will look on mobile devices. There two major ways of doing this: either test it on a real device or use a software emulator or simulator. Let's talk about the emulators - there are plenty of them available.

For example, you can use one of the handy tools like Ripple Emulator. This Chrome browser's extension will add a green icon on the right side of the browser's toolbar - click on it and enable Ripple to run in a Web Mobile default mode. Then select the mobile device from the dropdown on the left and copy/paste the URL of your HTML document into Chrome browser's address bar. [Figure 11-2](#) shows how our Web page would be rendered on the mobile phone Nokia97/5800.



There are emulators that target specific platform. For example, you can consider [Android Emulator](#) or use iOS simulator that comes with Apple's Xcode IDE. Chrome Developer Tools has an [emulator panel](#) too. For Nokia emulators browse their [developer's forum](#). Blackberry simulators are [here](#). Microsoft also offers [an emulator](#) for their phones. You can more detailed list of various emulators and simulators in the O'Reilly book "Programming the Mobile Web, 2nd Edition" by Maximiliano Firtman.

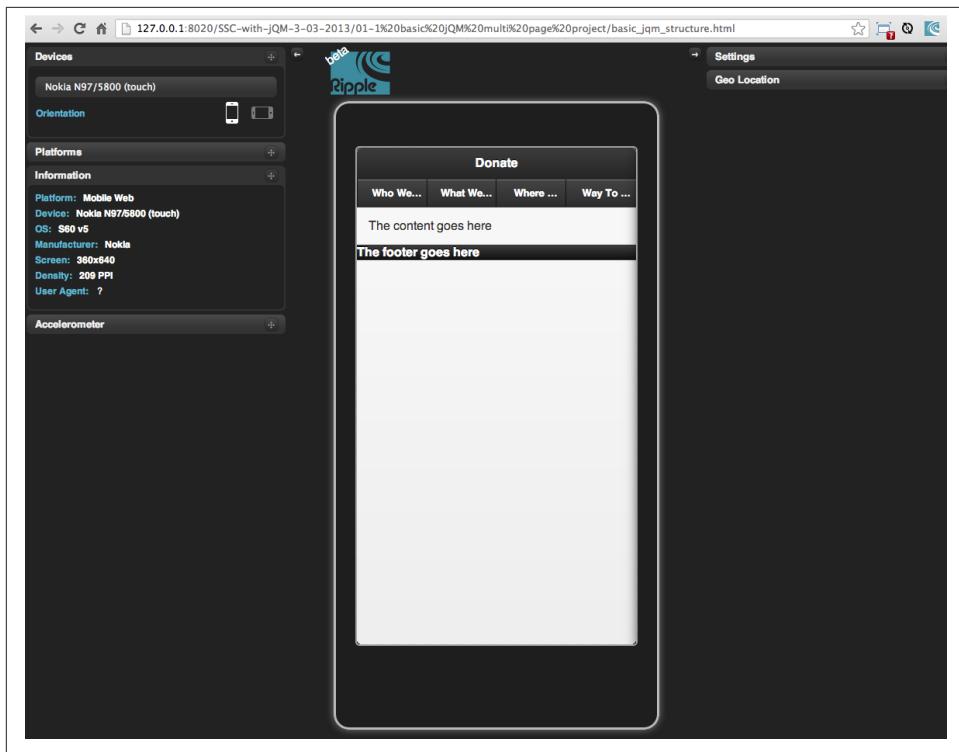


Figure 11-2. Viewing the document in Ripple Emulator

Using emulators really helps in the development. Ripple emulates not only the screen resolutions, but some of the hardware features as well (simulators usually simulate only the software). For example, you can test accelerometer by using the corresponding menu item under Devices (top left on [Figure 11-2](#)) or GEO Location under Settings (top right on [Figure 11-2](#)). But keep in mind that emulators run in in your desktop browser, which may render the UI not exactly the same way as a mobile browser running on the user's mobile phone. For example the fonts may look a little different. Hence testing your application on a real device is highly recommended even though it's impossible to test your Web application on thousands different devices people use.

If you can afford, hire real mobile users carrying different devices. You can do it at [Mob4Hire](#) testing as service (TaaS) Web site. The good news is that creators of jQuery Mobile use about [70 physical devices](#) for testing of their UI components, but still, you may want to see how your application looks and feels on a variety devices.

If you want to see how your application looks on a real device that you own, the easiest way is to deploy your application on a Web server with a static IP address or a dedicated domain name. After the code is modified, you need to transfer the code to that remote server and enter its URL in the address bar of your mobile device browser.

If you're developing for iOS on MAC OS X computer, the procedure is even easier if both devices are on the same Wi-Fi network. Connect your iOS device to your MAC computer via the USB input. In computer's System Preferences click on Networks and select your Wi-Fi connection on the left - you'll see the IP address of your computer on the right, e.g. 192.168.0.1. If your application is deployed under the local Web server, you can reach it from your iOS device by entering in its browser address bar the URL of your application using the IP address of your computer, e.g. <http://192.168.0.1/myApp/index.html>. For details, read [this blog](#).



If your mobile application behaves differently than on real device, see if there is an option for remote debugging on the device for your platform. For example, in [this document](#) Google explains how to do a remote debugging in Chrome browser running on Android devices. The Web browser Safari 7 supports remote debugging on iOS devices, [details here](#).

## Styling in jQuery Mobile

You may not like the design of the navigation bar shown on [Figure 11-1](#), but it has some style applied to it. Where the white letters on the black background are coming from? It happens because we've included the `data-role="navbar"` in the code. This is the power of the the custom `data-` attributes in action. Creators of the jQuery mobile included into their CSS predefined styling for different `data-` attributes including the inner buttons of the `navbar`.

What if you don't like this default styling? Create your own CSS, but first see if you might like some of the off-the-shelf themes offered by jQuery Mobile. You can have up to 26 pre-styled sets of toolbars, content and button colors called *swatches*. In the code you'll referr them as themes lettered from A to Z. Adding the `data-theme="a"` to the `<div data-role="page">` will make change the look of the entire page. But you can use the `data-theme` attribute with any HTML element, not necessarily for the entire page or other container.

By default, the header and the footer use swatch "a", and the content area - swatch "c". To change the entire color scheme of [Figure 11-2](#) to swatch "a" (the background of the content area will become dark gray) use the following line:

```
<div data-role="page" data-theme="a">
```

jQuery mobile has a tool [ThemeRoller](#) that allows you to create a unique combination of colors, fonts, backgrounds and shadows and assign it to one of the letters of the English alphabet (see [Figure 11-3](#)).

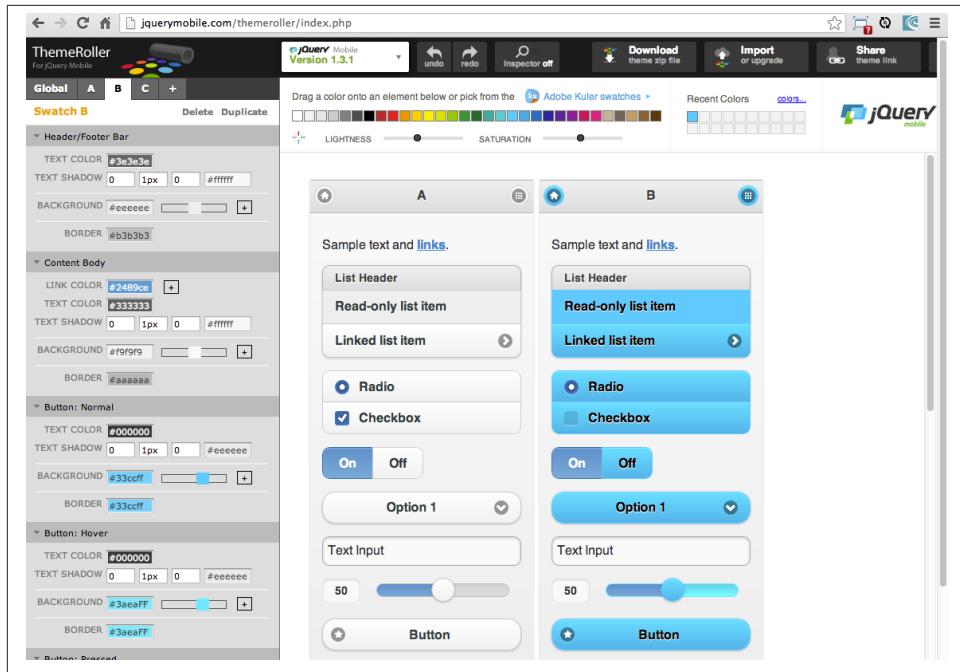


Figure 11-3. Theme Roller

You can learn about creating custom themes with ThemeRoller by visiting [this URL](#).

## Adding Page Navigation

In jQuery Mobile page navigation is defined by using the HTML anchor tag `<a href="">`, where the `href` attribute can either point at a page defined as a section in the same HTML document or at a page defined in a separate HTML document. Accordingly, you can say that that we're using either a *multi-page template* or a *single-page template*.

### Multi-Page Template

With multi-page template each page is a `<div>` (or other HTML container) with an id, and the `href` attribute responsible for navigation will include the hash tag followed by the corresponding id.

```

<body>
  <!-- Page 1 -->
  <div data-role="page" id="Donate" data-theme="e">
    <h1>Donate</h1>

    <a href="#Stats">Show Stats</a>
  </div>

```

```

<!-- Page 2 -->
<div data-role="page" id="Stats">
    <h1>Statistics</h1>
</div>
</body>

```

If you use multi-page document, the ID of the page with a hash (#) will be added to the URL. For example, if the name of the above document is navigation1.html, when the Stats page is open the browser's URL may look like this:

`http://127.0.0.1/navigation1.html#Stats`

Let's say that the only way to navigate from the Stats page is to go back to the page Donate. Now we'll turn the above code fragment into a working 2-page document with the Back button support. Both pages in the following HTML document have a designated areas with the `data-role="header"`, and the Stats page has yet another custom property `data-add-back-btn="true"`. This is all it takes to ensure that the Back button is displayed in the left side of the page header, and when the user will *tap* on it, the application will navigate to the Donate page.

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.css"/>
    </head>
    <body>
        <!-- Page 1 -->
        <div data-role="page" id="Donate">
            <div data-role="header" >
                <h1>Donate</h1>
            </div>
            <a href="#Stats">Show Stats</a>
        </div>

        <!-- Page 2 -->
        <div data-role="page" id="Stats" data-add-back-btn="true">
            <div data-role="header" >
                <h1>Statistics</h1>
            </div>
            Statistics will go here
        </div>

        <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
        <script src="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.js"></script>
    </body>
</html>

```

**Figure 11-4** shows a snapshot of the Ripple emulator after the user clicked on the link on the Donate page. The Statistics page now includes the fully functional Back button.

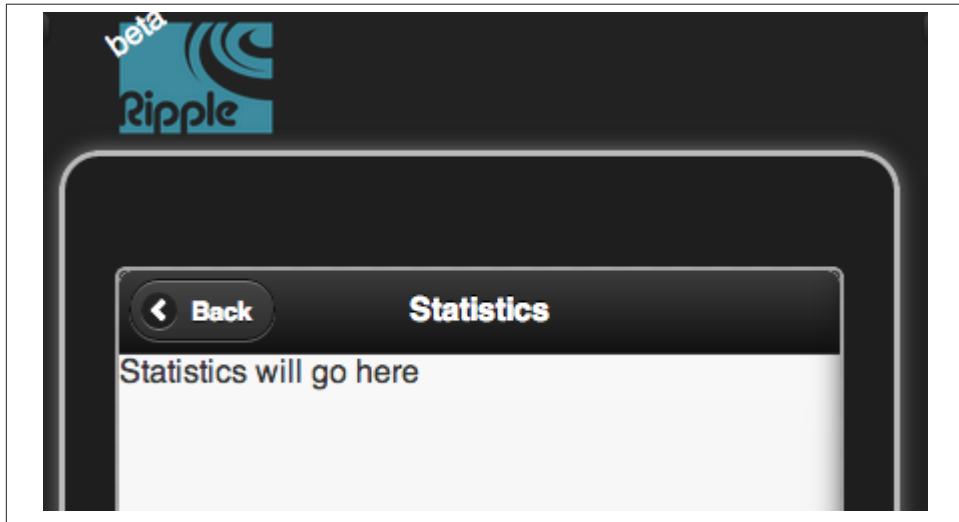


Figure 11-4. The Stats page with the Back button



The attribute `data-add-back-btn` works the same way in both the multi-page and single-page cases. The back button will appear only if the current page is not the first one and there is a previous page to navigate to.

## Single-Page Template

Now let's re-arrange the code of the above sample using a single-page template. We'll create a folder pages, which can contain multiple HTML files - one per page. In our case, we'll create there one file stats.html to represent the Statistics page. Accordingly, we'll remove the section marked as Page 2 from the main HTML file. The stats.html will look as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <div data-role="page" data-add-back-btn="true">
      <div data-role="header">
        <h1>Statistics</h1>
      </div>
    </div>
  </body>
</html>
```

```

<div data-role="content">
    Statistics data will go here
</div>
</body>
</html>

```

The main HTML file will contain only one home page, which is a Donate page in this example. The anchor tag will simply refer to the URL of the stats.html - there is no need to use hash tags or section ID any longer. In his case jQuery Mobile will load the stats.html using internal AJAX request. This is how the main page will look like:

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.css" type="text/css" media="screen,projection"/>
    </head>
    <body>
        <!-- Main page -->
        <div data-role="page" id="Donate">
            <div data-role="header">
                <h1>Donate</h1>
            </div>
            <!-- A Link to the second page -->
            <a href="pages/stats.html">Show Stats</a>
        </div>

        <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
        <script src="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.js"></script>
    </body>
</html>

```

Running this version of our simple two-page application will produce the same results and the second page will look exactly as in [Figure 11-4](#).

If you use single-page documents, the name of the file with the page will be added to the URL. For example, when the Stats page is open the browser's URL may look like this:

<http://127.0.0.1/pages/stats.html>

### Multi or Single-Page Template

So which template style should you use? Both have their pros and cons. If the code base of your application is large, use single-page template. The code will be split into multiple pages, will be easier to read and will give you a feeling of being modular without implementing any additional libraries for cutting the application into pieces. The home

page of the application comes quicker because you don't need to load the entire code base.

This all sounds good, but be aware that with single-page templates whenever you'll navigate from one page to another your mobile device makes a new request to the server. They user will see the wait cursor until the to-page has not arrived to the device. Even if the size of each page is small, additional requests to the server are costlier with mobile devices as they need another second just re-establish a radio link to the cell tower. After the communication with the server is done, the phone lowers its power consumption. The new request to the server for loading the page will start with increasing the power consumption again. Hence using the multi-page template may provide smoother navigation.

On the other hand, there is a way to **pre-fetch pages** into the DOM even in a single-page mode so the number of the server request id minimized. This can be done either with the HTML attribute `data-prefetch="true"` or programmatically using `$.mobile.loadPage()`. You can also ask the browser to cache previously visited pages with `$.mobile.page.prototype.options.domCache = true;`.

So what's the verdict? Test your application in both single and multi-page modes and see what's work best.

## Progressive Enhancement

Web developers use technique called *progressive enhancement*, especially in the mobile field. The idea is simple - first make sure that the basic functionality works in any browser, and then apply bells and whistles to make the application as fancy as possible using CSS and or framework-specific enhancements.

But what if you decide to go the opposite route and take a nice looking UI and remove its awesomeness? For instance, delete `<script>` and `<link>` tags from the above html file and open it in the Web browser - we are testing a situation when, for whatever reason, we need to remove the jQuery Mobile from our code base. The code still works! You'll see the first page, clicking on the link will open the second page. You'll lose the styling and that nice-looking Back button, but you can still use the browser's Back button. The Web browser ignores custom `data-` attributes without breaking anything.

This wouldn't be the case if we'd be using the multi-page template, where each page is a `<div>` or an `<article>` in the same HTML file. With multi-page template the Web browser would open all pages at once - one under another.

Here's another example. With jQuery Mobile you can create a button in many ways. There are multiple examples in the **Buttons section** of product documentation. The code below will produce five buttons, which will look the same, just the labels are different:

```
<a href="http://cnn.com" data-role="button">Anchor</a>
<form action="http://cnn.com">
    <button>Click me</button>
    <input type="button" value="Input">
    <input type="submit" value="Submit">
    <input type="reset" value="Reset">
</form>
```

If you chose to use the anchor link with `data-role="button"` and then remove the `<script>` tag that includes the code of jQuery Mobile library, the anchor tag will still work as a standard HTML link. It won't look as a button, but it will function as expected.

When you're making a decision about using any particular framework or library, ask yourself a question, "How easy it is to remove the framework from the application code if it doesn't deliver as expected". On several occasions the authors of this book were invited to help with projects, where the first task was removal of a wrongly-selected framework from the application code. Such surgery usually lasts at least two weeks. jQuery Mobile is non overly intrusive and is easily removable.

## Persistent Toolbars

One of the ways to arrange navigation is to add persistent toolbars that never go away while your application is running. You can add such a toolbar in the footer or header area or in both. We'll create a simple example illustrating this technique by adding a a navbar to the footer area of the application. Let's say, your application has a starting page and four other pages that can be selected by the user. [Figure 11-5.](#) shows the initial view of the application.



Figure 11-10 jQuery Mobile: the footer

If the user taps on one of the four pages in the footer, the program has to replace the starting page with the selected one, and the title of the selected page in the footer has to be highlighted. If you're reading the electronic version of this book you'll see in [Figure 11-6](#) that the rectangular area for Page #2 in the footer got the blue background. In the printed version of the book the different the background colors may not so obvious, but you have to trust us on this or run the code sample on your own. Besides, we'll be highlighting the selected page in a similar way while working on the prototype of the Save The Child application as per the mockups shown in the section "Prototyping Mobile Version".

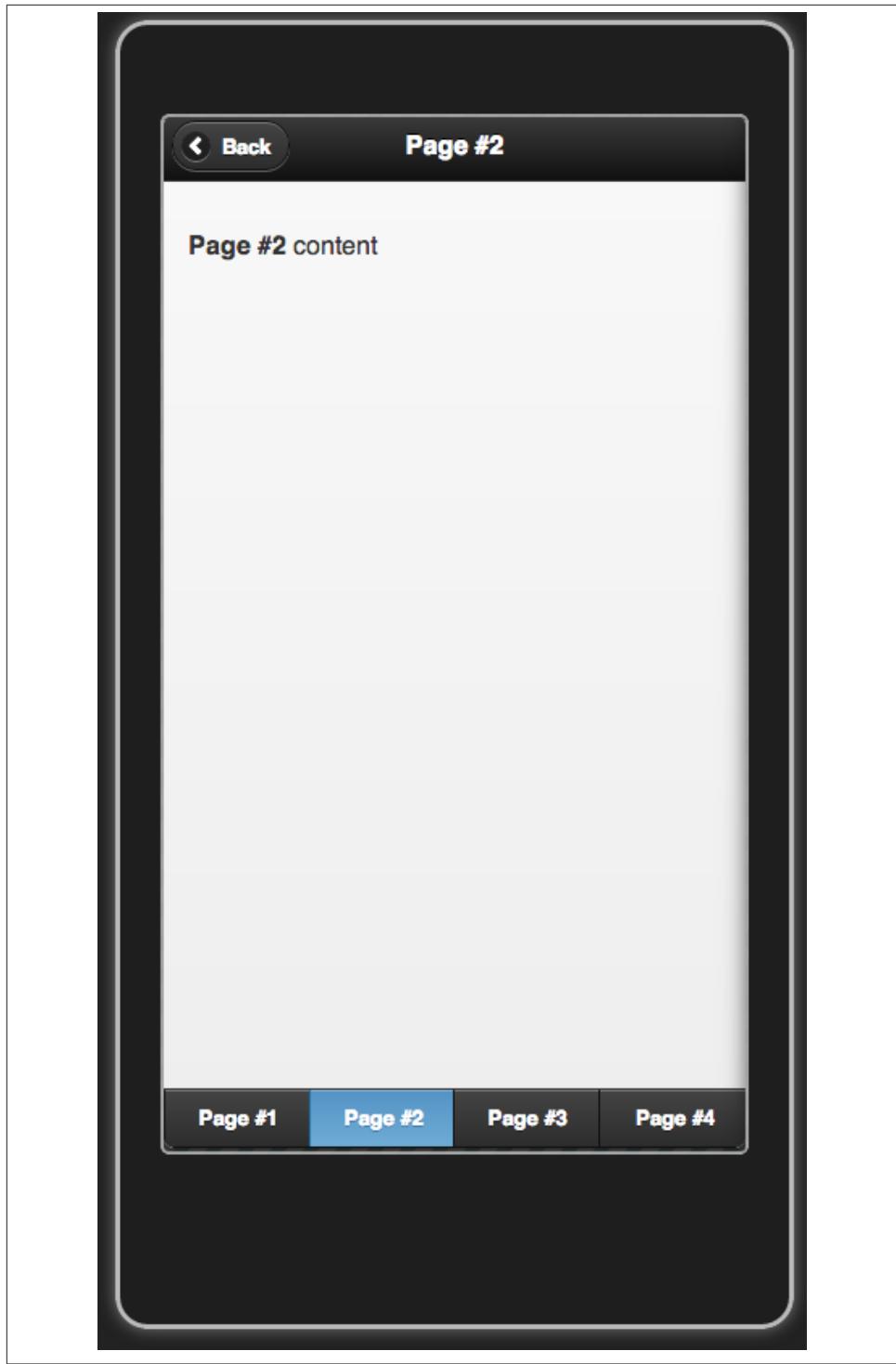


Figure 11-10 Chapter 11 Application

In jQuery Mobile implementing persistent toolbars is simple. The content of each of the page has to be located in a separate file and each of them has to have the footer and header with *the same data\_id*. Below is the code of the file page2.html, but page1, page3, and page 4 look similar - check them out in the source code that comes with the book.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <div data-role="page" data-add-back-btn="true">
      <div data-role="header" data-position="fixed"
           data-tap-toggle="false" data-id="persistent-header"> ①
        <h1>Page #2</h1>
      </div><!-- /header -->
      <div data-role="content" >
        <p>
          <b>Page #2</b> content
        </p>
      </div><!-- /content -->
      <div data-role="footer" data-position="fixed"
           data-tap-toggle="false" data-id="persistent-footer"> ②
        <div data-role="navbar">
          <ul>
            <li>
              <a href="page-1.html" data-transition="slideup">Page #1</a> ③
            </li>
            <li>
              <a href="#" class="ui-state-persist">Page #2</a> ④
            </li>
            <li>
              <a href="page-3.html" data-transition="slideup">Page #3</a>
            </li>
            <li>
              <a href="page-4.html" data-transition="slideup">Page #4</a>
            </li>
          </ul>
        </div><!-- /navbar -->
      </div><!-- /footer -->
    </div><!-- /page -->
  </body>
</html>
```

- ① To prevent the toolbar from being scrolled away from the screen we use `data-position="fixed"`. The attribute `data-tap-toggle="false"` disables the ability to remove the toolbar from the screen by tapping on the screen.
- ② The footer of page1, page2, page3, and page4 will have the same `data-id="persistent-footer"`.

- ③ While replacing the current page with another one, apply the transition effect so the page appears by sliding from the bottom up: `data-transition="slideup"`. Note that the anchor tags are automatically styled as buttons just because they are placed in the `navbar` container.
- ④ Since the Page 2 is already shown on the screen, tapping on the button “Page #2” in the navigation bar should not change the page, hence `href="#"`. The `class="ui-state-persist"` makes the framework to restore the active state each time when the existing in the DOM page is shown. The file `page3.html` will have a similar anchor for the button “Page #3” and so on.

The code of the main page `index.html` is shown below - it also defines the header, content, and footer areas:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1,
      user-scalable=no,maximum-scale=1">
    <title>Single-page template - start page</title>
    <link rel="stylesheet" href="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.css">
  </head>
  <body>

    <div data-role="page">
      <div data-role="header" data-position="fixed"
          data-tap-toggle="false" data-id="persistent-header">
        <h1>Start page</h1>
      </div>

      <div data-role="content" >
        <p>
          Single Page template. Start page content.
        </p>
      </div>

      <div data-role="footer" data-position="fixed"
          data-tap-toggle="false" data-id="persistent-footer">
        <div data-role="navbar">
          <ul>
            <li>
              <a href="pages/page-1.html" data-transition="slideup">Page #1</a>
            </li>
            <li>
              <a href="pages/page-2.html" data-transition="slideup">Page #2</a>
            </li>
            <li>
              <a href="pages/page-3.html" data-transition="slideup">Page #3</a>
            </li>
          </ul>
        </div>
      </div>
    </div>
  </body>
</html>
```

```

<li>
    <a href="pages/page-4.html" data-transition="slideup">Page #4</a>
</li>
</ul>
</div><!-- /navbar -->
</div><!-- /footer -->
</div><!-- /page -->

<script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
<script src="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.js"></script>
</body>
</html>

```



To avoid repeating the same footer in each HTML page, you may write a JavaScript function that will append the footer to each page on the `pagecreate` event. You can also consider using [HTML templating](#) to declare HTML fragments that are parsed on page load, but can be instantiated later on during the runtime. In particular, we can recommend you the [Handlebars](#), which lets you build semantic templates easily.

## Programmatic Navigation

The above code samples were illustration page navigation as a response to the user's action. Sometimes you need to change pages programmatically as a result of certain event, and the method `$.mobile.changePage()` can do this.

This method requires at least one parameter - the string defining the change-to-page, for example:

```
$.mobile.changePage("pages/stats.html");
```

But you can also invoke this method with a second parameter, which is an object, where you can specify such parameters as `data` - the data to send with AJAX page request, `changeHash` - a boolean to control if the hash in the URL should be updated and some others. For example, the following code sample changes the page using post request (`type: "post"`) and the new page should replace the current page in the browser's history (`changeHash: false`).

```
$.mobile.changePage("pages/stats.html", {
    type: "post",
    changeHash: false
});
```

## Save The Child with jQuery Mobile

After the brief introduction to jQuery Mobile library we (and you) are eager to start hands-on coding. The mobile version of the Save The Child won't show all the features

of this application. It'll be sliced into a set of screens (pages), and the user will see one page at a time.



You can test the working jQuery Mobile version of our sample application at <http://savesickchild.org:8080/ssc-jquery-mobile/>.

## Prototyping Mobile Version

It's time to go back to Jerry, the designer and his favorite prototyping tool Balsamiq Mockups introduced in Chapter 3. Designs and layouts for each screen of the mobile version are shown below as one of the images taken from Balsamiq tool. This is not a complete set of images as it doesn't include layouts for tablets. In this book we will test only the mobile devices with screen sizes of 640x960 and 320x480 pixels.

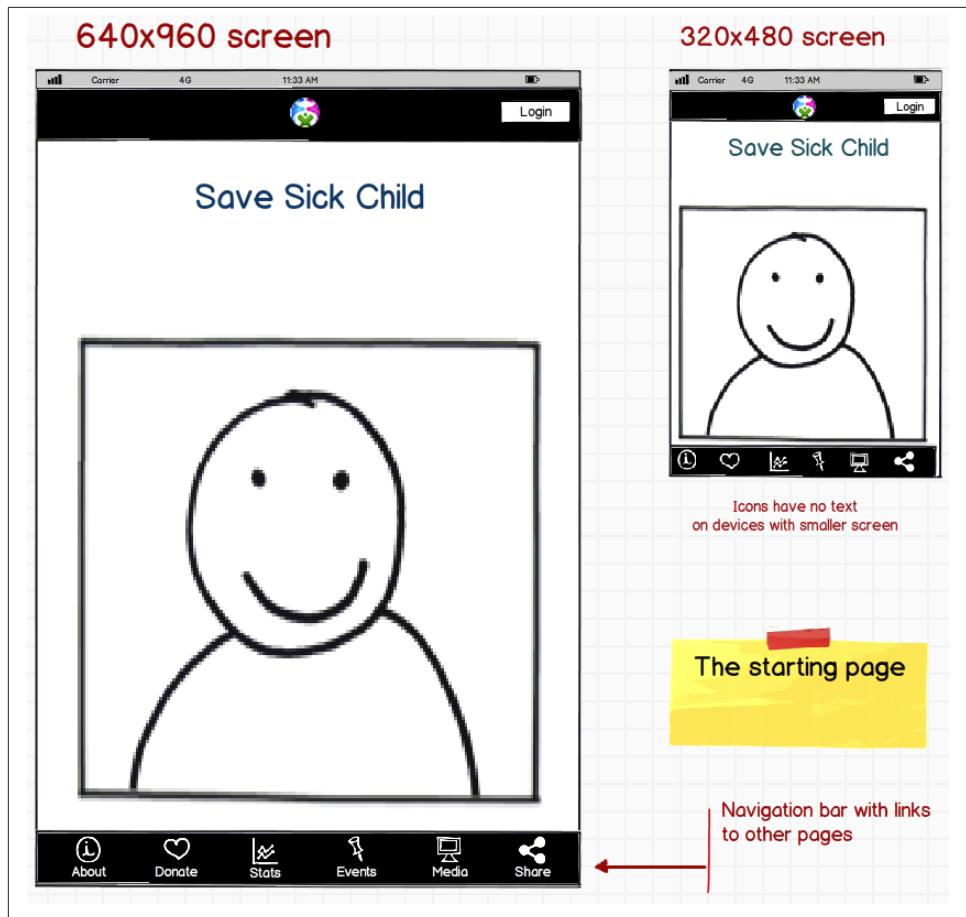


Figure 11-7. The Starting page (portrait)

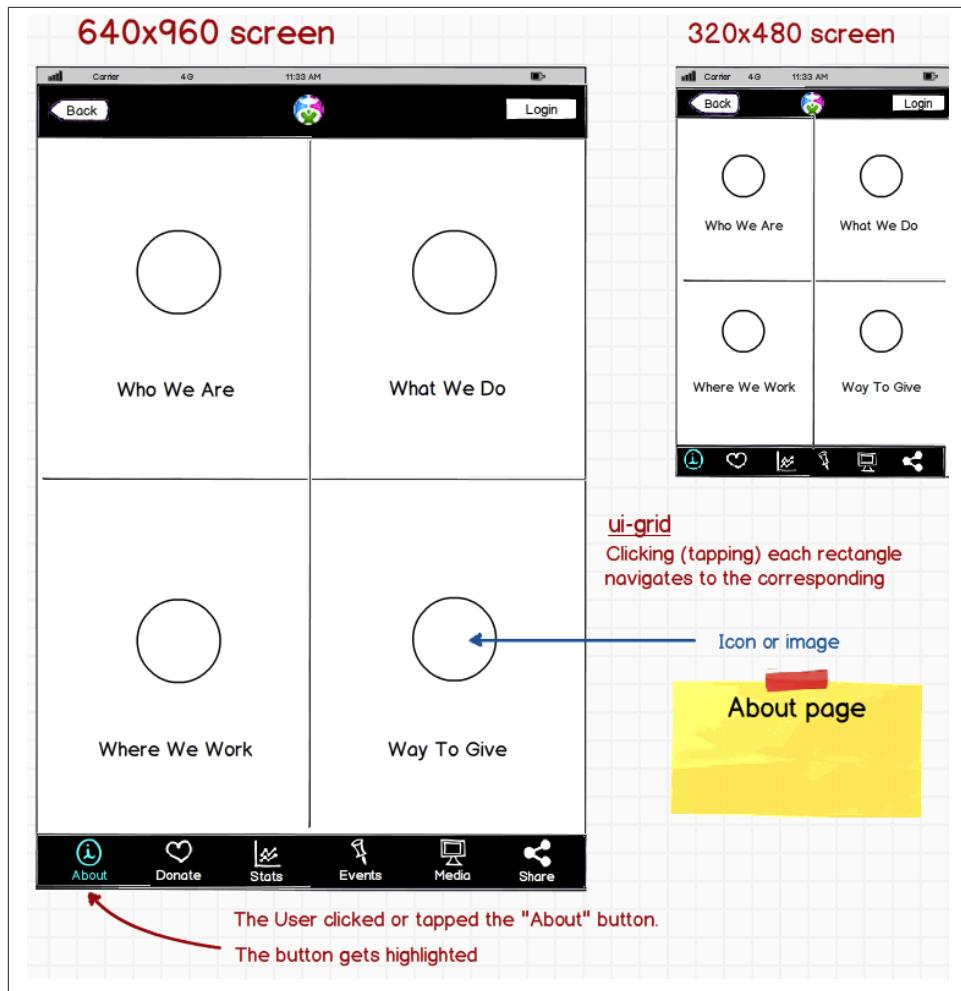
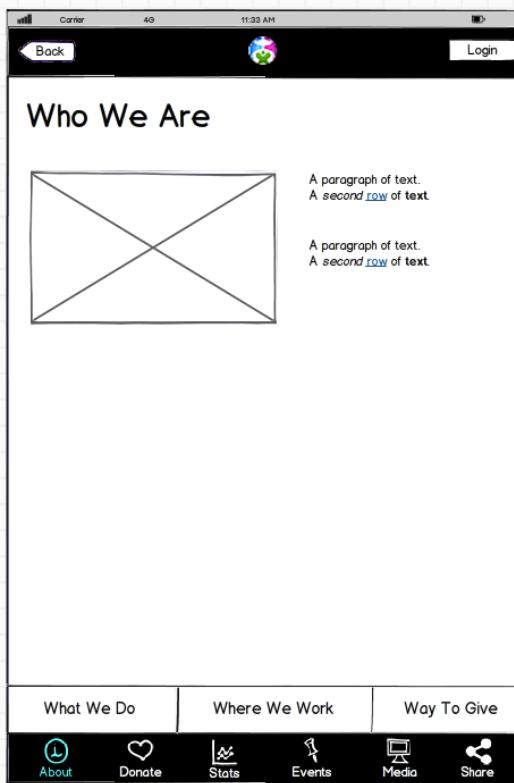
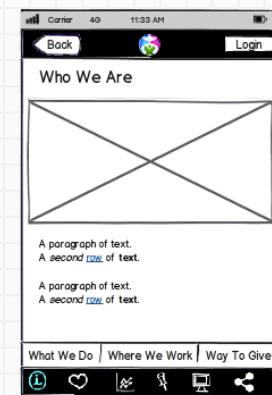


Figure 11-8. The About page (portrait)

640x960 screen



320x480 screen



From About to Who We Are

What We Do, Where We Work  
and Way To Give should  
have same layout

Additional nav-bar with links  
to other About sections

Figure 11-9. The Who We Are section of About page (portrait)

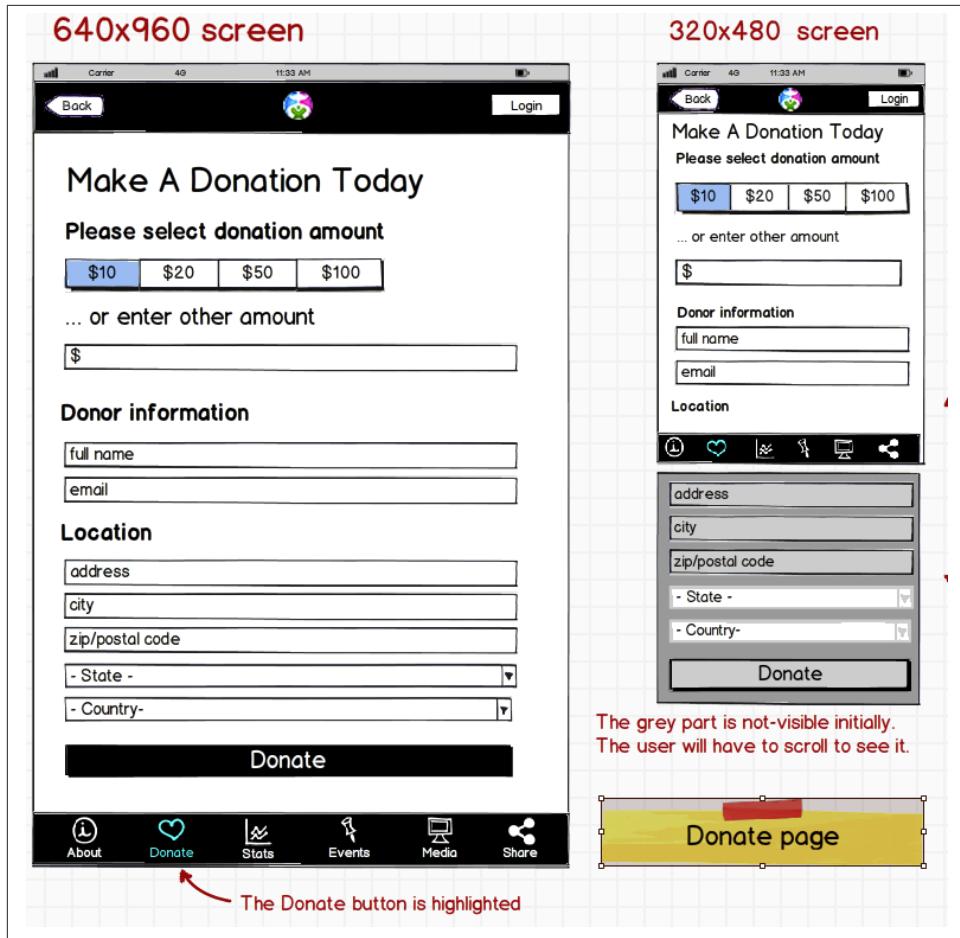


Figure 11-10. The Donate page (portrait)

The small screen version of the above Donate page illustrates a term *Above the Fold* used by Web designers. This term originated in the newspaper business where the first half of the folded newspaper contained the most important headlines - something that the potential buyer would notice immediately. In Web design the *Above the Fold* means the first page that the user can see without the need to scroll. But if with newspapers people know that there is something to read below the fold, in Web design people may not know that the scrolling could reveal more information. In this particular case, there is a chance that a user with a 320x480 screen may not immediately understand that to see the Donate he needs to scroll.

In general, it's a good idea to minimize the number of form fields that the user must manually fill out. Invest into analyzing the forms used in your application. See you can

design the form smarter: auto-populate some of the fields and show/hide fields based on the user's inputs.



If you have a long form that has to be shown on a small screen, split it into several `<div data-role="page">` sections all located inside the `<form>` tag. Arrange the navigation between these sections as it was done for multi-page documents in the section “Adding Page Navigation” above.

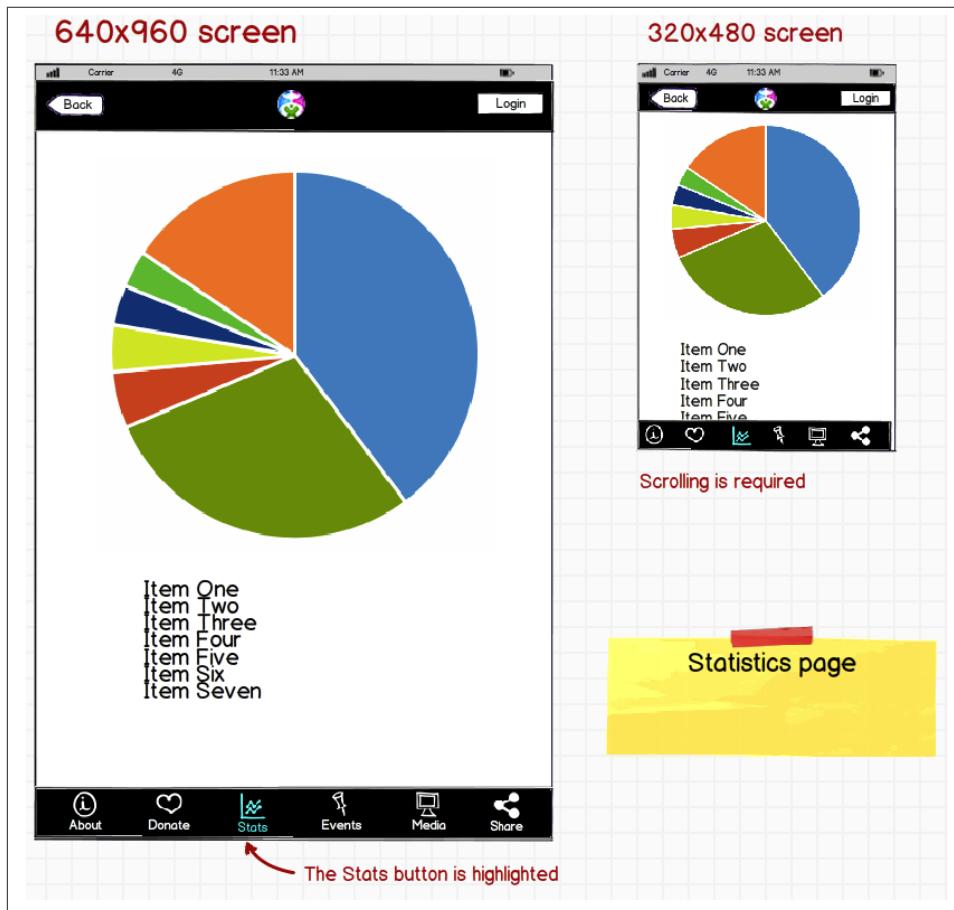
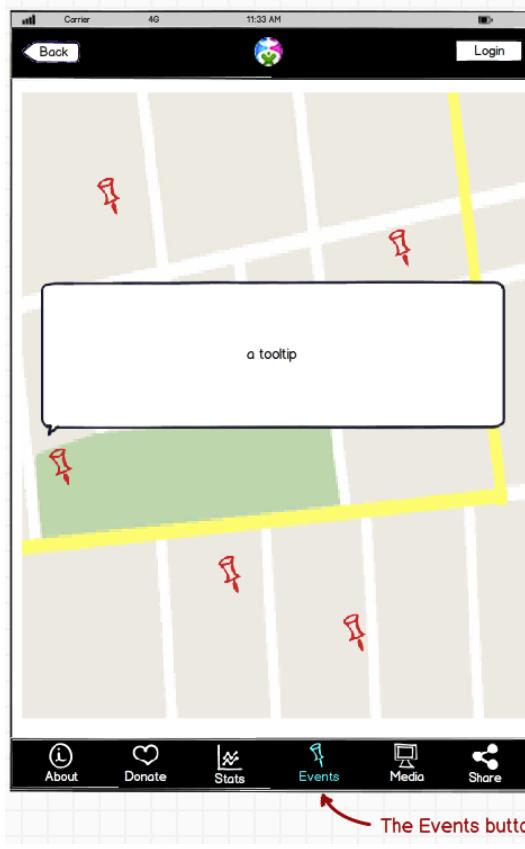


Figure 11-11. The Statistics page (portrait)

640x960 screen



320x480 screen

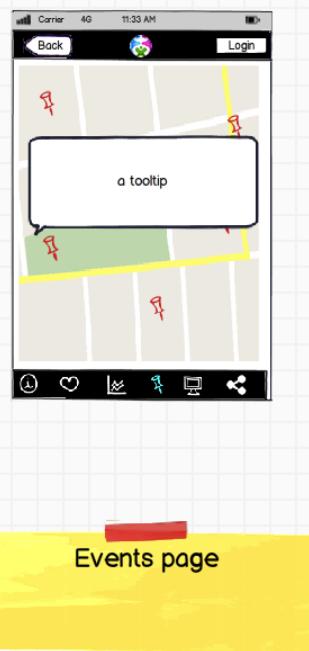
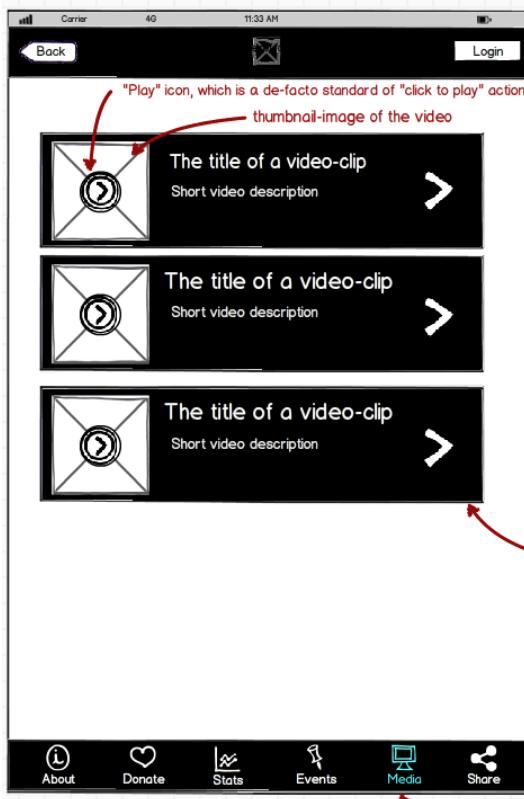


Figure 11-12. The Events page (portrait)

640x960 screen



320x480 screen



Listview

A tap (click) on list item will open a video in a popup layer (<div>)

"Media" app's screen  
is active

Figure 11-13. The Media page (portrait)

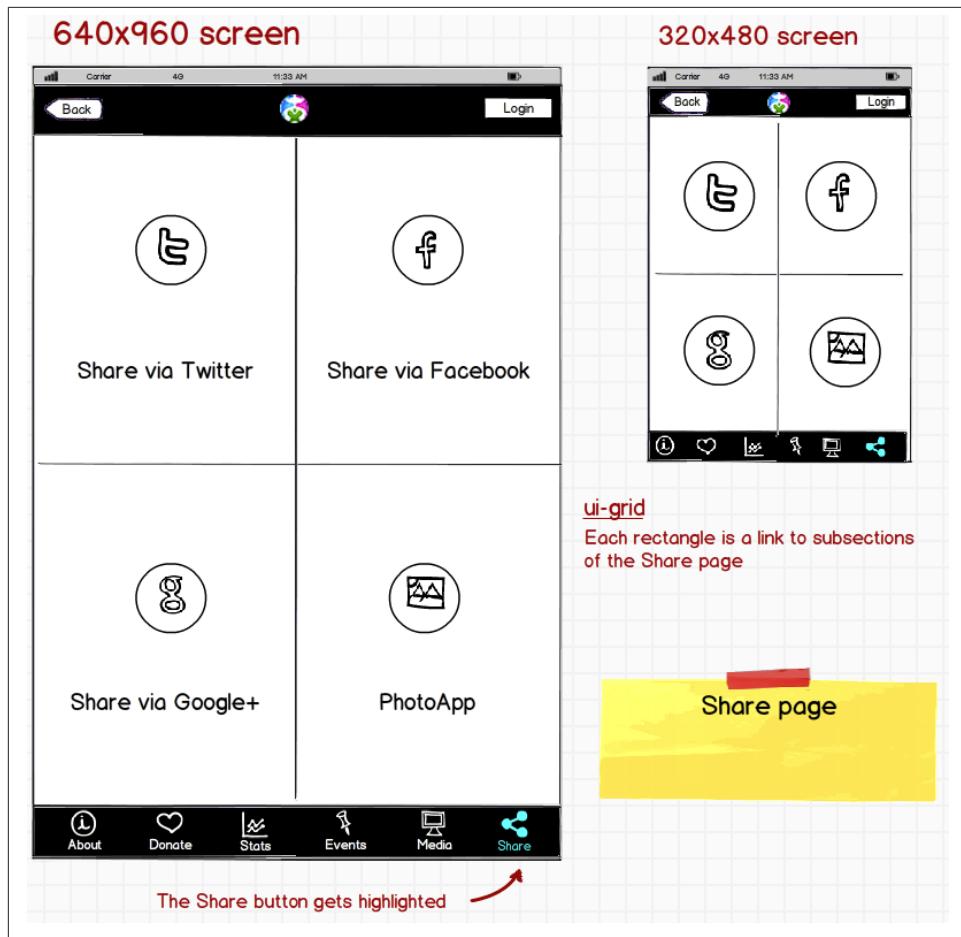


Figure 11-14. The Share page (portrait)

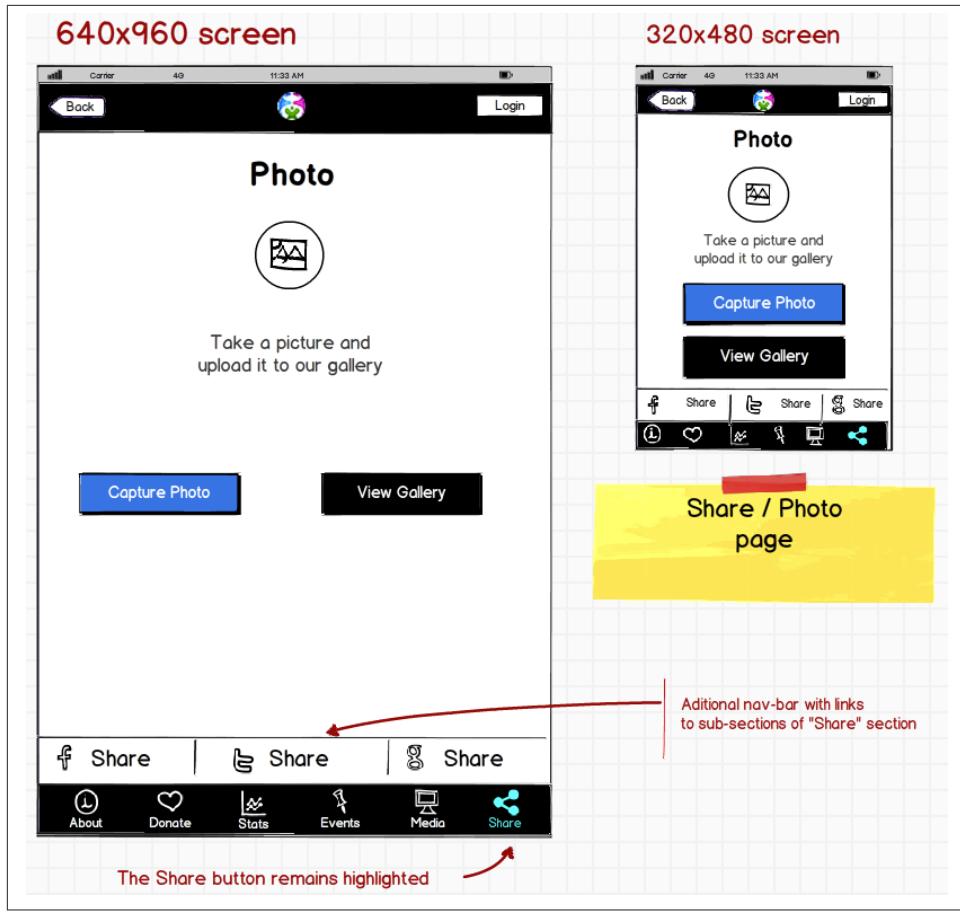


Figure 11-15. The Share/Photo page for Chapter 14 (portrait)

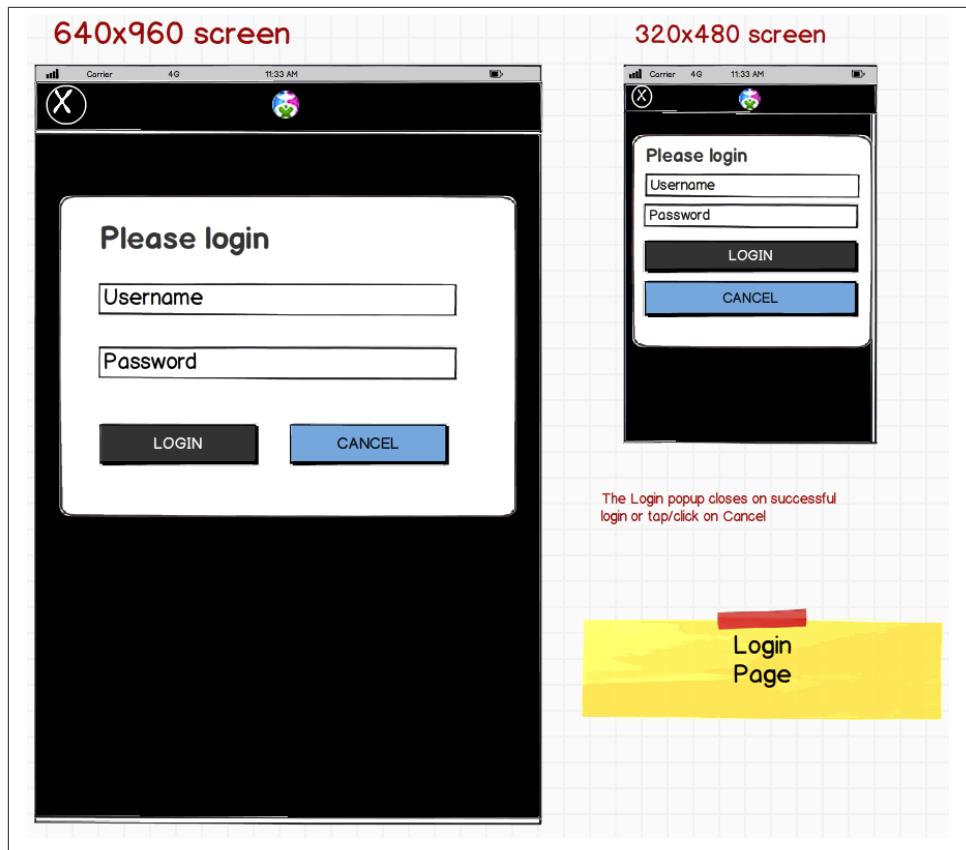


Figure 11-16. The Login popup (portrait)

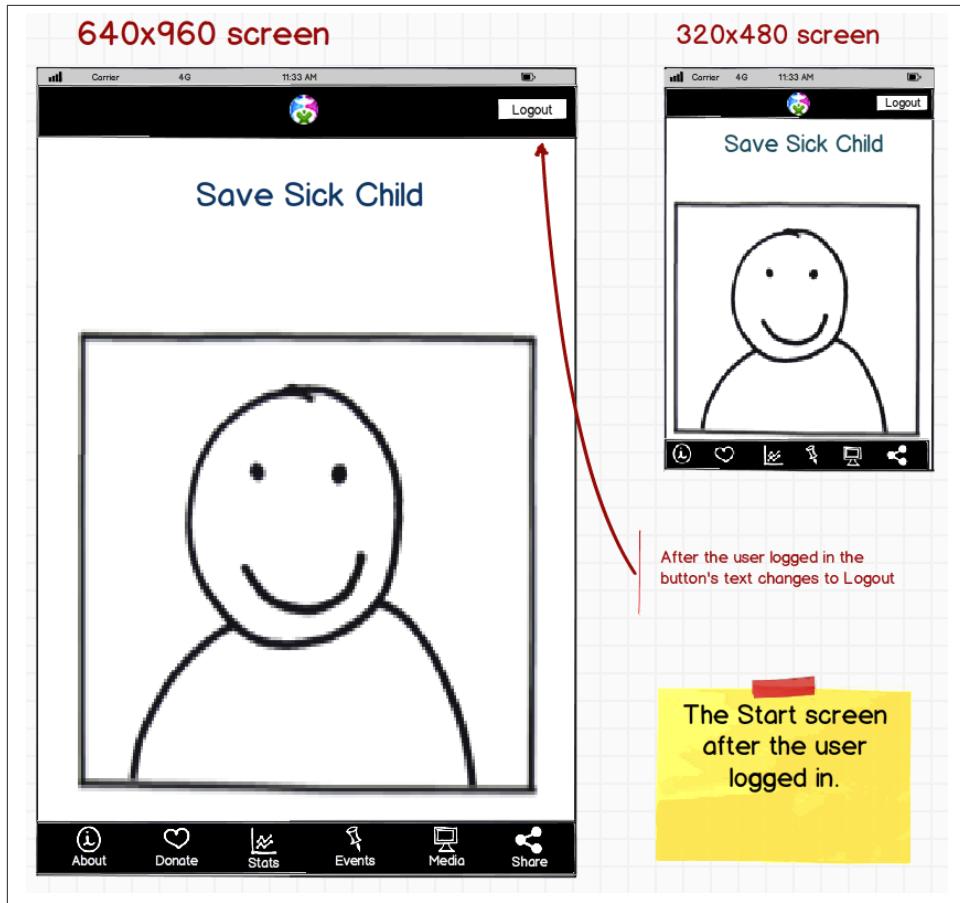


Figure 11-17. After the user logged in

This prototype will be used for the developing both jQuery Mobile and Sencha Touch versions of our Save The Child application. We've also included the design for the page that will integrate with the photo camera of the device (see [Figure 11-15](#)) - this functionality will be implemented in the last chapter dedicated to hybrid applications.

All of the above images show UI layouts when the mobile device is in the portrait mode, but you should ask your Web designer to prepare the mockups for the landscape mode too. Below are the couple of snapshots prepared by our Web designer Jerry.

640x960 screen

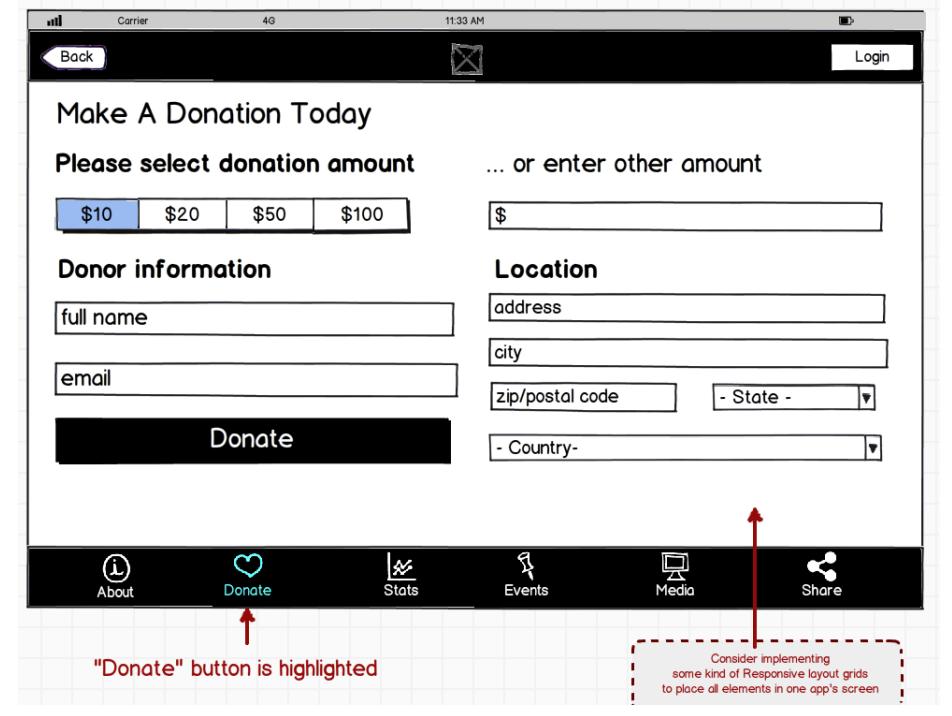


Figure 11-18. The Donate page (landscape, 640x960)

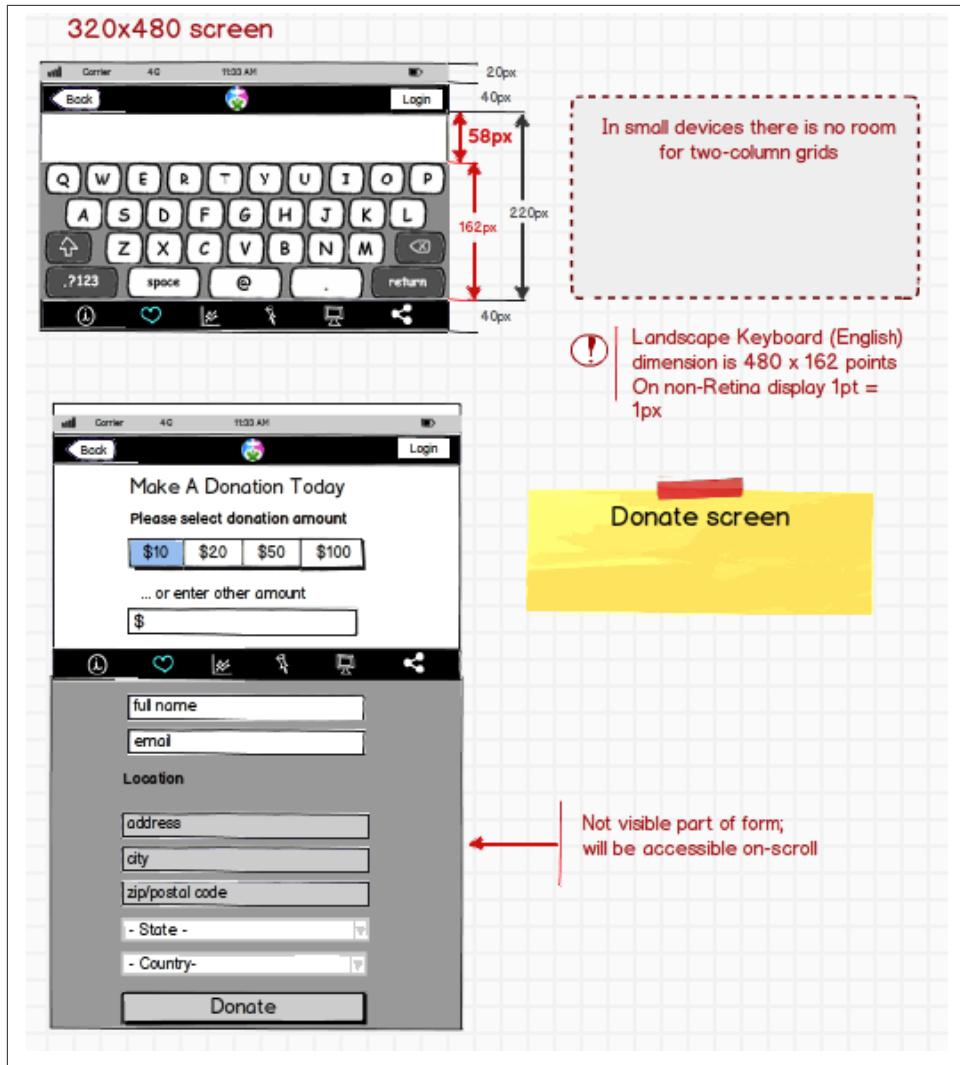


Figure 11-19. The Donate page (landscape, 320x480)

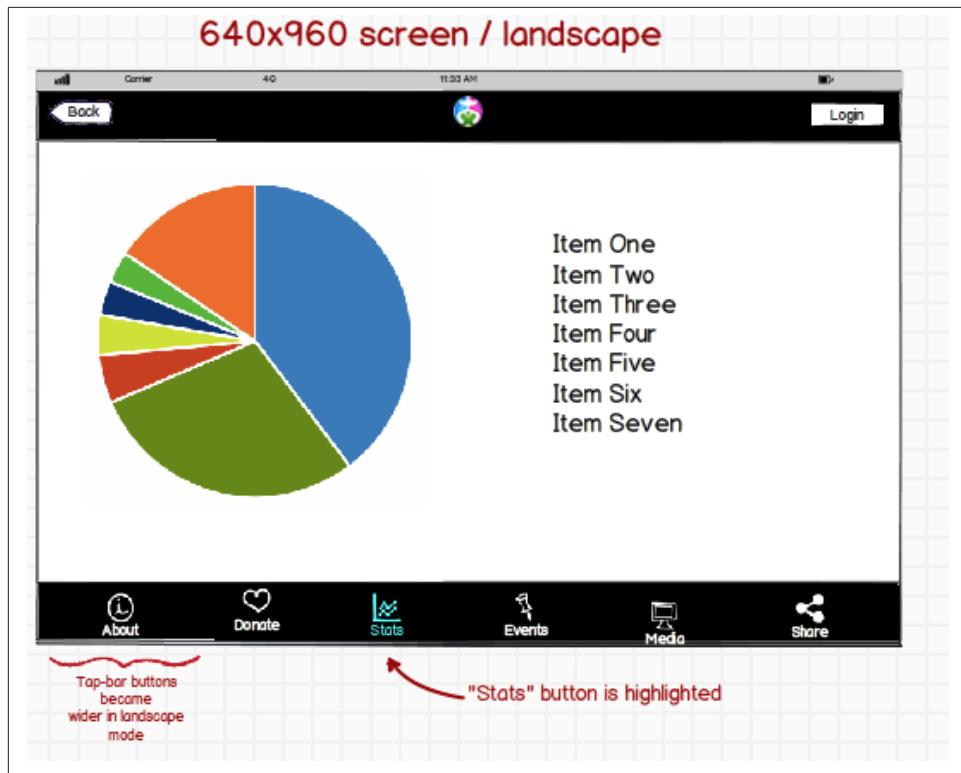


Figure 11-20. The Statistics page (landscape, 640x960)

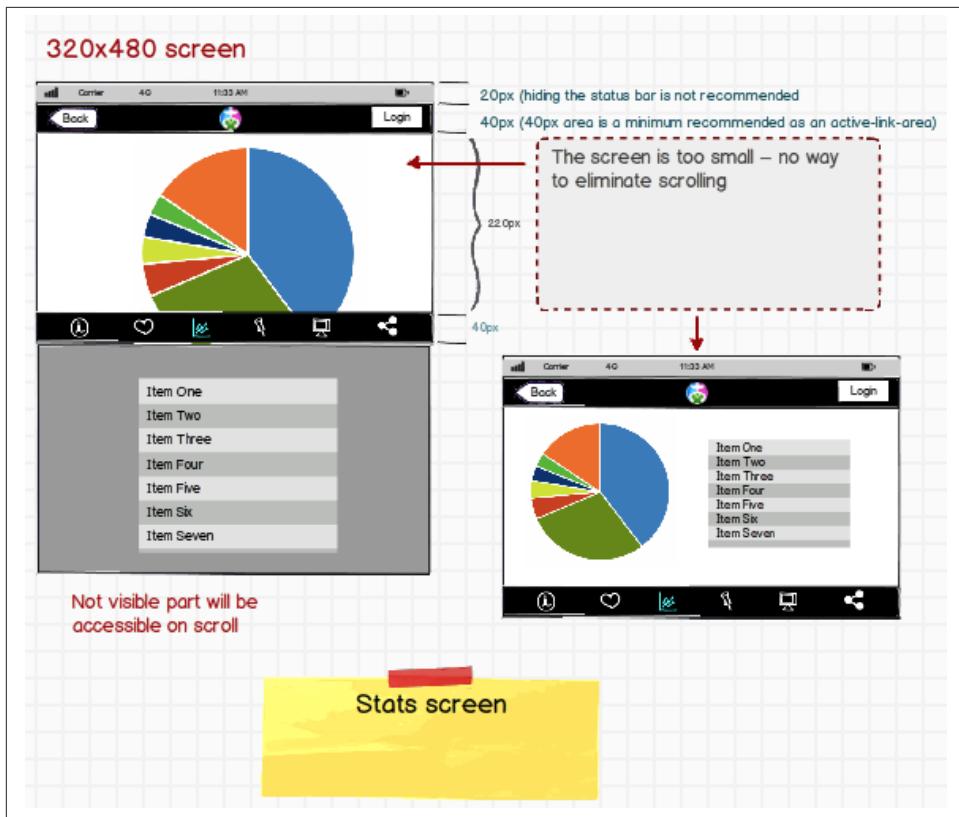


Figure 11-21. The Statistics page (landscape, 320x480)



If you want to add a link that will offer to dial a phone number, use the `tel:` scheme, for example: `<a href="tel:+12125551212">Call us</a>`. If you want the phone to look like a button, add the attribute `data-role="button"` to the anchor tag.

## The Project Structure and Navigation

This time the Save The Child project structure will look as in [Figure 11-22](#). We are using the single-page template here. The `index.html` is the home page of our application. All other pages are located in the `pages` folder. The JavaScript code is in the folder `js`, and fonts, images and CSS file are in the folder `assets`. We'll use the same JSON files as in the previous versions of this application, and they are located in the folder `data`.

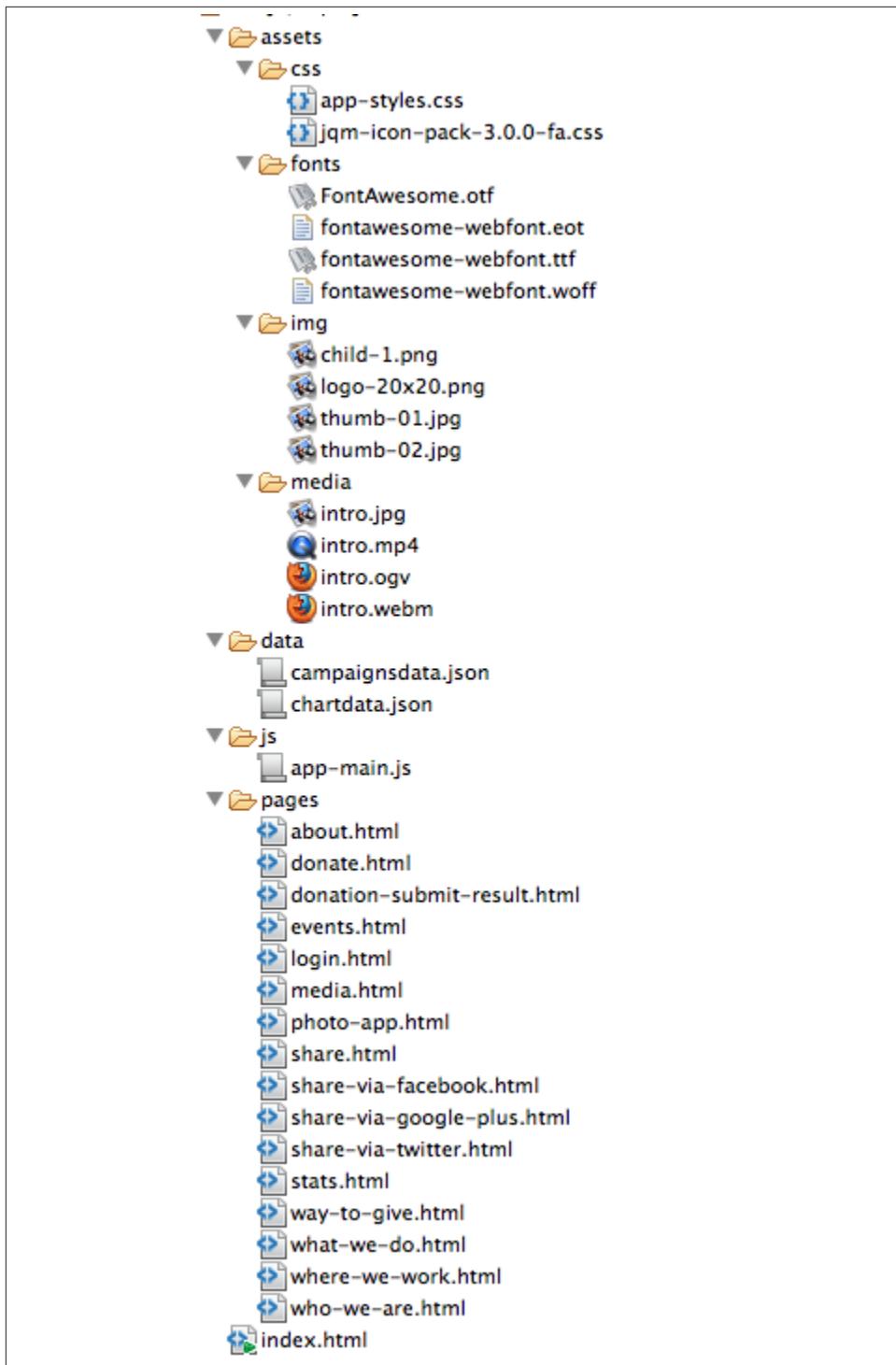


Figure 11.22 The project structure

Let's start implementing navigation based using the techniques described earlier in the section "Persistent Toolbars". The source code of the index.html is shown below. Note that we moved the <script> tags with jQuery Mobile code from that end of the <body> tag to the <head> section to avoid a popup of a non-styled page on the initial load of the application.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1,user-scalable=no,
maximum-scale=1">
    // ①
    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta name="apple-mobile-web-app-status-bar-style" content="black">

    <title>Save The Child</title>

    <link rel="stylesheet" href="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.css"
    <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
    <script src="http://code.jquery.com/mobile/1.3.1/jquery.mobile-1.3.1.min.js"></script>

    // ②
    <link rel="stylesheet" href="assets/css/jqm-icon-pack-3.0.0-fa.css" />

    <link rel="stylesheet" href="assets/css/app-styles.css" /> // ③
  </head>
  <body>

    <div data-role="page">
    // ④
      <div data-role="header" data-position="fixed" data-tap-toggle="false"
data-id="persistent-header">
        <a href="pages/login.html" data-icon="chevron-down" data-iconpos="right"
class="ui-btn-right login-btn" data-rel="dialog">Login</a>
        <h1> <
      </div>
    // ⑤
    <div data-role="content" >
      <h2>Save The Child</h2>
      <p>
        <b>Start page</b> content.
      </p>
    </div>
    // ⑥
    <div data-role="footer" data-position="fixed" data-tap-toggle="false"
data-id="persistent-footer">
      <div data-role="navbar" class="ssc-navbar">
        <ul>
          <li>
            <a href="pages/about.html" data-iconshadow="false"
```

```

        data-icon="info-sign"
        data-transition="slideup">>About</a> // ⑦
    </li>
    <li>
        <a href="pages/donate.html" data-iconshadow="false" data-icon="heart"
        data-transition="slideup">Donate</a>
    </li>
    <li>
        <a href="pages/stats.html" data-iconshadow="false" data-icon="bar-chart"
        data-transition="slideup">Stats</a>
    </li>
    <li>
        <a href="pages/events.html" data-iconshadow="false" data-icon="map-marker"
        data-transition="slideup">Events</a>
    </li>
    <li>
        <a href="pages/media.html" data-iconshadow="false" data-icon="film"
        data-transition="slideup">Media</a>
    </li>
    <li>
        <a href="pages/share.html" data-iconshadow="false" data-icon="share"
        data-transition="slideup">Share</a>
    </li>
</ul>
</div><!-- /navbar -->
</div><!-- /footer -->
</div><!-- /page -->
<script src="js/app-main.js"></script>
</body>
</html>
```

- ➊ The metatags to request the full screen mode and black status bar on iOS devices. The main goal is to remove the browser's address bar. Some developers suggest JavaScript tricks like `window.scrollTo(0,1)`; (Google on it for details). But we are not aware of a reliable solution for a guaranteed full screen mode in Web applications on all devices.
- ➋ This project uses jQuery Mobile Icon Pack - an extension of standard jQuery Mobile icons.
- ➌ Our CSS will override some of the jQuery Mobile classes and add new styles specific to our application.
- ➍ The header shows a Login button and the application logo.
- ➎ The content of the main page should go here
- ➏ All the navigation buttons are located in the footer.

- 7 jQuery Mobile includes a number of icons that you can use by specifying their names in the `data-icon` attribute (read the Note on icons below). The icon position is controlled by the attribute `data-iconpos`. If you don't want to show text, use `data-iconpos="notext"`.

Figure 11-23 shows how the landing page of the Save The Child application will look in the Ripple Emulator. Run it and click on each of the buttons in the navigation bar.



Figure 11-28 The user interface on SSC home page

NOTE:

In this application we use **icon fonts** to be displayed on the navigation bar. The main advantage over using images for icons is that icon fonts are maintenance free. You don't need to resize and redraw icons. The disadvantage of the icon fonts is that they are single-colored, but for the navigation bar buttons having multi-colored images is not important.

In the above code we've been using the jQuery Mobile Icon Pack that's available on [GitHub](#). It's an adaptation of the Twitter Bootstrap's Font Awesome for jQuery Mobile. If you need fancier images for your mobile application, consider using [Glypish icons](#).

The content of our custom CSS file app-styles.css comes next.

```
/* First, we want to stop jQuery Mobile using it's standard images for icons. */

.ui-icon-plus, .ui-icon-minus, .ui-icon-delete, .ui-icon-arrow-r, .ui-icon-arrow-l,
.ui-icon-arrow-u, .ui-icon-arrow-d, .ui-icon-check, .ui-icon-gear,
.ui-icon-refresh, .ui-icon-forward, .ui-icon-back, .ui-icon-grid, .ui-icon-star, .ui-icon-alert,
.ui-icon-info, .ui-icon-home, .ui-icon-search, .ui-icon-searchfield:after, .ui-icon-checkbox-off,
.ui-icon-checkbox-on, .ui-icon-radio-off, .ui-icon-radio-on,
.ui-icon-email, .ui-icon-page, .ui-icon-question, .ui-icon-foursquare, .ui-icon-dollar,
.ui-icon-euro, .ui-icon-pound, .ui-icon-apple, .ui-icon-chat,
.ui-icon-trash, .ui-icon-mappin, .ui-icon-direction, .ui-icon-heart, .ui-icon-wrench,
.ui-icon-play, .ui-icon-pause, .ui-icon-stop, .ui-icon-person,
.ui-icon-music, .ui-icon-wifi, .ui-icon-phone, .ui-icon-power,
.ui-icon-lightning, .ui-icon-drink, .ui-icon-android {
    background-image: none !important;
}

/* Override the jQuery Mobile CSS class selectors with the icon fonts. Whenever you create custom

.ui-icon-arrow-l:before {
    content: "\f053";
    margin-top: 2px
}
.ui-icon-delete:before {
    content: "\f00d";
    margin-left: 3px;
    margin-top: -2px
}
.ui-icon-arrow-r:before {
    content: "\f054";
    padding-left: 2px;
}
.ui-icon-arrow-d:before {
    content: "\f078";
}
.ui-icon-home:before {
```

```

    content: "\f015";
}

.header-logo {
    vertical-align: middle;
    padding-right: 0.3em;
    margin-top: -2px;
}

/* Create some custom styles for the Save The Child application. */

.ssc-navbar .ui-btn-text {
    font-size: 0.9em
}

/* overwide, customize icons css */
.ssc-navbar .ui-icon {
    background: none !important;
    margin-top: 2px !important;
}
/* jQM allows not more than 5 items per line in navbar.
We need 6. Hence we should override the default CSS rule.
Each block will occupy 1/6 of the width: 16.66%
*/
.ssc-navbar .ui-block-a {
    width: 16.66% !important;
}
.ssc-navbar .ui-block-b {
    width: 16.66% !important;
}

.ssc-grid-nav {
    display: block;
    text-align: center;
    border-top: 1px solid #c0c0c0;
    text-decoration: none;
    color: #555 !important;
    overflow: hidden;
    box-sizing: border-box
}
.ssc-grid-nav:nth-child(odd) {
    border-right: 1px solid #c0c0c0;
}
.ssc-grid-item-icon {
    display: block;
    font-size: 2em;
    padding-bottom: 0.5em
}

```

## Selected Code Fragments

All the code that implements Save The Child with jQuery Mobile is available to download from the publisher of this book (see the URL in the Preface), and we're not going to include all program listings here. But we will show and comment selected code fragments that illustrate various features of jQuery Mobile.

### Grid Layouts

While testing this initial version of the Save The Child application, note that the content of the About and Share pages is implemented as in mockups shown on [Figure 11-8](#) and [Figure 11-14](#), which looks like grids. jQuery Mobile has several pre-defined layouts that will allow showing the content as rows and columns. Keep in mind that on small devices you should avoid displaying grids with multiple rows and columns as the data there will be hardly visible. But in our case the grid will contain just four large cells. The source code of the share.html followed by brief comments comes next (the code of the about.html looks similar).

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>

    <div data-role="page" data-add-back-btn="true" id="Share">
      <div class="ssc-grid-header" data-role="header" data-position="fixed"
           data-tap-toggle="false" data-id="persistent-header">
        <a href="login.html" data-icon="chevron-down" data-iconpos="right"
           class="ui-btn-right login-btn" data-rel="dialog">Login</a>
        <h1></h1>
      </div>

      <div data-role="content" style="padding:0">
        <div class="ui-grid-a" > // ①
          <div class="ui-block-a" > // ②
            <a href="#" class="ssc-grid-nav">
              <span class="ssc-grid-item-icon ui-icon-twitter"></span>
              <br/>
              Share via Twitter</a>
            </div>
          <div class="ui-block-b" >
            <a href="#" class="ssc-grid-nav">
              <span class="ssc-grid-item-icon ui-icon-facebook"></span>
              <br/>
              Share via Facebook</a>
            </div>
          <div class="ui-block-a" >
```

```

<a href="#" class="ssc-grid-nav">
<span class="ssc-grid-item-icon ui-icon-google-plus"></span>
<br/>
    Share via Google+</a>
</div>
<div class="ui-block-b">
    <a href="#" class="ssc-grid-nav">
        <span class="ssc-grid-item-icon ui-icon-camera"></span>
        <br/>
        Photo App</a>
    </div>
</div>
</div>

<div class="ssc-grid-footer" data-role="footer" data-position="fixed" data-tap-toggle="false"
data-id="persistent-footer">
    <div data-role="navbar" class="ssc-navbar">
        <ul>
            <li>
                <a href="about.html" data-iconshadow="false" data-icon="info-sign"
                    data-transition="slideup">About</a>
            </li>
            <li>
                <a href="donate.html" data-iconshadow="false" data-icon="heart"
                    data-transition="slideup">Donate</a>
            </li>
            <li>
                <a href="stats.html" data-iconshadow="false" data-icon="bar-chart"
                    data-transition="slideup">Stats</a>
            </li>
            <li>
                <a href="events.html" data-iconshadow="false" data-icon="map-marker"
                    data-transition="slideup">Events</a>
            </li>
            <li>
                <a href="media.html" data-iconshadow="false" data-icon="film"
                    data-transition="slideup">Media</a>
            </li>
            <li>
                <a href="#" data-iconshadow="false" data-icon="share"
                    class="ui-state-persist">Share</a>
            </li>
        </ul>
    </div><!-- /navbar -->
</div><!-- /footer -->
</div><!-- /page -->
</body>
</html>

```

- ① The grid from [Figure 11-8](#) is implemented using jQuery Mobile multi-column layout using `ui-grid` classes (see explanations below).

- ② Each of the cells in the grid is classes by the `ui-block-a` for the first grid row and `ui-block-b` for the second one. Hence “Share via Twitter” is in the left cell, and “Share via Facebook is on the right”.

There are four **preset configurations** for grids containing two, three, four, and five columns called `ui-grid-a`, `ui-grid-b`, `ui-grid-c`, and `ui-grid-d` respectively. The Stats and About screens split into four sections, which can be laid out in two columns with `ui-grid-a`. With two-column layout, each of the column gets 50% of the width, with three-column layout - about 33% et al.

Each of the cells is laid out with the class that’s named with `ui-block-` followed by the corresponding letter, e.g. `ui-block-c` for the cells located in the third column. **Figure 11-24** is a fragment from jQuery Mobile documentation, and it serves as a good illustration of the grid presets.

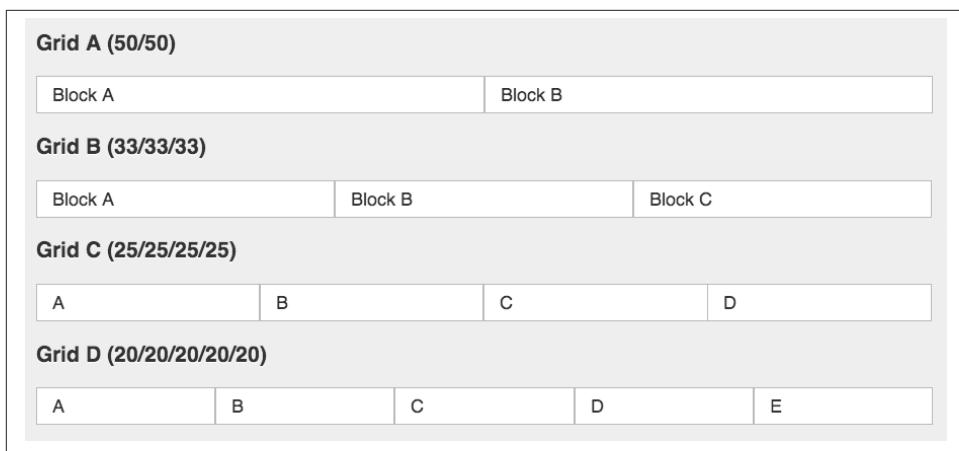


Figure 11-24. Preset grid layouts

The class `.ui-responsive` allows to set breakpoints to grids that are less than 35em (560px) wide.

### Control Groups

In the Donation screen, there us a section to allow the user to select one of the donation amounts. This is a good example of a set of UI controls that belong to the same group. In the desktop version of the application we’ve been using radio buttons grouped by the same `name` attribute like `<input type="radio" name = "amount" ...>`. Revisit Chapter 3 and you’ll find the complete code example in the section titled “The Donate Section”.

jQuery Mobile has a concept of **control groups** that comes handy in grouping and styling components. The code looks very similar, but now it's wrapped in the `<fieldset>` container with the `data-role="controlgroup"`.

```
<div class="donation-form-section">
  <label class="donation-heading">Please select donation amount</label>

  <fieldset data-role="controlgroup" data-type="horizontal" id="radio-container">

    <input type="radio" name="amount" id="d10" value="10"/>
    <label for="d10">$10</label>
    <input type="radio" name="amount" id="d20" value="20" />
    <label for="d20">$20</label>
    <input type="radio" name="amount" id="d50" checked="checked" value="50" />
    <label for="d50">$50</label>
    <input type="radio" name="amount" id="d100" value="100" />
    <label for="d100">$100</label>

  </fieldset>
  <label class="donation-heading">...or enter other amount</label>

  <input id="customAmount" name="amount" value="" type="text"
    autocomplete="off" placeholder="$"/>
```

jQuery Mobile will render this code as shown in [Figure 11-25](#). The buttons are laid out horizontally because of the attribute `data-type="horizontal"`. If you don't like the default styling of the radio buttons input fields, feel free to specify the appropriate `data-theme` either for the entire group or for each input field.

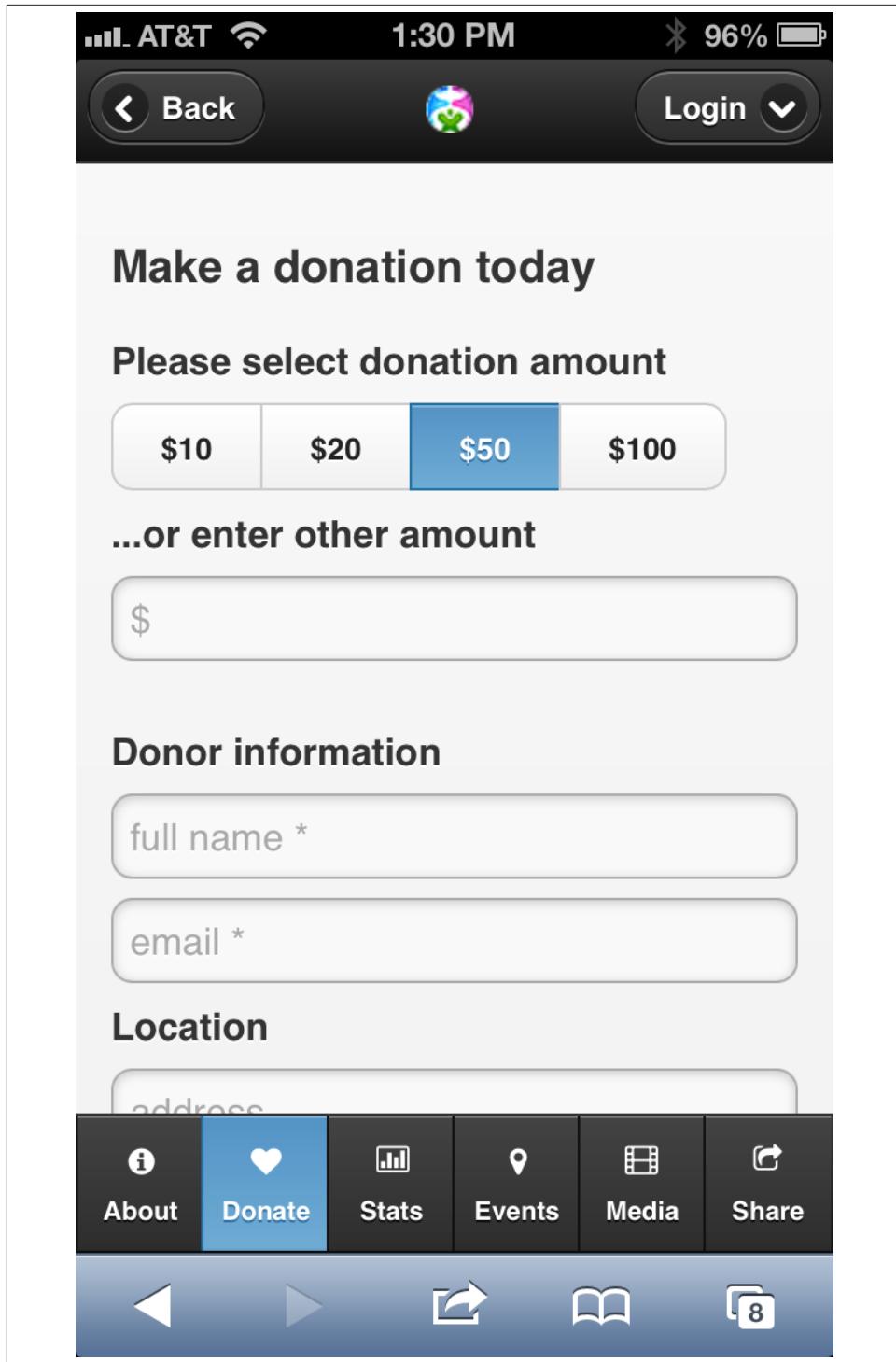


Figure 11-25. Controlgroup for donation amount

## **Dropdowns and Collapsibles**

Having an ability to use the minimum amount of screen real estate is especially important in mobile applications. Such controls can drop down or popup a list with some information when the user taps on a smaller component. Controls that we know as comboboxes or dropdowns in the desktop applications look different on the mobile devices, but the good news is that you don't need to do any special coding to display a fancy-looking dropdown on the iPhone shown on [Figure 11-26](#). Just use the HTML tag `<select>`, and the mobile browser will render it with a native look on the user's device.

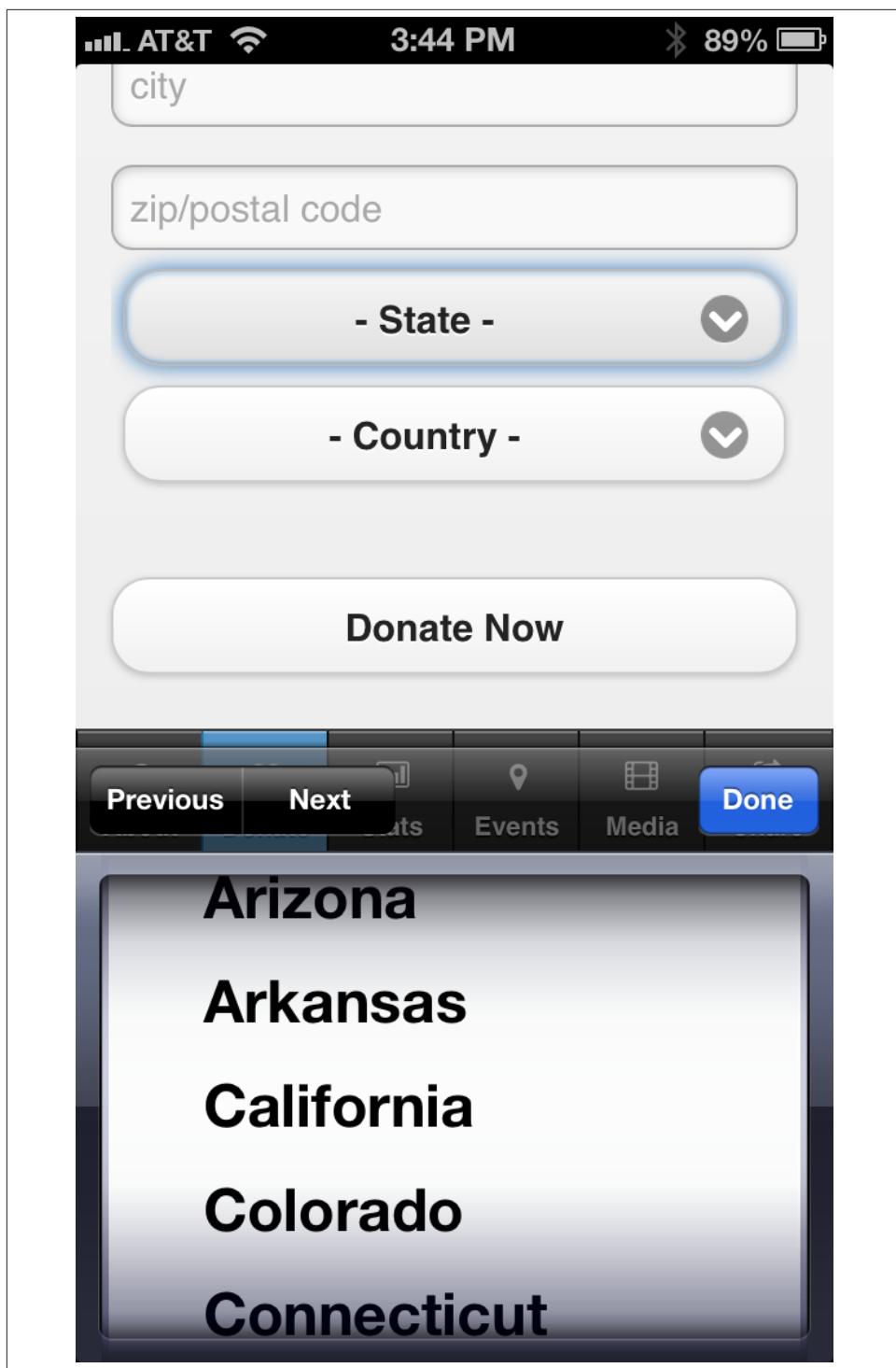


Figure 11-26. States dropdown in the Donate form

The bad news is that sometimes you don't want the default behavior offered by the `<select>` element. For example, you may want to create a menu that shows a list of items. First, we'll show you how to do it using a popup that contains a listview. The next code is taken from the jQuery Mobile documentation - it suggests to implement a listview inside a popup:

```
<a href="#popupMenu" data-rel="popup" data-role="button"
    data-transition="pop">Select Donation Amount</a>

<div data-role="popup" id="popupMenu" >
    <ul data-role="listview" data-inset="true" style="min-width:210px;">
        <li data-role="divider">Choose the amount</li>
        <li><a href="#">$10</a></li>
        <li><a href="#">$20</a></li>
        <li><a href="#">$50</a></li>
        <li><a href="#">$100</a></li>
    </ul>
</div>
```

Initially the screen will look as in Figure 11-27 - it's an anchor styled as a button....

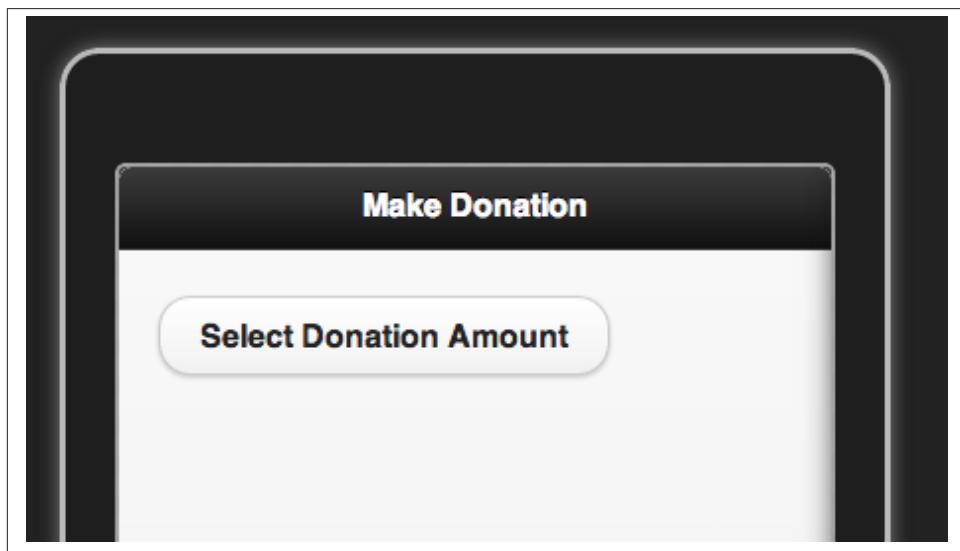


Figure 11-27. Select Donation Amount before the tap

After the user taps on the Set Donation Amount the menu pops up and it'll look as in Figure 11-28.



Figure 11-28. Select Donation Amount after the tap

Another way of creating dropdowns is by using so called **collapsibles**. If the data role of a container is set to be collapsible, the content of the container won't be initially shown. It'll be collapsed showing only its header with a default icon (the plus sign) until the user will tap on it.

```
<div data-role="collapsible" data-theme="b"
      data-content-theme="c">
  <h2>Select Donation Amount</h2>

  <ul data-role="listview">
    <li><a href="#">$10</a></li>
    <li><a href="#">$20</a></li>
    <li><a href="#">$50</a></li>
    <li><a href="#">$100</a></li>
  </ul>
</div>
```

If you'll test the above code in Ripple Emulator, the initial screen will look as on **Figure 11-29** - it's a `<div>` with the `data-role=collapsible`. Note that the this code sample also illustrates using different themes for the collapsed and expanded version of

this <div>. If you are reading the electronic version of this book on a color display, the collapsed version will have the blue background: `data-theme="b"`.

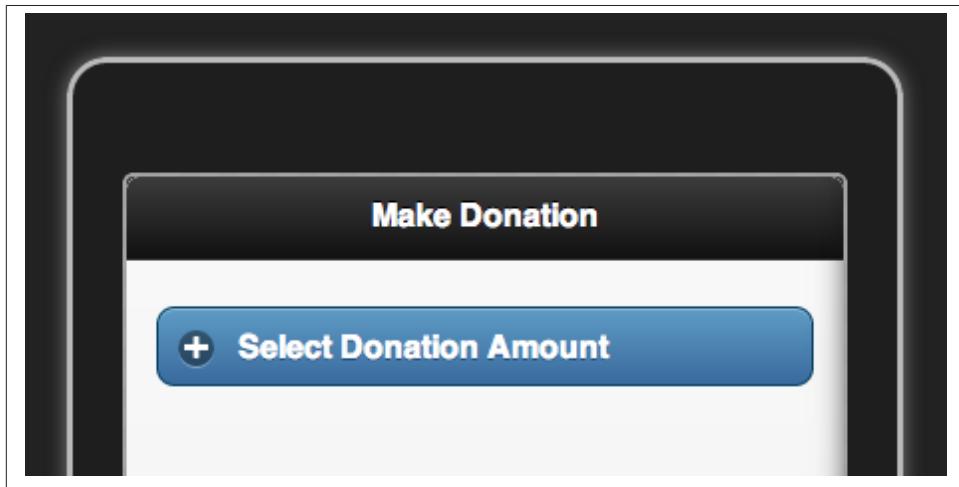


Figure 11-29. Select Donation Amount before the tap

After the user taps on the Set Donation Amount the menu pops up and it'll look as in Figure 11-30. The icon on the header changes from the plus sign to minus.

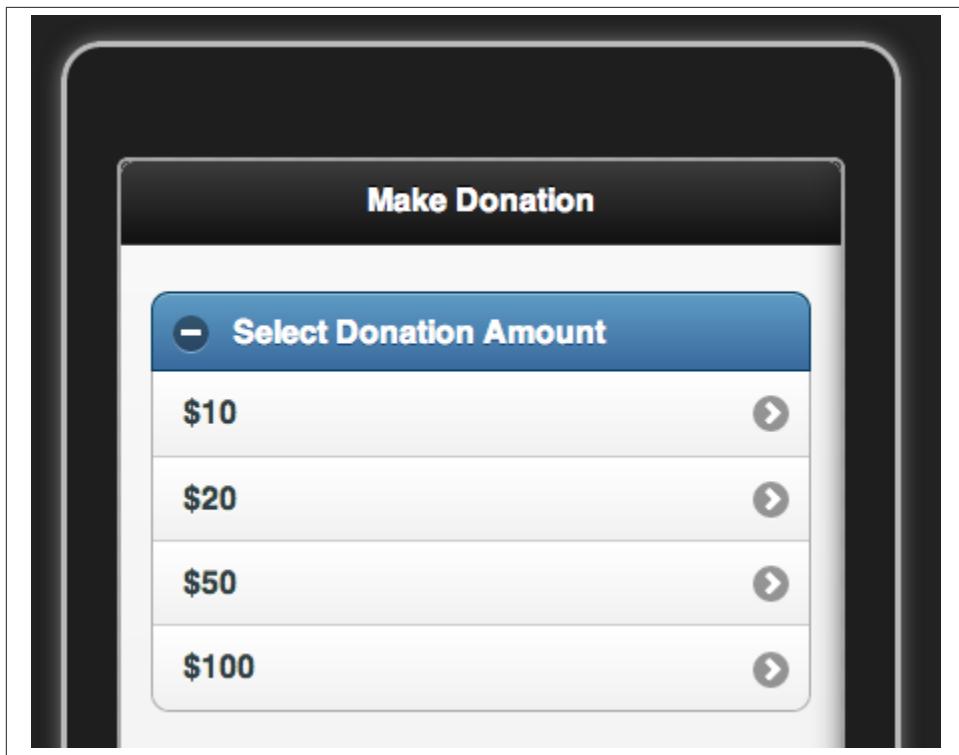


Figure 11-30. Select Donation Amount after the tap

## Listviews

In the section on Collapsibles you saw how easy it was to create a nicely looking list ([Figure 11-30](#)) with `data-role="listview"`. jQuery Mobile offers many ways of arranging items in lists and we encourage you to pay a visit to the [Listviews](#) section in online documentation.

Each list item can contain literally any HTML elements. The media page of the Save The Child application uses `listview` to arrange videos in the list. Below is the code fragment from `media.html`:

```
<div data-role="header"> ... </div>

<div data-role="content" >
  <ul data-role="listview" data-theme="a" data-inset="true" id="video-list">
    <li data-icon="chevron-right">
      <a href="#popupHtmlVideo" data-rel="popup" id="video-1">  <h3>The title of a video-clip</h3>
      <p>
        Video description goes here. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
      </p> </a>
    </li>
  </ul>
</div>
```

```

</li>
<li data-icon="chevron-right">
  <a href="#ytVideo" data-rel="popup">  <h3>The title of a video-clip</h3>
  <p>
    Video description goes here. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  </p> </a>
</li>
</ul>

</div>

<div data-role="footer"> ... </div>

<!-- html5 video in a popup -->
<div data-role="popup" id="popupHtmlVideo" data-transition="slidedown"
data-theme="a" data-position-to="window" data-corners="false">
  <a href="#" data-rel="back" data-role="button" data-theme="a" data-icon="delete" data-icon-class="ui-btn-right">Close</a>
  <video controls="controls" poster="../assets/media/intro.jpg" preload="metadata">
    <source src="../assets/media/intro.mp4" type="video/mp4">
    <source src="../assets/media/intro.webm" type="video/webm">
    <p>Sorry, your browser doesn't support the video element</p>
  </video>
</div>

<!-- YouTube video in a popup -->
<div data-role="popup" id="ytVideo" data-transition="slidedown" data-theme="a"
data-position-to="window" data-corners="false">
  <a href="#" data-rel="back" data-role="button" data-theme="a" data-icon="delete" data-icon-class="ui-btn-right">Close</a>
  <iframe id="ytplayer" src="http://www.youtube.com/embed/VGZcer0hCuo?wmode=transparent&hd=1"
    frameborder="0" width="480" height="270" allowfullscreen></iframe>
</div>
</div>

```

This code uses an unordered HTML list `<ul>`. Each list item `<li>` contains three HTML elements: `<a>`, `<p>`, and `<span>`. The anchor contains a link to the corresponding video to show in a popup. The content of each popup is located in a `<div data-role="popup">`. The `data-rel="popup"` in the anchor means that the resource from `href` has to be opened as a popup when the user taps on this link.

The `<div id="popupHtmlVideo">` illustrates how to include a video using HTML5 tag `<video>`, and `<div id="ytVideo">` shows how to embed a Youtube video. Note that both of these `<div>` elements are placed below the footer, and jQuery Mobile won't show them until the user taps on the links.

Note that jQuery Mobile `listview` is styled in a way that each list item looks like a large rectangle, and the user can tap on the list item with his finger without being afraid of

touching the neighbor controls. There is no such problem with desktop applications because the mouse pointer has a lot better precision than a finger or even a stylus.

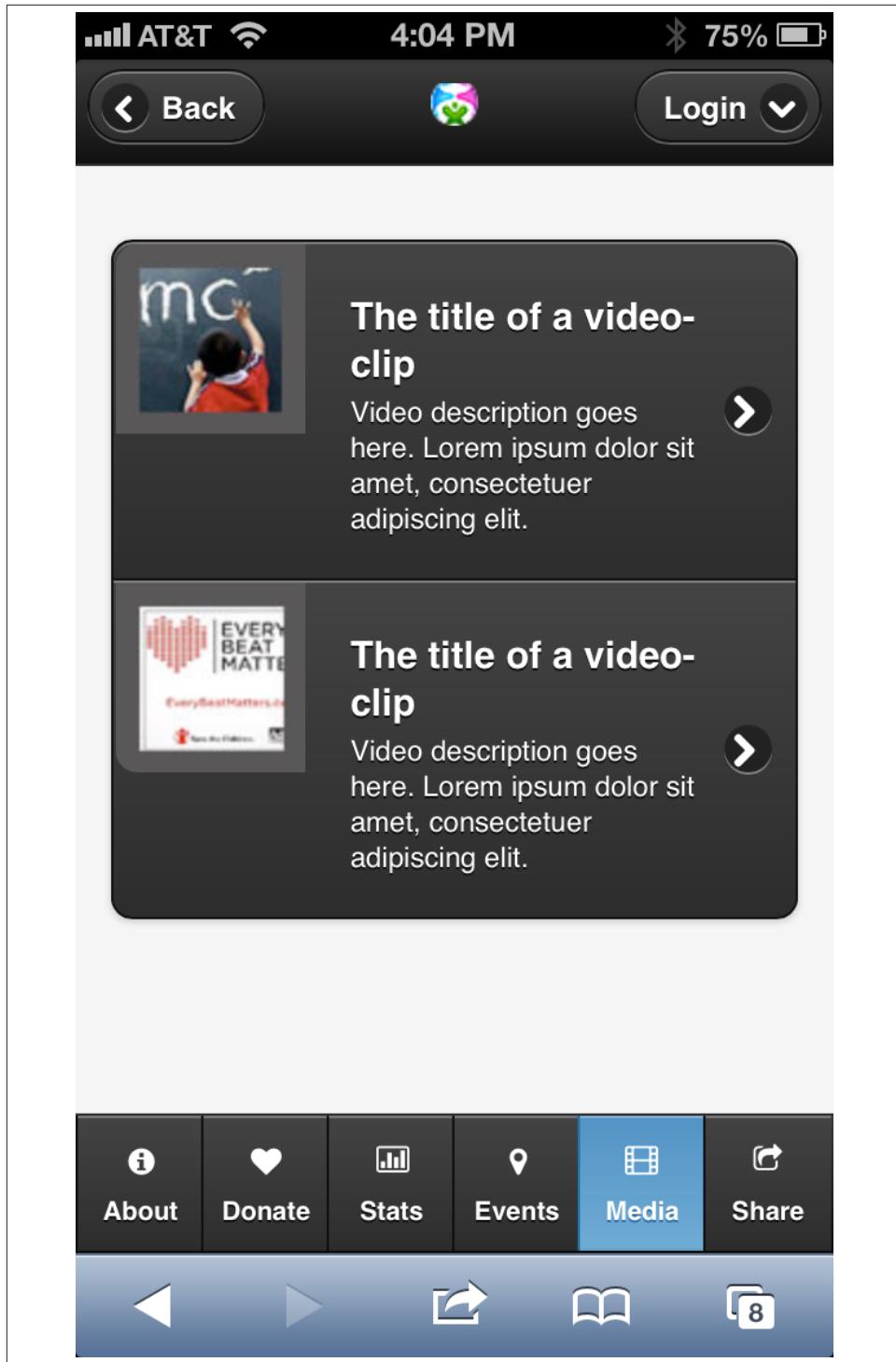


Figure Chapter 10: jQuery Mobile in media.html



The <video> tag has an attribute `autoplay`. But since some of the mobile users are being charged by their phone companies based on their data usage, you may not automatically start playing video until the user explicitly taps the button play. There is no such restrictions in the desktop browsers.

## jQuery Mobile Events

jQuery Mobile Events can be grouped by their use. There are events that deal with the page life cycle. For detailed description of events read the [Events section](#) in the online documentation. We'll just briefly mention some of the events available in jQurMobile.

You should be using `$(document).on("pageinit")` and not `$(document).ready()` because the former is triggered even for the pages loaded as result of AJAX calls while the latter won't. Prior to `pageinit` two more events are being dispatched: `pagebeforecreate` and `pagecreate` - after these two the widget enhancement takes place.

The `pagebeforeshow` and `pageshow` events are happening right before or after the to-page is displayed. Accordingly, `pagebeforehide` and `pagehide` are dispatched on the from-page. The `pagechange` event is dispatched when the page is being changed as the result of the programmatic invocation of the `changePage()` method.

If you are loading an external page (e.g. a user clicked on a link `<a href="external page.html">Load External</a>`), expect two events: `pagebeforeload` and `pageload` (or `pageloadfailed`).

Touch events is another group of events that are dispatched when the user touches the screen. Depending on how the user touches the screen, your application may receive `tap`, `taphold`, `swipe`, `swipeleft`, and `swiperight` events. The tap event handlers may or may not work reliably on iOS devices.

The `touchend` event may be more reliable. Create a combined event handler for `click` and `touchend` events and your code will work on both desktop and mobile devices, for example:

```
$('#radio-container .ui-radio').on('touchend click', function() {  
    // the event handler code goes here  
})
```

Orientation events are important if your code needs to intercept the moments when the mobile device changes orientation. This is when jQuery Mobile fires the `orientationchange` event. The event object will have a property `orientation`, which will have either `portrait` or `landscape` in it.

There is one event that you can use to set some configuration options for the jQuery Mobile itself. The name of this event is `mobileinit`, and you should call the script to

apply overrides after the jQuery Core, but before jQuery Mobile scripts are loaded.  
Details in [online documentation](#).

## Adding JavaScript

So far we were able to get by with HTML and CSS only - jQuery Mobile library was doing its magic, which was very helpful for the most part. But we still need a place for Javascript - Save The Child application has several hundreds of lines of JavaScript code and we need to find it a new home. You'll find pretty much the same code that we used in previous chapters to deal with login, donate, maps and stats. It's located in the *jquerymobile* sample project in the file js/app-main.js.

You may also need to write some scripts specific to jQuery Mobile workflows because, in some cases, you may want to override certain behavior of this library. In such cases you'd need to write JavaScript functions to serve as event handlers. For example, jQuery Mobile has a restriction that you can put not more than five buttons on the `navbar`. But we need six. Just to remind you, the footer contains an attribute `data-role="navbar"` and it has an unordered list `ul` with six `<li>` items (not shown below for brevity):

```
<div data-role="footer" data-position="fixed" data-tap-toggle="false"
      data-id="persistent-footer">
  <div data-role="navbar" class="ssc-navbar">
    <ul>
      ...
    </ul>
  </div>
</div><
```

Run the application with six buttons in the `navbar`, and get ready for the surprise. You'll see a footer with a two-column and three-row grid as shown in [Figure 11-32](#), which is a screen snapshot of a Ripple Emulator with open Chrome Developer Tools panel while inspecting the `navbar` element in the footer.

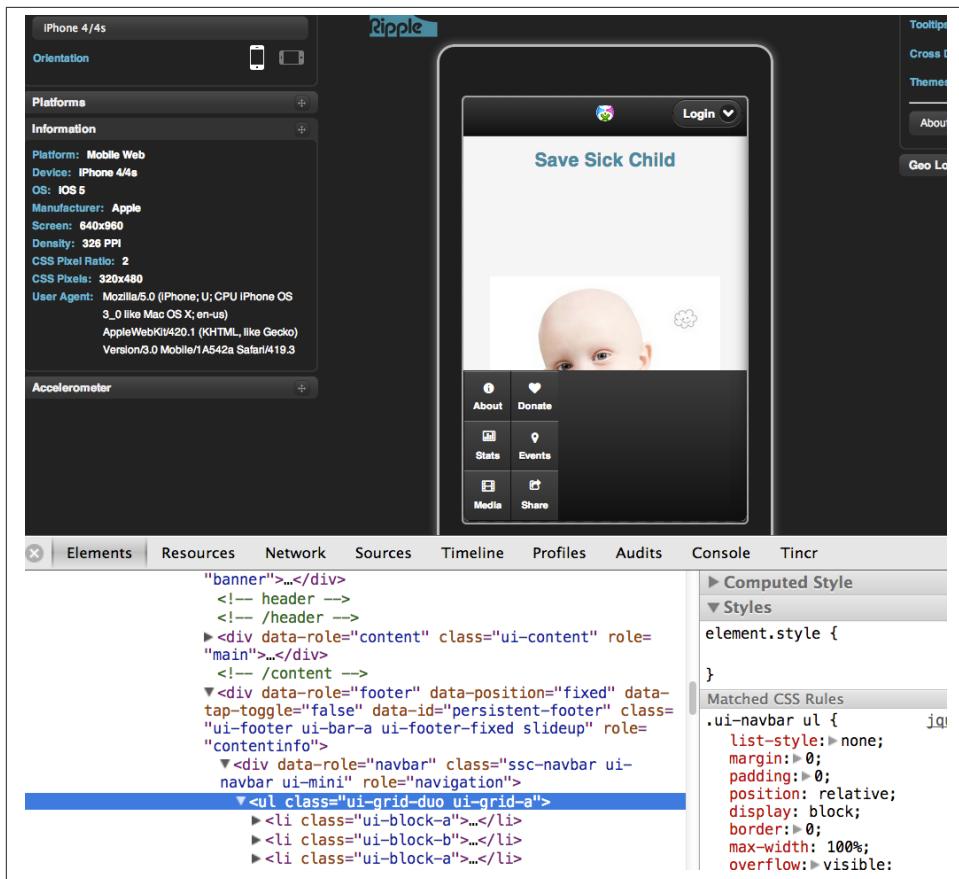


Figure 11-32. Using listview in media.html

Take a look at the styling of the navbar. Our original `<ul>` HTML element didn't include the class `ui-grid-a`. jQuery Mobile couldn't find the predefined layout for a six-button navigational bar and "decided" to allocate it as `ui-grid-a`, which is a two column grid (see the section Grid Layouts above).

The CSS file `app-styles.css` (see section The Project Structure and Navigation) has the provision for giving 16.6% of the width for each of six buttons, but we need to programmatically remove that `ui-grid-a`, which jQuery Mobile injected into our code. We'll do it in JavaScript in the handler for `pagebeforeshow` event. The next code snippet from `app-main.js` finds the `ul` element that includes `ssc-navbar` as one of the styles and removes the class `ui-grid-a` from this unordered list:

```
$(document).on('pagebeforeshow', function() {
  $(".ssc-navbar > ul").removeClass("ui-grid-a");
```

Now the 16.6% of width will take effect and properly allocate all six buttons in a row. This was an example of overriding unwanted behavior using JavaScript. The rest of the code contains familiar functionality from the previous sections. We won't repeat it here, but will show you some of the code sections that are worth commenting.

```
$(document).on('pagebeforeshow', function() {  
    $(".ssc-navbar > ul").removeClass("ui-grid-a");  
  
    if ( typeof(Storage) != "undefined" ) {  
        var loginVal = localStorage.sscLogin; // ①  
  
        if (loginVal == "logged") {  
            $('.login-btn').css('display', 'none');  
            $('.logout-btn').css('display', 'block');  
        } else {  
            $('.login-btn').css('display', 'block');  
        }  
    } else {  
        console.log('No web storage support...');  
    }  
});  
  
function logIn(event) {  
    event.preventDefault();  
  
    var userNameValue = $('#username').val();  
    var userNameValueLength = userNameValue.length;  
    var userPasswordValue = $('#password').val();  
    var userPasswordLength = userPasswordValue.length;  
  
    //check credential  
    if (userNameValueLength == 0 || userPasswordLength == 0) {  
        if (userNameValueLength == 0) {  
            $('#error-message').text('Username is empty');  
        }  
        if (userPasswordLength == 0) {  
            $('#error-message').text('Password is empty');  
        }  
        if (userNameValueLength == 0 && userPasswordLength == 0) {  
            $('#error-message').text('Username and Password are empty');  
        }  
        $('#login-submit').parent().removeClass('ui-btn-active');  
        $('[type="submit"]').button('refresh');  
    } else if (userNameValue != 'admin' || userPasswordValue != '1234') {  
        $('#error-message').text('Username or password is invalid');  
    } else if (userNameValue == 'admin' && userPasswordValue == '1234') {  
        $('.login-btn').css('display', 'none');  
        $('.logout-btn').css('display', 'block');  
    }  
    localStorage.sscLogin = "logged"; // ②  
    history.back();  
}
```

```

        }
    }

    $('#login-submit').on('click', logIn);

    ...

    $(document).on('pageshow', "#Donate", function() { // ③
        ...
    }

    $(document).on("pageshow", "#Stats", function() { // ④
        ...
    }

    $(document).on("pageshow", "#Events", function() { // ⑤
        ...
    }
}

```

- ① The button Login is located on the header of each page, and it turns into the button Logout when the user logs in. When the user moves from page to page, the old pages are being removed from DOM. To make sure that the login status is properly set, we check if the variable `sscLogin` in the local storage has the value `logged` (see explanation below).
- ② When the user logs in, the program saves the word `logged` in the local storage and closes Login popup by calling `history.back()`.
- ③ The Donate form code is located in this function. No AJAX calls are being made in this version of the Save The Child application.
- ④ The SVG charts are created in this function.
- ⑤ The GeoLocation code that uses Google Maps API goes here

While experimenting with Save The Child application we've created one more version using the multi-page template just to get a feeling of how smooth transitioning between the pages will look like if the entire code base will be loaded upfront. Of course, the wait cursor between the pages was gone, but the code itself became less manageable.



Ripple Emulator described earlier in this chapter allows you to test the look and feel of the jQuery Mobile version of the Save The Child application on various iOS and Android devices. But again, nothing beats testing on real devices.

## Summary

In this chapter you've got familiar with a simple to use mobile framework. We've been using its version 1.3.1, which works pretty stable, but it's not a mature library just yet. You can still run into situations when a feature advertised in the product documentation doesn't work (e.g. [page prefetching breaks images](#)). So be prepared to study the code of this library and do the fixes to the critical features on your own. But there is a group of people who are actively working on bug fixing and improving jQuery Mobile, and using it in production is pretty safe.

By now you should have a pretty good understanding of how to start creating user interface with jQuery Mobile and where to find more information. Find some time and read the entire online documentation on jQuery Mobile. The learning curve is not steep, but there is a lot to read if you want to become productive with jQuery Mobile.

---

## CHAPTER 12

# Sencha Touch

Sencha Touch framework is a little brother of Ext JS. They both have the same creator: [Sencha Inc](#), and they both are built on the same set of core classes. But Sencha Touch is created for developing mobile Web, while Ext JS is for desktop Web applications.

Enterprise IT managers need to be aware of another important difference: Ext JS offers free licenses only for open source projects, but [Sencha Touch licenses](#) are free unless you decide to purchase this framework bundled with developers tools.

This chapter is structured similarly to the previous one that described jQuery Mobile - minimum theory followed by the code. The fundamental difference though is that if Chapter 11 has almost no JavaScript, while this chapter will have almost no HTML.

We'll try minimizing repeating the information you can find in [Sencha Touch Learning Center](#) and extensive product documentation, which has multiple well written [Guides](#) on various topics. This chapter will start with a brief overview of the features of Sencha Touch followed by the code review of yet another version of the Save The Child application. In this chapter we are going to use Sencha Touch 2.3.1, which is the latest version at the time of this writing. It supports iOS, Android, Blackberry, and Window Phone.



If you haven't read Chapter 4 on Ext JS, please do it now. Both of these frameworks are built on the same foundation and we assume that you are familiar with such concepts as MVC architecture and things like xType, SASS and some others explained in Chapter 4. For the most part Ext JS and Sencha Touch non-UI classes are compatible, but there are some differences that may prevent you from 100% code reuse between these frameworks(e.g. see section "Stores and Models"). The future releases Sencha should come up with some standard solutions to remove the differences in class systems of both frameworks.

# Sencha Touch Overview

Let's start with downloading Sencha Touch from <http://www.sencha.com/products/touch/download>. If you want to get a free commercial license just specify your email address, and you'll receive the link to download in the email. Sencha Touch framework comes as a zip file, which you should unzip in any directory - later you'll copy the framework's code either into your project directory or in the document root of your Web server.



Commercial license of Sencha Touch doesn't include charts (you'd need to get either Sencha Complete or Sencha Touch Bundle for the chart support). Because of this we'll be using the General Public License (GPL) of Sencha Touch for the open source Save The Child project, and our users will see a little watermark "Powered by Sencha Touch GPLv3" as shown on Figure [Figure 12-1](#) below.

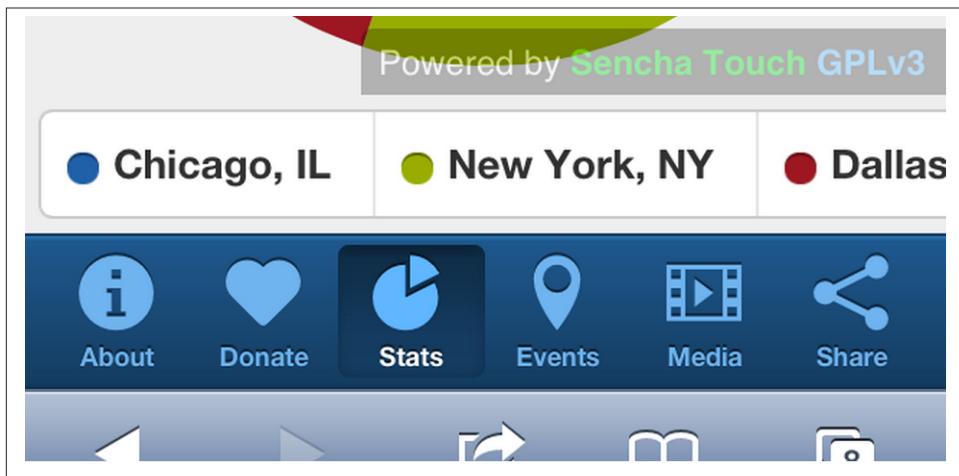


Figure 12-1. A GPL license watermark

After downloading Sencha Touch we've unzipped it into the directory `/Library/touch-2.3.1`, but the code generation process will copy this framework into our application directory.

## Code Generation and Distribution

If you haven't downloaded and installed the Sencha CMD tool, do it now as described in Chapter 4. This time we'll use Sencha CMD to generate a mobile version of Hello World. After opening a Terminal or Command Window enter the following command

specifying the absolute path to your ExtJS SDK directory and to the output folder, where the generated project should reside.

```
sencha -sdk /Library/touch-2.3.1 generate app HelloWorld /Users/yfain11/hellotouch
```

After the code generation is complete, you'll see the folder *hello* of the structure shown on Figure [Figure 12-2](#). It follows the MVC pattern discussed in the Ext JS chapter.

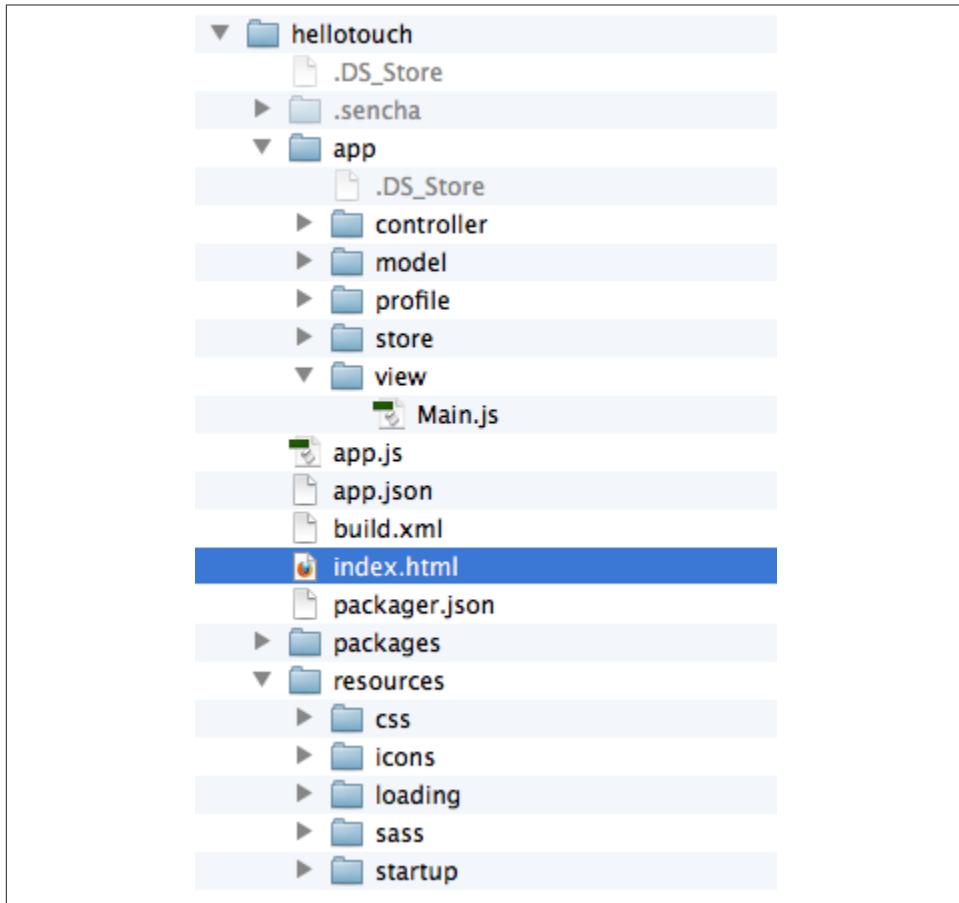


Figure 12-2. CMD-generated project

To test your newly generated application make sure that the directory *hellotouch* is deployed under a Web server (simple opening of *index.html* in the Web browser won't work). You can either install any Web server or just follow the instructions from section about XAMPP and Apache Web Server in Chapter 4. In the same chapter you can find the command to start the Jetty Web server embedded in the Sencha CMD tool.

Here we are going to use the internal Web server that comes with WebStorm IDE. It runs on the port 63342, and if your project's name is helloworld, the URL to test it is <http://localhost:63342/helloworld>.



To debug your code inside WebStorm IDE, select the menu Run | Edit Configurations, press the plus sign in the top left corner and in JavaScript Debug | Remote panel enter the URL <http://localhost:63342> followed by the name of your project(e.g. ssctouch) and name your new debug configuration. After that you'll be able to debug your code in your Chrome Web browser (it'll ask you to install the JetBrains IDE Support extension of the first run).



MAC OS X users can install a small application [Anvil](#) that can easily serve static content of any directory as a Web server with a URL that ends with .dev.

Figure [Figure 12-3](#) shows how the generated Hello World application will look in Chrome browser. It'll consist of two pages controlled by the buttons in the footer toolbar.

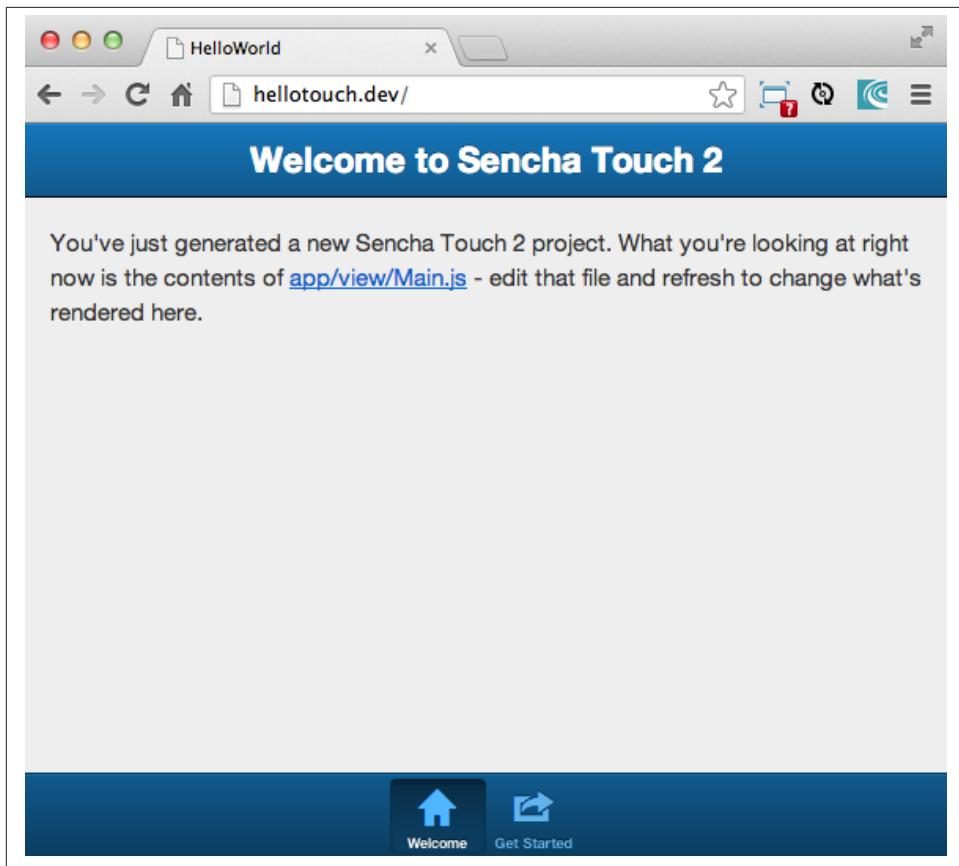


Figure 12-3. Running CMD-generated Hello World

### Microloader and Configurations

The main application entry is the JavaScript file app.js. But if in Ext JS, this file was directly referenced in index.html, Sencha Touch applications generated by CMD tool use a separate microloader script, which starts with loading the file app.json that contains the names of the resources needed for your application including the app.js. The only script included in the generated index.html is this one:

```
<script id="microloader" type="text/javascript"
       src="touch/microloader/development.js"></script>
```

This script uses one of the scripts located in the microloader folder, which gets the object names to be loaded from the configuration file app.json. This file contains a JSON object with various attributes like js, css, resources and others. So if your application needs to load the scripts sencha-touch.js and app.js, they should be located in the js array.

Here's what the `js` attribute of the `app.json` contains after the initial code generation by Sencha CMD:

```
"js": [
  {
    "path": "touch/sencha-touch.js",
    "x-bootstrap": true
  },
  {
    "path": "app.js",
    "bundle": true,
    "update": "delta"
  }
]
```

Eventually, if you'll need to load additional JavaScript code, CSS files or other resources add them to the appropriate attribute in the file `app.json`.

Introducing a separate configuration file and additional microloader script may seem like an unnecessary complication, but it's not. On the contrary, it gives you the flexibility of maintaining clean separation between development, testing, and production environments. You can find three different loader scripts in the folder `touch/microloader`: `development.js`, `production.js`, and `testing.js`. Each of them can load different configuration file.



Our sample application includes some sample video files. Don't forget to include "resources/media" folder in the `resources` section of the `app.json`.

If you open the source code of the production loader, you'll see that it uses application cache to save files locally on the device (see section Application Cache in Appendix B for a refresher), so the user can start the application even without having the Internet connection.

The production microloader of Sencha Touch offers a smarter solution for minimizing unnecessary loading of cached JavaScript and CSS files than HTML5 Application Cache. The standard HTML5 mechanism doesn't know which resources have changed and reloads all cacheable files. CMD-generated production builds for Sencha Touch keep track of changes and create deltas, so the mobile device will download only those resources that have been actually changed. To create a production build, open a Terminal or a command window, change to your application directory and run the following command:

```
sencha app build production
```

See the section “[Deploying Your Application](#)” for more details on Sencha CMD builds. When we start building our Save The Child application, you’ll see how to prompt the user that the application code has been updated. Refer to the [online documentation](#) on using Sencha CMD with Sencha Touch for details.

## Code Distribution and Modularization

The ability of Sencha Touch to monitor modified pieces of code helps with deployment - just change the SomeFile.js on the server and it’ll be automatically downloaded and saved on the user’s mobile device. This may have some effect on the application modularization decisions you will take.

Reducing the startup latency and implementing lazy loading of certain parts of the application are the main reasons for modularizing Web applications. The other reason for modularization is an ability to redeploy certain portions of the code vs. the entire application if the code modifications are limited in scope.

So should we load the entire code base from the local storage (it’s a lot faster than getting the code from remote servers) or still use loaders to bring up the portion of the code (a.k.a. modules) on as needed basis? There is no general answer to this question - every application is different.

If your application is not too large and the mobile device has enough memory, loading the entire code of the application from the local storage may lower the need for modularization. For larger applications consider the [Workspaces](#) feature of Sencha CMD, which allows to create some common code to be shared by several scripts.

## The Code of Hello World

Similarly to Ext JS, the starting point of the Hello World application is the app.js script.

```
Ext.Loader.setPath({
    'Ext': 'touch/src',           // ①
    'HelloWorld': 'app'
});

Ext.application({
    name: 'HelloWorld',

    requires: [
        'Ext.MessageBox'
    ],

    views: [
        'Main'
    ],
    icon: {
```

```

    '57': 'resources/icons/Icon.png',
    '72': 'resources/icons/Icon~ipad.png',
    '114': 'resources/icons/Icon@2x.png',
    '144': 'resources/icons/Icon~ipad@2x.png'
},
isIconPrecomposed: true,

startupImage: {
    '320x460': 'resources/startup/320x460.jpg',
    '640x920': 'resources/startup/640x920.png',
    '768x1004': 'resources/startup/768x1004.png',
    '748x1024': 'resources/startup/748x1024.png',
    '1536x2008': 'resources/startup/1536x2008.png',
    '1496x2048': 'resources/startup/1496x2048.png'
},
launch: function() {
    // Destroy the #appLoadingIndicator element
    Ext.fly('appLoadingIndicator').destroy();

    // Initialize the main view
    Ext.Viewport.add(Ext.create('HelloWorld.view.Main'));
},
onUpdated: function() { // ②
    Ext.Msg.confirm(
        "Application Update",
        "This application has just successfully
        been updated to the latest version. Reload now?",
        function(buttonId) {
            if (buttonId === 'yes') {
                window.location.reload();
            }
        }
    );
}
});

```

- ① This code instructs the loader that any class that starts with *Ext* can be found in the directory *touch/src* or its subdirectories. The classes with names that starts with *HelloWorld* are under the *app* directory.
- ② This is an interception of the event that's triggered if the code on the server was updated. The user is warned that the new version of the application has been downloaded. See more on this in the comments to *app.js* in the section Save The Child With Sencha Touch.

The code of the generated main view of this application (*Main.js*) is shown next. It extends the class *Ext.tab.Panel* so each page of the application is one tab in this panel.

Figure [Figure 12-4](#) is a snapshot of a collapsed version of Main.js taken from [WebStorm IDE](#) from JetBrains, which is our IDE of choice in this chapter.

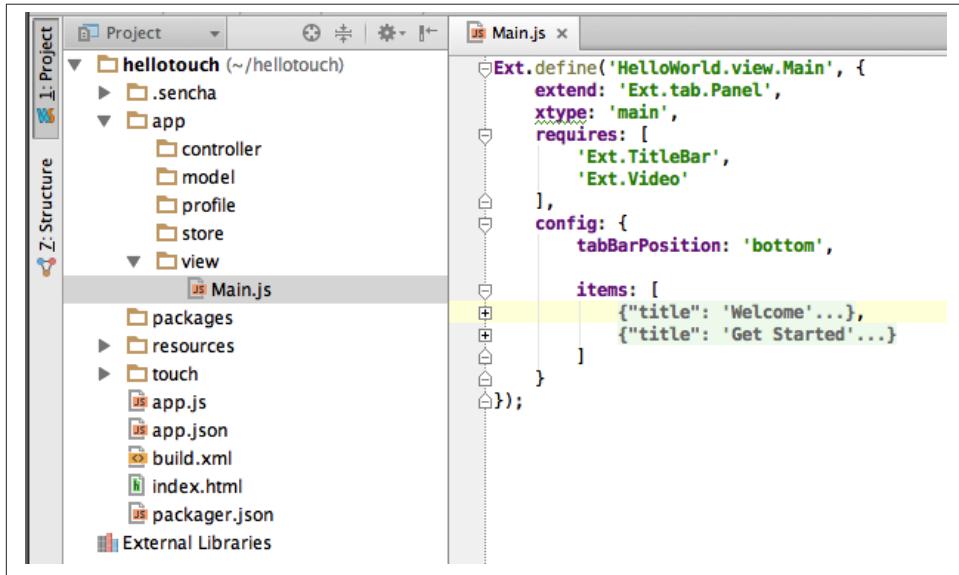


Figure 12-4. Collapsed version of Main.js from Hello World

As you see from this figure the `items[]` array includes two objects: Welcome and Get Started - each of them represents a tab (screen) on the panel.

```
Ext.define('HelloWorld.view.Main', {
    extend: 'Ext.tab.Panel',
    xtype: 'main',
    requires: [
        'Ext.TitleBar',
        'Ext.Video'
    ],
    config: {
        tabBarPosition: 'bottom', // ①
        items: [
            { // ②
                title: 'Welcome',
                iconCls: 'home',
                styleHtmlContent: true,
                scrollable: true,
                items: {
                    docked: 'top',
                    xtype: 'titlebar'
                }
            },
            { // ③
                title: 'Get Started',
                iconCls: 'star',
                styleHtmlContent: true,
                scrollable: true,
                items: {
                    docked: 'top',
                    xtype: 'titlebar'
                }
            }
        ]
});
```

```

        title: 'Welcome to Sencha Touch 2'
    },

    html: [
        "You've just generated a new Sencha Touch 2 project. What you're looking at right
        "contents of <a target='_blank' href=\"app/view/Main.js\">app/view/Main.js</a> - e
        "and refresh to change what's rendered here."
    ].join("")
},
{
    title: 'Get Started',
    iconCls: 'action',

    items: [
        {
            docked: 'top',
            xtype: 'titlebar',
            title: 'Getting Started'
        },
        {
            xtype: 'video',
            url: 'http://av.vimeo.com/64284/137/87347327.mp4?token=
            1330978144_f9b698fea38cd408d52a2
            393240c896c',
            posterUrl: 'http://b.vimeocdn.com/ts/261/062/261062119_640.jpg'
        }
    ]
},
]
);

```

- ❶ The tab bar has to be located at the bottom of the screen.
- ❷ The first tab is a Welcome screen.
- ❸ The second tab is the Getting Started screen. It has `xtype: video`, which means it's ready for playing video located at the specified `url`.

This application has no controllers, models or stores. But it does include the default theme from SASS stylesheet resources/sass/app.scss, which was compiled by Sencha CMD generation process into the file resources/css/app.css.

## Constructing UI

Sencha Touch has a number UI components specifically designed for mobile devices, which include lists, forms, toolbars, buttons, charts, audio, video, carousel and more. The quickest way to get familiar with UI components is by browsing the **Kitchen Sink** Web site, where you can find the examples of how UI components look and see the source code of these examples.

## Containers

In general, the process of implementing of a mobile application with Sencha Touch will consist of selecting the appropriate containers and arranging the navigation between them. Each screen that user sees is a container. Pretty often it'll include a toolbar *docked* on top or bottom of the container.

Containers can be nested - they are needed for better grouping of UI components on the screen. The lightest container is `Ext.Container`. It inherits all the functionality from its ancestor `Ext.Component` plus it can contain other components. When you'll be reviewing the code of the Save The Child application, note that the main view `SSC.view.Main` from `Main.js` extends `Ext.Container`. The hierarchy of Sencha Touch containers is shown on Figure [Figure 12-5](#).

<p>ALTERNATE NAMES</p> <p>Ext.lib.Container</p>
<p>HIERARCHY</p> <pre>Ext.Base   Ext.Evented     Ext.AbstractComponent       Ext.Component         Ext.Container</pre>
<p>INHERITED MIXINS</p> <p>Ext.mixin.Observable Ext.mixin.Traversable</p>
<p>REQUIRES</p> <p>Ext.ItemCollection Ext.Mask Ext.behavior.Scrollable Ext.layout.*</p>
<p>SUBCLASSES</p> <p>Ext.Map Ext.Panel Ext.SegmentedButton Ext.TitleBar Ext.Toolbar Ext.carousel.Carousel Ext.dataview.DataView Ext.dataview.NestedList Ext.dataview.component.Container Ext.dataview.component.DataItem Ext.draw.Component Ext.form.FieldSet Ext.navigation.View Ext.slider.Slider Ext.tab.Panel Ext.viewport.Default</p>

Figure 12-5. Sencha Touch Containers Hierarchy

The `FieldSet` is also a pretty light container - it simply adds the title to a group of fields that belong together. You'll see several code samples in this chapter with `xtype: 'fieldset'` (e.g. Login or Donate screens).

If your containers display forms with such inputs as text field, text area, password, and numbers, the virtual keyboard will automatically show up occupying half of the user's screen. On some platforms, virtual keyboards will adapt to the type of the input field, for example, if the field has `xtype: 'emailfield'`, the keyboard will be modified for easier input of emails. Figure [Figure 12-6](#) is a snapshot taken from the Donate screen of the Save The Child application when the user tapped inside the email field - note the key with the at-sign on the main keyboard, which wouldn't be shown for non-email inputs.

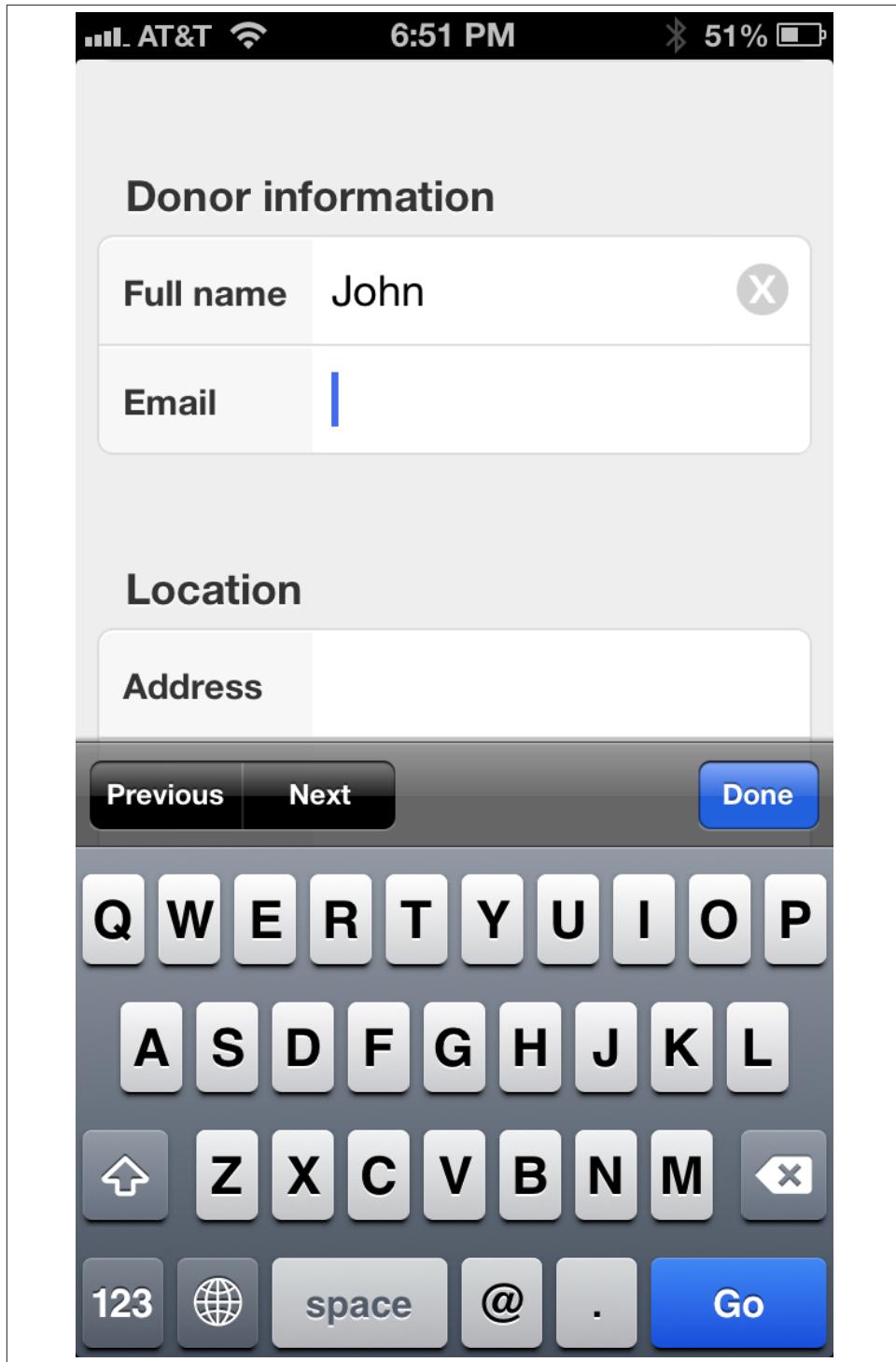


Figure 10-10 The iPhone's virtual keyboard for entering emails

If the field is for entering a URL (`xtype: 'urlfield'`) expect to see a virtual keyboard with the button labeled as “.com”. If the input field has `xtype: 'numberfield'` the user may see a numeric keyboard when the focus gets into this field.



If you need to detect the environment on the user’s mobile device, use such classes as `Ext.os`. for detecting the Operating System, `Ext.browser` for browser, and `Ext.feature` for supported features.

## Layouts

Besides grouping components, containers allow you to assign a `Layout` to control its children arrangements. In desktop applications physical screens are larger, and pretty often you can place multiple containers on the same screen at the same time. In mobile world you don’t have such a luxury and typically you’ll be showing just one container at a time. Not all `layouts` are practical to use on smaller screens, which is the reason why not all Ext JS layouts are supported in Sencha Touch.

Figure [Figure 12-10](#) illustrates the main container that shows either the `tabpanel` or `loginform`. The `tabpanel` is a container with a special layout that shows only one of its child containers at a time (e.g. About, Donate, et al). You can see all these components in action at [savesickchild.org](http://savesickchild.org) - just run the Sencha Touch version of our Save Sick Child application and view the sources.

By default, a container’s layout is `auto`, which instructs the rendering engine to use the entire width of the container, but use just enough height to display the children. This behavior is similar to the `vbox` layout (vertical box), where all components are being added to the container vertically - one under another. Accordingly, the `hbox` will arrange all components horizontally - one next to the other.



If you want to control how much of a vertical or horizontal screen space is given to each component use the `flex` property as described in Chapter 4 in the section “The flex Property”.

The `fit` layout will fill the entire container’s space with its child element. If you have more than one child element in the container - the first one will fill the entire space, and the other one will be ignored.

The `card` layout can accommodate multiple children while displaying only one at a time. The container’s method  `setActiveItem()` allows programmatically select the “card” to be on top of the deck. With card layout all containers are being preloaded to the device,

but if you want to create new containers during the runtime, you can use the method  `setActiveItem()` passing a `config` object describing the new container.

You can find examples of `card` and `fit` layouts in the code of Main.js of the Save The Child application. Figure [Figure 12-11](#) shows `card` layout, but if you'll expand the `tabpanel` container, each tab has the `fit` layout.

The classes `TabPanel` and `Carousel` represent two different implementations of the containers with `card` layout.

## Events

Events can be initiated either by the browser or by the user. Chapter 4 has the section with the same title - it covers general rules of dealing with events in Ext JS framework. Lots of system events are being dispatched during UI component rendering. The online documentation lists every event that can be dispatched on Sencha classes. Look for the Events section on the top toolbar in the online documentation. Figure [Figure 12-7](#) is a snapshot from online documentation for the class `Ext.Container`, which has 32 events.

The screenshot shows the Sencha Touch online documentation interface. At the top, there's a navigation bar with tabs: Components, Animations, Workspace, Logger, Using Sencha, Using Native, and Views. Below the navigation bar, the title "Ext.Container" is displayed with a gear icon, and the subtitle " xtype: container". There are tabs for Configs (56), Properties (0), Methods (170), and Events (32). A "Filter class members" input field is also present. The main content area contains a summary of what a Container is, followed by a list of events. On the right side, there's a sidebar with a list of event names.

A Container has all of the abilities of [Component](#) and lets you nest other Components inside it. Applications are made up of lots of components, usually nested inside one another. Containers give you the ability to render and arrange child Components within them. Most apps have a single top-level Container, called a Viewport, which takes up the entire screen. Inside of this are child components, for example in a mail app the Viewport Container's two children might be a message List and an email preview panel.

Containers give the following extra functionality:

- Adding child Components at instantiation or run time
- Removing child Components
- Specifying a [Layout](#)

**Events**

activate	5	maxheightchange
activeitemchange	6	maxwidthchange
add	4	minheightchange
bottomchange		minwidthchange
centeredchange		move
deactivate	1	painted
destroy		remove
disabledchange		resize
dockedchange		rightchange
erased		scrollablechange
flexchange		show
floatingchange		topchange
fullscreen		updatedata
heightchange		widthchange
hiddenchange		
hide		
initialize		
leftchange		

Figure 12-7. Events in Online documentation

Sencha Touch knows how to handle various mobile-specific events. Check out the documentation for the class [Ext.dom.Element](#) - you'll find there such events as `touchstart`, `touchend`, `tap`, `doubletap`, `swipe`, `pinch`, `longpress`, `rotate`, and others.

You can add event listeners using different techniques. One of them is defining the `listeners` config property during the object instantiation. This property is declared in the `Ext.Container` object and allows you to define more than one listener at a time. You should use it while calling the `Ext.create()` method:

```
Ext.create('Ext.button.Button', {
    listeners: {
        tap: function() { // handle event here }
    }
})
```

If you need to handle an event only once, you can use the option `single: true`, which will automatically remove the listener after the first handling of the event. For example:

```
listeners: {
    tap: function() { // handle event here },
    single: true
}
```



Read the comments to the code of `SSC.view.CampaignsMap` in Chapter 4 about the right place for declaring listeners.

You can also define event handlers using yet another `config` property control from `Ext.Container`. For example the following code fragment from the Login controller of the Save The Child application shows how to assign the `tap` event handler functions `showLoginView()` and `cancelLogin()` for the buttons Login and Cancel.

```
Ext.define('SSC.controller.Login', {
    extend: 'Ext.app.Controller',
    config: {
        control: {
            loginButton: {
                tap: 'showLoginView'
            },
            cancelButton: {
                tap: 'cancelLogin'
            }
        }
    },
    showLoginView: function () {
        // code of this function is removed for brevity
    },
    cancelLogin: function () {
        // code of this function is removed for brevity
    }
});
```



With the proliferation of the touch screens Sencha has introduced the `tap` gesture, which is semantically equivalent to `click` event.

Read more about the role of controllers in event handling in the section titled Controller later in this chapter. Online documentation includes the [Event Guide](#) - it describes the process of handling events in detail.



If you want to fire custom events, use the method `fireEvent()`, providing the name of your event. The procedure for defining the listeners for custom events remains the same.



The Bring Your Own Device is getting more and more popular in enterprises. Sencha offers a product called Sencha Space, which is a secure and managed environment to deploy enterprise HTML5 application that can be run on a variety of devices that employees can bring to the workplace. Sencha Space promises a clear separation between work-related applications and personal data. It uses secure database and secure File API and allows App-to-App communication. For more details visit the [Sencha Space Web page](#).

## Save The Child With Sencha Touch

The Sencha Touch version of the Save The Child application will be based on the mockup from Chapter 11, section “Prototyping Mobile Version” with some minor changes. This time the home page of the application will be a slightly different version of the About page shown on [Figure 12-8](#).

### Building the Application

The materials presented in this chapter were tested with Sencha Touch 2.3.1 framework, which was current at the time of this writing, and you can use the source code of the Save The Child application that comes with the book. It's packaged with Sencha 2.3.1. We've also deployed this application at <http://savesickchild.org:8080/ssc-touch-prod>.

In case you need to use a newer version of Sencha Touch framework, just download and unzip it to the directory of your choice (in our case it was `/Library/touch-2.3.1`). Download the book code and remove the content of the `touch` directory from `Lesson12/ssc-mobile`. After that, `cd` to this directory and copy a newer version of Sencha touch there. For example, on MAC OS we did it as follows:

```
cd ssc-mobile cp -r /Library/touch-2.3.1/ touch
```

Then run the Sencha CMD (version 4 or above) command to make a production build of the application and start the embedded Web server:

```
sencha app build sencha web start
```

Finally, open this application at <http://localhost:1841> in one of the emulators or just on your desktop browser. You'll see the starting page that looks as in [Figure 12-8](#).

AT&T 11:20 AM 100%

# Save Sick Child

Login



**Who We Are**

**What We Do**

**Where We Work**

**Way To Give**

**About** **Donate** **Stats** **Events** **Media** **Share**

We'll review the code of this application next.

## The Application Object

The code of the app.js in the Save The Child project is shown below (we've just removed the default startup images and icons for brevity). For the most part it has the same structure as Ext JS applications.

```
Ext.application({
    name: 'SSC',
    requires: [
        'Ext.MessageBox'
    ],
    views: [
        'About',
        'CampaignsMap',
        'DonateForm',
        'DonorsChart',
        'LoginForm',
        'LoginToolbar',
        'Main',
        'Media',
        'Share',
        'ShareTile'
    ],
    stores: [
        'Campaigns',
        'Countries',
        'Donors',
        'States',
        'Videos'
    ],
    controllers: [
        'Login'
    ],
    launch: function() {
        // Destroy the #appLoadingIndicator element
        Ext.fly('appLoadingIndicator').destroy();

        // Initialize the main view
        Ext.Viewport.add(Ext.create('SSC.view.Main'));
    },
    onUpdated: function() {
        Ext.Msg.confirm(
            "Application Update",
            "This application has just successfully been updated to the latest version. Reload now?"
        );
    }
});
```

```

        function(buttonId) {
            if (buttonId === 'yes') {
                window.location.reload();
            }
        );
    });
});

```



Compare this application object with the Ext JS one shown in Chapter 4 in the section on Model View Controller - they are similar.

The application loads all the dependencies listed in app.js and will instantiate models and stores. The views that require data from the store will either mention the store name like `store: 'Videos'` or will use the `get` method from the class `StoreMgr`, for example `Ext.StoreMgr.get('Campaigns');`. After this is done, the `launch` function will be called - this is where the main view is created.

In this version of the Save The Child application we have only one controller `Login` that doesn't use any stores, but the mechanism of pointing controllers to the appropriate store instances is the same as for views. The application instantiates all controllers automatically. Accordingly, all controllers live in the context of the `Application` object.

We don't use explicitly defined models here - all the data are hard-coded in the stores in the `data` attributes.

You'll see the code of the views a bit later, but we wanted to draw your attention to the `onUpdated()` event handler. In the section "Microloader and Configurations" we've mentioned that production builds of Sencha Touch applications are watching the locally cached JavaScript and CSS files listed in the JS and CSS sections of the configuration file `app.json` and compare them with their peers on the server. They also watch all the files listed in the `appCache` section of `app.json`. If any of these files changes, the `onUpdated` event handler is invoked. For illustration purposes we decided to intercept this event and [Figure 12-9](#) shows how the update prompt can look like on iPhone 5.

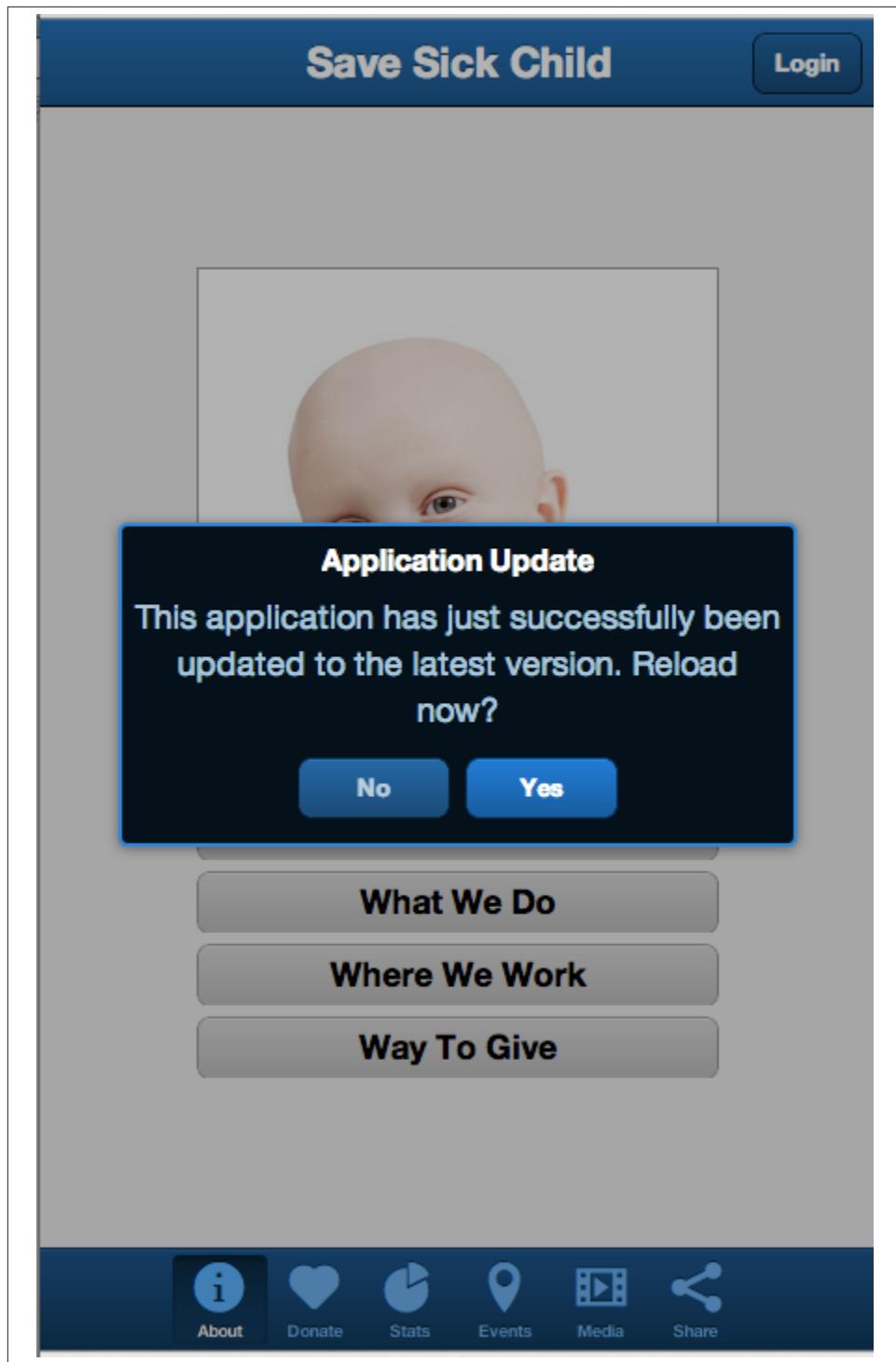


Figure 12-9. The code on the server has changed

At this point the user can either select working with the previous version of the application or reload the new one.

Our index.html file beside the microloader script includes one more script that supports Google Maps API.

```
<script type="text/javascript" src="http://maps.google.com/maps/api/js?sensor=true"></script>
```



If you want your program documentation look as good as Sencha's use [JSDuck](#) tool.

## The Main View

The code of the UI landing page of this application is located in the *views* folder in the file Main.js. First, take a look at the screen shot from WebStorm IDE on figure [Figure 12-10](#) that there are only two objects on the top level: the container and a login form.

```
▼ ssctouch (~/Documents/Farata/Enterprise)
  ▶ .sencha
  ▶ app
    ▶ controller
      Login.js
    ▶ store
    ▶ view
      About.js
      CampaignsMap.js
      DonateForm.js
      DonorsChart.js
      LoginForm.js
      LoginToolbar.js
      Main.js
      Media.js
      Share.js
      ShareTile.js

mainview
Ext.define('SSC.view.Main', {
  extend: 'Ext.Container',
  xtype: 'mainview',
  requires: [
    'Ext.tab.Panel',
    'Ext.Map',
    'Ext.Img'
  ],
  config: {
    layout: 'card',
    items: [
      {"xtype": "container"...},
      {"xtype": "loginform"...}
    ]
});
```

Figure 12-10. The Main.js in a collapsed form

The **card** layout means that the user will see either the content of that container or the login form - one at a time. Let's open up the container. It has an array of children, which are our application pages. The figure [Figure 12-11](#) shows the titles of the children.

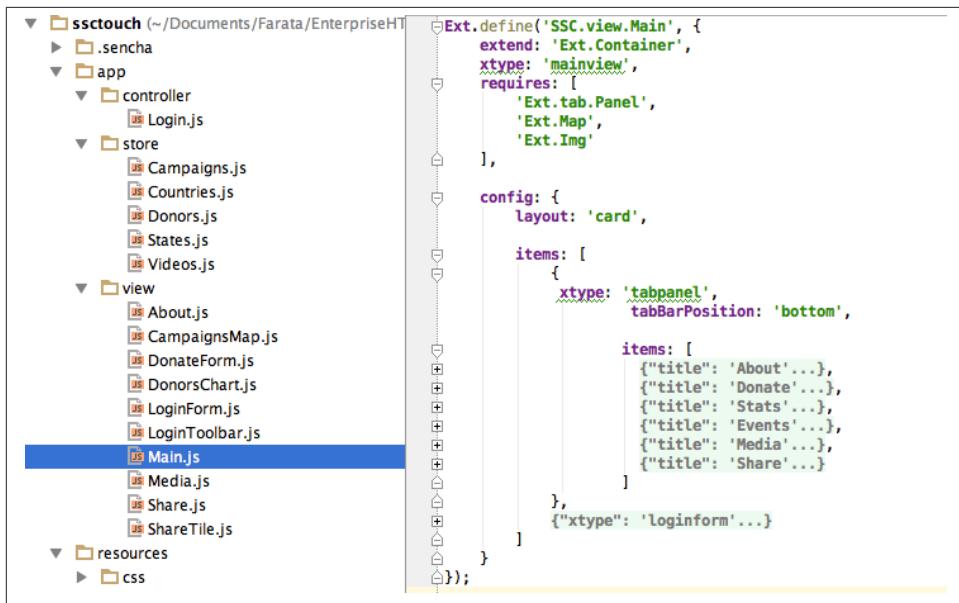


Figure 12-11. TabPanel's children in a collapsed form

The entire code of the Main.js is shown next.

```
Ext.define('SSC.view.Main', {
    extend: 'Ext.Container',
    xtype: 'mainview', // ①
    requires: [
        'Ext.tab.Panel',
        'Ext.Map',
        'Ext.Img'
    ],

    config: {
        layout: 'card',
        items: [
            {
                xtype: 'tabpanel', // ②
                tabBarPosition: 'bottom',
                items: [
                    {
                        title: 'About',
                        iconCls: 'info', // ③
                        layout: 'fit', // ④
                        items: [
                            {xtype: 'aboutview'}
                        ]
                    }
                ]
            },
            {"xtype": "loginform"...}
        ]
    }
});
```

```

        ],
    },
{
    title: 'Donate',
    iconCls: 'love',
    layout: 'fit',
    items: [
        {xtype: 'logintoolbar', // ⑤
         title: 'Donate'
        },
        {xtype: 'donateform'
        }
    ]
},
{
    title: 'Stats',
    iconCls: 'pie',
    layout: 'fit',
    items: [
        {xtype: 'logintoolbar',
         title: 'Stats'
        },
        {xtype: 'donorschart'
        }
    ]
},
{
    title: 'Events',
    iconCls: 'pin',
    layout: 'fit',
    items: [
        {xtype: 'logintoolbar',
         title: 'Events'
        },
        {xtype: 'campaignsmap'
        }
    ]
},
{
    title: 'Media',
    iconCls: 'media',
    layout: 'fit',
    items: [
        {xtype: 'mediaview'
        }
    ]
},
{
    title: 'Share',
    iconCls: 'share',
    layout: 'fit',
    items: [

```

```

        {xtype: 'logintoolbar',
         title: 'Share'
        },
        {xtype: 'sharereview'
        }
    ]
}
],
{
xtype: 'loginform',
showAnimation: {
    type: 'slide',
    direction: 'up',
    duration: 200
}
}
]
}
);

```

- ➊ We've assigned the `xtype: 'mainview'` to the main view so to allow the Login controller refer to it (see its code below).
- ➋ Note that the `tabpanel` doesn't explicitly specify any layout - it uses `card` by default.
- ➌ Each of the tabs has a corresponding button on the toolbar. It shows the text from the `title` attribute and the icon specified in the class `iconCls`.
- ➍ Each of the view has `fit layout`, which forces the content to expand to fill the layout's container.
- ➎ Each view will have a Login button on the toolbar. It's implemented in the `LoginToolbar.js` shown later in this chapter.

Sencha Touch can render icons using icon fonts from [Pictos library](#) located in the folder `resources/sass/stylesheets/fonts`. We've used icon fonts in the jQuery Mobile version of our application, and in this version we'll also fonts, which take a lot less memory than images. Below is the content of our `app.scss` file that includes several font icons used in the Save The Child application.

```

@import 'sencha-touch/default';
@import 'sencha-touch/default/all';

@include icon-font('IcoMoon', inline-font-files('icomoon/icomoon.woff', woff,
'icomoon/icomoon.ttf', truetype,'icomoon/icomoon.svg', svg));
@include icon('info', '!', 'IcoMoon');
@include icon('love', "", 'IcoMoon');
@include icon('pie', '#', 'IcoMoon');
@include icon('pin', '$', 'IcoMoon');
@include icon('media', '%', 'IcoMoon');

```

```

@include icon('share', '&', 'IcoMoon');

.child-img {
    border: 1px solid #999;
}

// Reduce size of the icons to fit 6 buttons in the tabbar; add Share tab
.x-tabbar.x-docked-bottom .x-tab {
    min-width: 2.8em;

    .x-button-icon:before {
        font-size: 1.4em;
    }
}

// Share icons
.icon-twitter, .icon-facebook, .icon-google-plus, .icon-camera {
    font-family: 'icomoon';
    speak: none;
    font-style: normal;
    font-weight: normal;
    font-variant: normal;
    text-transform: none;
    line-height: 1;
    -webkit-font-smoothing: antialiased;
}
.icon-twitter:before {
    content: "\27";
}
.icon-facebook:before {
    content: "\28";
}
.icon-google-plus:before {
    content: "\29";
}
.icon-camera:before {
    content: "\2a";
}

// Share tiles
.share-tile {
    top: 25%;
    width: 100%;
    position: absolute;
    text-align: center;
    border-width: 0 1px 1px 0;

    p:nth-child(1) {
        font-size: 4em;
    }
    p:nth-child(2) {

```

```

        margin-top: 1.5em;
        font-size: 0.9em;
    }
}

$sharetile-border: #666 solid;

.sharetile-twitter {
    border: $sharetile-border;
    border-width: 0 1px 1px 0;
}

.sharetile-facebook {
    border: $sharetile-border;
    border-width: 0 0 1px;
}

.sharetile-gplus {
    border: $sharetile-border;
    border-width: 0 1px 0 0;
}

// Media
.x-videos {
    .x-list-item > .x-innerhtml {
        font-weight: bold;
        line-height: 18px;
        min-height: 88px;

        > span {
            display: block;
            font-size: 14px;
            font-weight: normal;
        }
    }

    .preview {
        float: left;
        height: 64px;
        width: 64px;
        margin-right: 10px;
        background-size: cover;
        background-position: center center;
        background: #eee;
        @include border-radius(3px);
        -webkit-box-shadow: inset 0 0 2px rgba(0,0,0,.6);
    }
}

.x-item-pressed,
.x-item-selected {
    border-top-color: #D1D1D1 !important;
}

```

```
}
```

The first two lines of the app.scss import the icons from the default theme. We've added several more. Note that we had to reduce the size of the icons to fit six buttons in the application's toolbar. All the @include statements use SASS mixin icon().

If you need more icons use the [IcoMoon application](#). Pick an icon there and press the button Font to generate the custom font (see Figure Figure 12-12). Download and copy the generated fonts into your `resources/sass/stylesheets/fonts` directory and add them to the app.scss using `@include icon-font` directive. The downloaded zip file will contain the fonts as well as index.html file that will show you the class name and the code of the generated font icon(s).



Figure 12-12. Generating twitter icon font with IcoMoon application

When you compile the SASS with [compass](#) (or build the application with Sencha CMD), the SASS styles are converted into a standard CSS file resources/css/app.css.

## Controller

Now let's review the code of the Login page controller, which reacts on the user's actions performed in the view LoginForm. The name of the controller's file is Login.js. It's located in the folder `controller`, and here's the code:

```
Ext.define('SSC.controller.Login', {
    extend: 'Ext.app.Controller',
    config: {
        refs: {
```

```

        mainView: 'mainview',                      // ①
        loginForm: 'loginform',                    // ②
        loginButton: 'button[action=login]', // ③
        cancelButton: 'loginform button[action=cancel]'
    },
    control: {                                     // ④
        loginButton: {
            tap: 'showLoginView'
        },
        cancelButton: {
            tap: 'cancelLogin'
        }
    },
    showLoginView: function () {
        this.getMainView(). setActiveItem(1); // ⑤
    },
    cancelLogin: function () {
        this.getMainView(). setActiveItem(0); // ⑥
    }
});

```

- ① Including the `mainView: 'mainview'` in the `refs` attribute forces Sencha Touch to generate a getter function `getMainView()` providing the access to the main view if need be.
- ② This controller uses components from the `LoginForm` view (it's code comes a bit later).
- ③ The `loginButton` is the one that has `action=login`. The `cancelButton` is the one that's located inside the `loginform` and has `action=cancel`.
- ④ Defining the event handlers for `tap` events for the buttons Login and Cancel from the `LoginForm` view.
- ⑤ The main view has two children (see [Figure 12-10](#)). When the user taps on the Login button, show the second child: `setActiveItem(1)`.
- ⑥ When the user clicks on the Cancel button, show the main container - the first child of the main view: `setActiveItem(0)`.



Controllers are automatically instantiated by the `Application` object. If you want some controller's code to be executed even before the application `launch` function is called, put it in the `init` function. If you want some code to be executed right after the application is launched, put it in the controller's `launch` function.

For illustration purposes we'll show you a shorter (but not necessarily better) version of the Login.js. The above code defines the reference to the login form and button selectors in the `refs` section. Sencha Touch will find the references and will generate the getter for these buttons. But in this particular example we are using these buttons only to assign them the event handlers. Hence, we can make the `refs` section slimmer and use the selectors right inside the `control` section as shown below.

```
Ext.define('SSC.controller.Login', {
    extend: 'Ext.app.Controller',

    config: {
        refs: {
            mainView: 'mainview',
        },
        control: {
            'button[action=login]': {
                tap: 'showLoginView'
            },
            'loginform button[action=cancel]': {
                tap: 'cancelLogin'
            }
        }
    },
    showLoginView: function () {
        this.getMainView().setActiveItem(1);
    },
    cancelLogin: function () {
        this.getMainView().setActiveItem(0);
    }
});
```

This version of the Login.js is shorter, but the first one is more generic. In both versions the button selectors are the shortcuts for the `ComponentQuery` class, which is a singleton used for searching of components.

With MVC pattern, the event processing logic is often located in controller classes. Using `refs` and `ComponentQuery` selectors allows you to reach event generating objects located different classes. For example, if the user tapped on a button in a view, controller's code includes the `tap` event handler, where it triggers an event on a store class to initiate the data retrieval.

But if the `control config` is defined not in the controller, but in a component, the scope where `ComponentQuery` operates is limited to the component itself. You'll see the example of using the `control config` inside `DonateForm.js` later in this chapter.

## The Other Views

Let's do a brief code review of other Save The Child views.

### LoginForm

Figure [Figure 12-13](#) is a snapshot of Login view taken from iPhone 5, which was the only mobile device we've tested this application on.

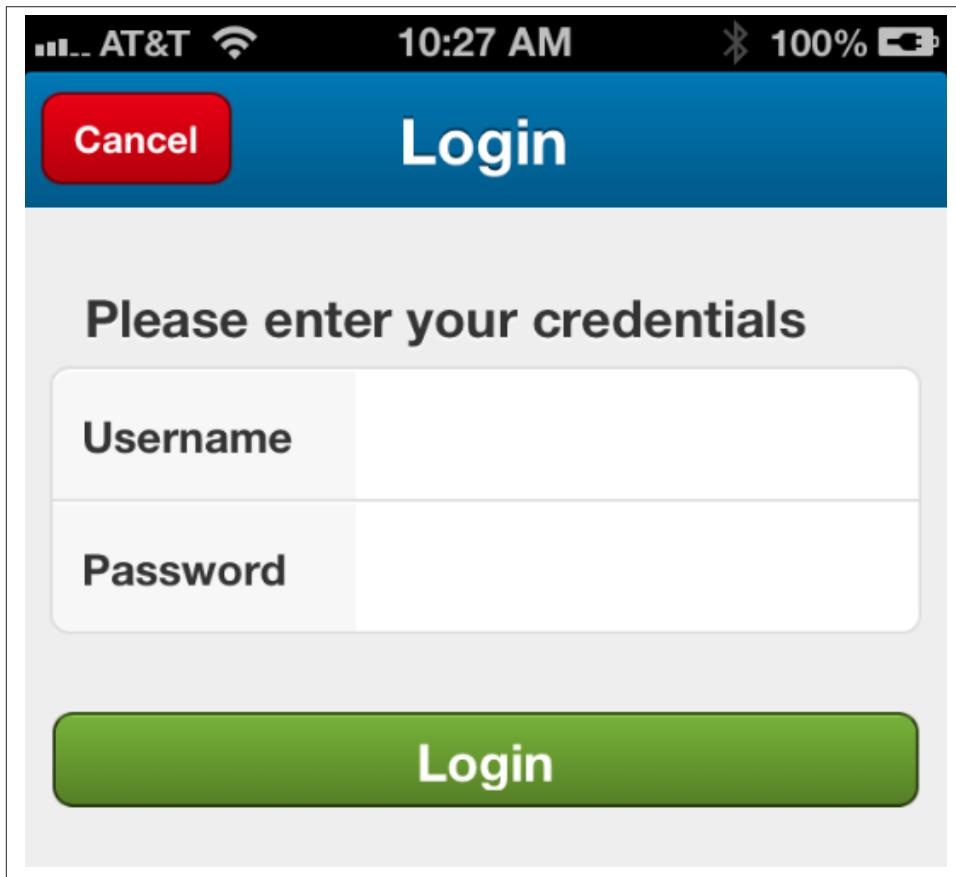


Figure 12-13. The Login Form View

This is how the code of the Login form view looks like - it's self explanatory. The `ui: 'decline'` is the [Ext.Button style](#) that causes the Cancel button have a red background.

```
Ext.define('SSC.view.LoginForm', {  
    extend: 'Ext.form.Panel',  
    xtype: 'loginform',  
    requires: [
```

```

        'Ext.field.Password'
    ],
    config: {
        items: [
            { xtype: 'toolbar',
                title: 'Login',
                items: [
                    { xtype: 'button',
                        text: 'Cancel',
                        ui: 'decline',
                        action: 'cancel'
                    }
                ]
            },
            { xtype: 'fieldset',
                title: 'Please enter your credentials',
                defaults: {
                    labelWidth: '35%'
                },
                items: [
                    { xtype: 'textfield',
                        label: 'Username'
                    },
                    { xtype: 'passwordfield',
                        label: 'Password'
                    }
                ]
            },
            { xtype: 'button',
                text: 'Login',
                ui: 'confirm',
                margin: '0 10'
            }
        ]
    }
});

```



One of the reviewers of this book reported that the text fields from this Login form are not shown on his Android Nexus 4 smartphone. This can happen, and in the real world applications should be tested in a variety of mobile devices. If you run into a similar situation while developing your application with Sencha Touch, use [platform-specific themes](#), which are automatically loaded based on the detected user's platform (see [platformConfig object](#)). Sencha Touch offers a number of [out of the box schemes](#) and [theme switching capabilities](#).

The Login form will be displayed when the user clicks on the button Login that is displayed on each other page in the toolbar. For example, the Figure Figure 12-14 shows the top portion of the Donate view.

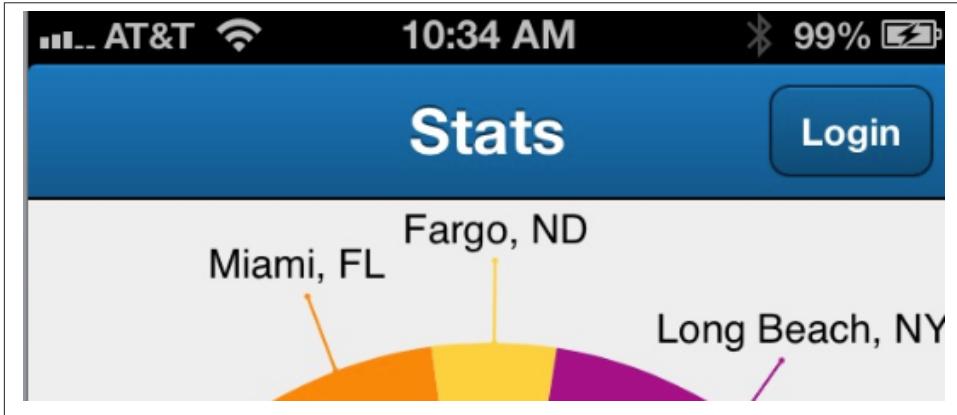


Figure 12-14. The Login Toolbar

The Login button is added as xtype: 'logintoolbar' to the top of each view in the Main.js. It's implemented in the LoginToolbar.js shown next.

```
Ext.define('SSC.view.LoginToolbar', {
    extend: 'Ext.Toolbar',
    xtype: 'logintoolbar',

    config: {
        title: 'Save The Child',
        docked: 'top', // ①
        items: [
            {
                xtype: 'spacer' // ②
            },
            {
                xtype: 'button',
                action: 'login',
                text: 'Login'
            }
        ]
    }
});
```

- ① The login toolbar has to located at the top of the screen

- ② Adding the Ext.Spacer component to occupy all the space before the button Login. By default, spacer has flex value of 1, which means take all the space in this case. You can read more about it in Chapter 4 in the section “The flex Property”.



If you'll add the Save The Child application as an icon to the home screen on iOS devices, the browser's address bar will not be displayed.

## DonateForm

We wanted to make the Donate view look as per our Web designer's mockup shown on Figure [Figure 11-10](#). With jQuery Mobile it was simple - the HTML container `<fieldset data-role="controlgroup" data-type="horizontal" id="radio-container">` with a bunch of `<input type="radio">` rendered the horizontal button bar shown on Figure [Figure 11-25](#). Here the fragment from the initial Sencha Touch version of `DonateForm.js`.

```
config: {
    title: 'DonateForm',

    items: [
        { xtype: 'fieldset',
            title: 'Please select donation amount',

            defaults: {
                name: 'amount',
                xtype: 'radiofield'
            },

            items: [
                { label: '$10',
                    value: 10
                },
                { label: '$20',
                    value: 20
                },
                { label: '$50',
                    value: 50
                },
                { label: '$100',
                    value: 100
                }
            ]
        },
        { xtype: 'fieldset',
            title: '... or enter other amount',
            ...
        }
    ]
},
```

```
        items: [
            { xtype: 'numberfield',
                label: 'Amount',
                name: 'amount'
            }
        ]
    }
```

It's also a **fieldset** with several radio buttons - `xtype: 'radiofield'`. But the result was not what we expected. These four radio buttons occupied half of the screen and looked as on Figure [Figure 12-15](#):

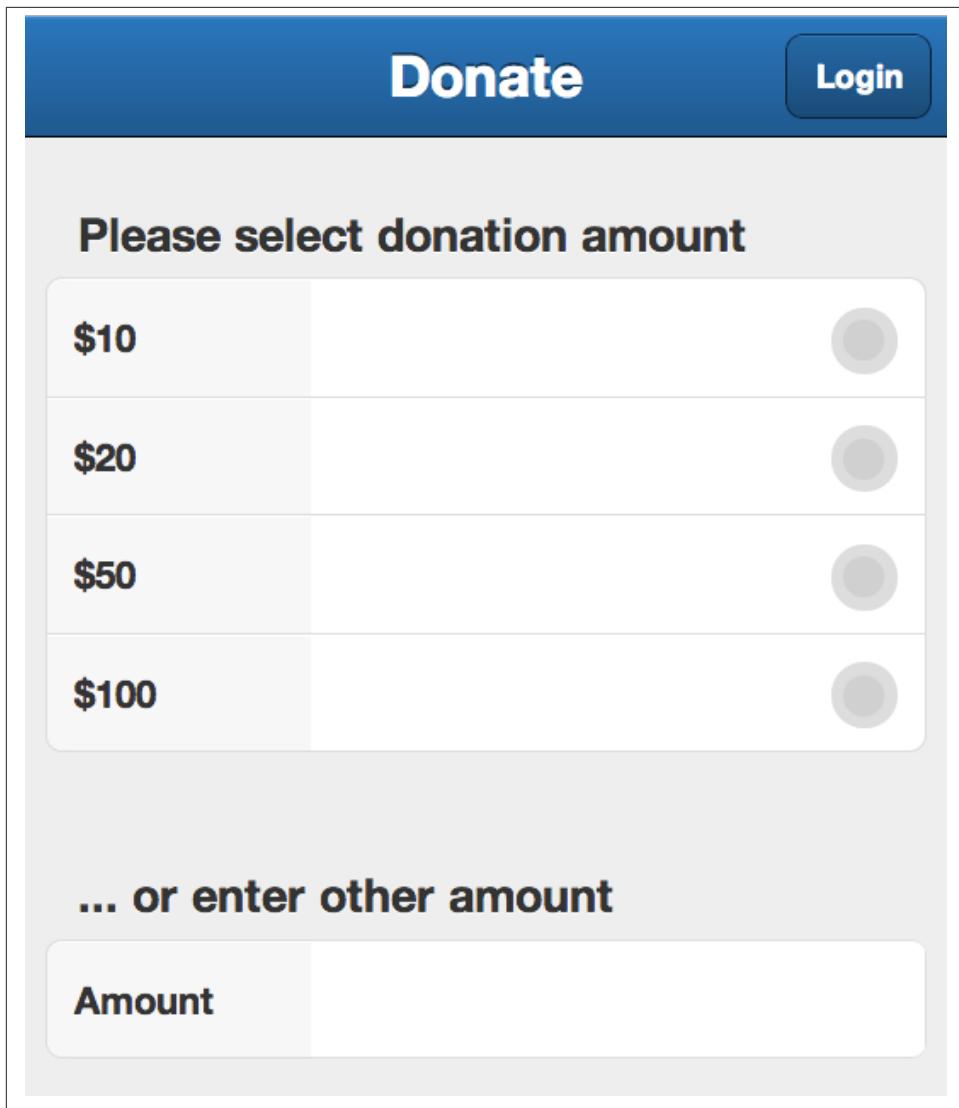


Figure 12-15. Rendering of xtype radiofield

After doing some research, we found out that Sencha Touch has the UI component called [Ext.SegmentedButton](#) that allows create horizontal bar with a number of toggle buttons, which is exactly what was needed from the rendering perspective. The resulting Donate screen is shown on Figure [Figure 12-16](#).

AT&T 2:04 PM 100%

Donate Login

Please select donation amount:

\$10 \$20 \$50 \$100

... or enter other amount

Amount

Donor information

Full name

Email

Location

Address

About Donate Stats Events Media Share

Figure 12-16. Donate form with SegmentedButton

This looks nice, but as opposed to regular HTML form with inputs, the `SegmentedButton` is not an HTML `<input>` field and its value won't be automatically submitted to the server. This required a little bit of a manual coding, which will be explained as a part of the `DonateForm` code review, which follows (we've split it into two fragments for better readability).

```
Ext.define('SSC.view.DonateForm', {
    extend: 'Ext.form.Panel',
    xtype: 'donateform',
    requires: [
        'Ext.form.FieldSet',
        'Ext.field.Select',
        'Ext.field.Number',
        'Ext.field.Radio',
        'Ext.field.Email',
        'Ext.field.Hidden',
        'Ext.SegmentedButton',
        'Ext.Label'
    ],
    config: {
        title: 'DonateForm',
        control: {
            'segmentedbutton': { // ①
                toggle: 'onAmountButtonChange'
            },
            'numberfield[name=amount]': {
                change: 'onAmountFieldChange'
            }
        },
        items: [
            { xtype: 'label',
                cls: 'x-form-fieldset-title', // ②
                html: 'Please select donation amount:'
            },
            { xtype: 'segmentedbutton', // ③
                margin: '0 10',
                defaults: {
                    flex: 1
                },
                items: [
                    { text: '$10',
                        data: {
                            value: 10 // ④
                        }
                    },
                    { text: '$20',
                        data: {
                            value: 20
                        }
                    }
                ]
            }
        ]
    }
});
```

```

        data: {
            value: 20
        }
    },
    { text: '$50',
        data: {
            value: 50
        }
    },
    { text: '$100',
        data: {
            value: 100
        }
    }
]
},
{ xtype: 'hiddenfield',           // ⑤
    name: 'amount'
},

```

- ❶ Defining event listeners for the `segmentedbutton` and the field for entering other amount. When the control section is used not in a controller, but in a component it's scoped to the object in which it was defined. Hence the `ComponentQuery` will be looking for `segmentedbutton` and `numberfield[name=amount]` only within the `DonateForm` instance. If these event handlers would be defined in the controller, the scope would be global.
- ❷ Borrowing the class that Sencha Touch uses for all `fieldset` container so our title looks the same.
- ❸ The `segmentedbutton` is defined here. By default, its config property `allowToggle=true`, which allows only one button to be pressed at a time.
- ❹ The `segmentedbutton` has no property to store the value of each of its button. But any subclass of `Ext.Component` has a property `data`. We are extending the `data` property to store the button's value. It'll be available in the event handler in `button.getData().value`.
- ❺ Since the buttons in the `segmentedbutton` are not input fields, we define a hidden field to remember the currently selected amount.

The second half of `SSC.view.DonateForm` comes next.

```

{ xtype: 'fieldset',
    title: '... or enter other amount',

    items: [
        { xtype: 'numberfield',      // ❻
            label: 'Amount',
            name: 'amount'
        }
    ]
}

```

```

        ]
    },
{
    xtype: 'fieldset',
    title: 'Donor information',
    items: [
        { name: 'fullName',
          xtype: 'textfield',
          label: 'Full name'
        },
        { name: 'email',
          xtype: 'emailfield',
          label: 'Email'
        }
    ]
},
{
    xtype: 'fieldset',
    title: 'Location',
    items: [
        { name: 'address',
          xtype: 'textfield',
          label: 'Address'
        },
        { name: 'city',
          xtype: 'textfield',
          label: 'City'
        },
        { name: 'zip',
          xtype: 'textfield',
          label: 'Zip'
        },
        { name: 'state',
          xtype: 'selectfield',
          autoSelect: false,
          label: 'State',
          store: 'States',
          valueField: 'id',
          displayField: 'name'
        },
        { name: 'country',
          xtype: 'selectfield',
          autoSelect: false,
          label: 'Country',
          store: 'Countries',
          valueField: 'id',
          displayField: 'name'
        }
    ]
},

```

```

        {
            xtype: 'button',
            text: 'Donate',
            ui: 'confirm',
            margin: '0 10 20'
        }
    ],
},
onAmountButtonChange: function (segButton,
                                button, isPressed) { // ②
    if (isPressed) { // ③
        this.clearAmountField();
        this.updateHiddenAmountField(button.getData().value);
        button.setUi('confirm'); // ④
    }
    else {
        button.setUi('normal');
    }
},
onAmountFieldChange: function () { // ⑤
    this.depressAmountButtons();
    this.clearHiddenAmountField();
},
clearAmountField: function () {
    var amountField = this.down('numberfield[name=amount]');
    amountField.suspendEvents(); // ⑥
    amountField.setValue(null);
    amountField.resumeEvents(true); // ⑦
},
updateHiddenAmountField: function (value) {
    this.down('hiddenfield[name=amount]').setValue(value);
},
depressAmountButtons: function () {
    this.down('segmentedbutton').setPressedButtons([]);
},
clearHiddenAmountField: function () {
    this.updateHiddenAmountField(null);
}
});

```

- ① This numberfield stores the *other amount* if entered. Note that it has the same name `amount` as the hidden field. The methods `clearAmountField()` and `clearHiddenAmountField()` will ensure that only one of the amounts has a value.

- ② When the `toggle` event is fired it comes with an object that contains the reference to the button that was toggled, and if the button becomes pressed as the result of this event.
- ③ The `toggle` event is dispatched twice - one for the button that becomes pressed, and another for the button that was pressed before. If the button becomes pressed (`isPressed=true`), clean the previously selected amount and store a new one in the hidden field.
- ④ Change the style of the button to make it visibly highlighted. We use the predefine`confirm` style (see the [Kitchen Sink](#) application for other button styles).
- ⑤ When the *other amount* field loses focus, this event handler is invoked. The code cleans up the hidden field and removes the pressed state from all buttons.
- ⑥ Temporarily suspend dispatching events while setting the value of the amount `numberfield` to null. Otherwise setting to null would cause unnecessary dispatching of the `change` event.
- ⑦ Resume event dispatching. The `true` argument is for discarding all the queued events.

Previous versions of the Save The Child application illustrated how to submit the Donate form to the server for further processing. The Sencha Touch version of this application doesn't include this code. If you'd like to experiment with this, just create a new controller class that extends `Ext.app.Controller` and define there an event handler for the button `Donate` (see the `Login` controller as an example).

On the `tap` event invoke `donateform.submit()` specifying the URL of the server that knows how to process this form. You can find details on submitting and populating forms in the online documentation for [Ext.form.Panel](#) - the ancestor of the `Donate Form`.



If you want to use the AJAX-based form submission, use `submit()`, otherwise use the method `standardSubmit()`, which will do a standard HTML form submission.

## Charts

The charting support is just great in Sencha Touch (and similar to Ext JS). It's JavaScript based, the charts are live and can get the data from the stores and model. The Figure [Figure 12-17](#) shows how the chart looks on iPhone when the user selects the Stats page:

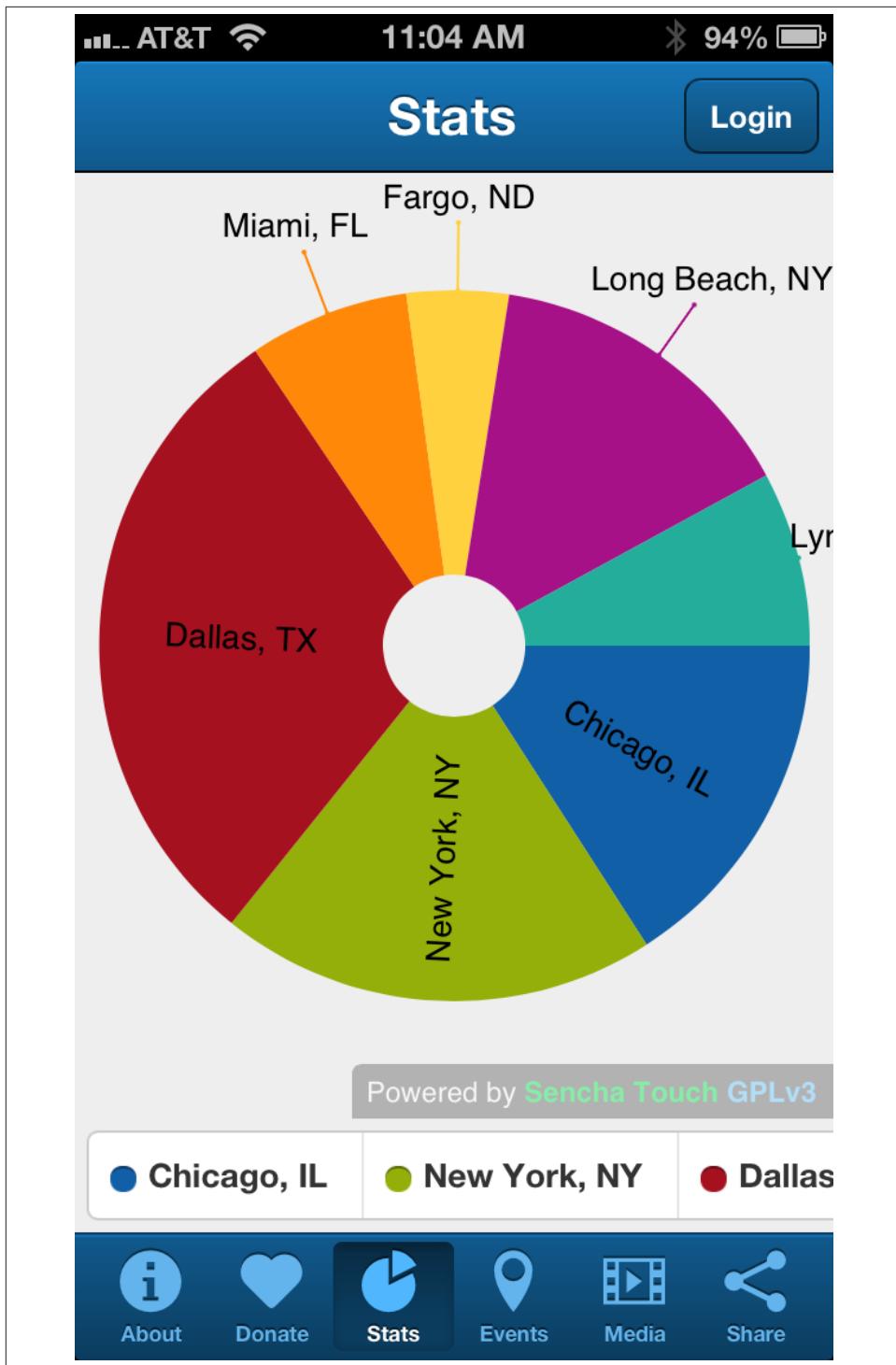


Figure 12-17. Donor's statistics chart

The code that support the UI part of the chart is located in the view `DonorsChart` that's shown next. It uses he classes located in the Sencha Touch framework in the folder `src/chart`.

```
Ext.define('SSC.view.DonorsChart', {
    extend: 'Ext.chart.PolarChart',           // ①
    xtype: 'donorschart',

    requires: [
        'Ext.chart.series.Pie',               // ②
        'Ext.chart.interactions.Rotate'
    ],

    config: {
        store: 'Donors',                   // ③
        animate: true,
        interactions: ['rotate'],

        legend: {                         // ④
            inline: false,
            docked: 'left',
            position: 'bottom'
        },
        series: [
            {
                type: 'pie',
                donut: 20,
                xField: 'donors',
                labelField: 'location',
                showInLegend: true,
                colors: ["#115fa6", "#94ae0a", "#a61120", "#ff8809",
                         "#ffd13e", "#a61187", "#24ad9a", "#7c7474", "#a66111"]
            }
        ]
    }
});
```

- ① Create a chart that uses polar coordinates.
- ② The `Rotate` class allows the user to rotate (with a finger) a polar chart around its central point.
- ③ The data shown on the chart come from the store named `Donors`, which is shown in the section “Stores and Models”.
- ④ The legend is a bar at the bottom of the screen. The user can horizontally scroll it with a finger.

## Media

The Media page of our application displays the list of available videos. When the user taps on one of them, the new page opens where the user have to tap on the button play. We use the `Ext.dataview.List` component to display video titles from the `Videos` store.

The `Media` view extends `Ext.NavigationView`, which is a container with the card layout that also allows to push a new view into this container - we use it to create a view for the selected from the list video. The code of the `Media` view is shown in the next listing.

```
Ext.define('SSC.view.Media', {
    extend: 'Ext.NavigationView',
    xtype: 'mediaview',
    requires: [
        'Ext.Video' // ①
    ],

    config: {
        control: {
            'list': {
                itemtap: 'showVideo' // ②
            }
        },
        useTitleForBackButtonText: true, // ③
        navigationBar: {
            items: [
                { xtype: 'button',
                    action: 'login',
                    text: 'Login',
                    align: 'right'
                }
            ]
        },
        items: [
            { title: 'Media',
                xtype: 'list',
                store: 'Videos',
                cls: 'x-videos',
                variableHeights: true,
                itemTpl: [ // ④
                    '<div class="preview"'
                    style="background-image:url(resources/media/{thumbnail});"></div>',
                    '{title}',
                    '<span>{description}</span>'
                ]
            }
        ],
        showVideo: function (view, index, target, model) {
    
```

```

        this.push(Ext.create('Ext.Video', {           // ⑤
            title: model.get('title'),
            url: 'resources/media/' + model.get('url'),
            posterUrl: 'resources/media/' + model.get('thumbnail')
        }));
    }
});

```

- ① Sencha Touch offers `Ext.Video` a wrapper for the HTML5 `<video>` tag. In Chapter 4 we used the HTML5 tag `<video>` directly.
- ② Defining the event listener for the `itemtap` event, which fires whenever the list item is tapped.
- ③ When the video player's view will be pushed to the Media page, we want its Back button to display the previous view's title, which is "Media". It's a config property in the `NavigationView`.
- ④ The list with descriptions of videos is populated from the store `Videos` using the list's config property `'itemTpl'`. This is an HTML template for rendering each item. We decided to use the `<div>` showing the content of store's properties `title`, `description` with a background image from the property `thumbnail`, and the video located at the specified `url`. The source code of the store `Videos` is included in the section "Stores and Models" below.
- ⑤ Create a video player and push it into the `NavigationView`. When the `itemtap` event is fired, it passes several values to the function handler. We just use the `model` that corresponds to the tapped list item. For all available config properties refer to the [Ext.Video documentation](#).



A template [Ext.Template] represents an HTML fragment. The values in curly braces are being passed to the template from the outside. In the above example the values are coming from the store `Videos`. The class `Ext.XTemplate` offers advanced templating, e.g. auto-filling HTML with the data from an array, which is used here.

## Maps

Integration with Google Maps is a pretty straightforward task in Sencha Touch, which comes with `Ext.Map` - a wrapper class for Google Maps API. Our view `CampainsMap` is a subclass of `Ext.Map`. Note that we've imported Google Maps API in the file `index.html` as follows:

```
<script type="text/javascript" src="http://maps.google.com/maps/api/js?sensor=true"></script>
```

Figure [Figure 12-18](#) shows the iPhone's screen when the button Events is pressed.

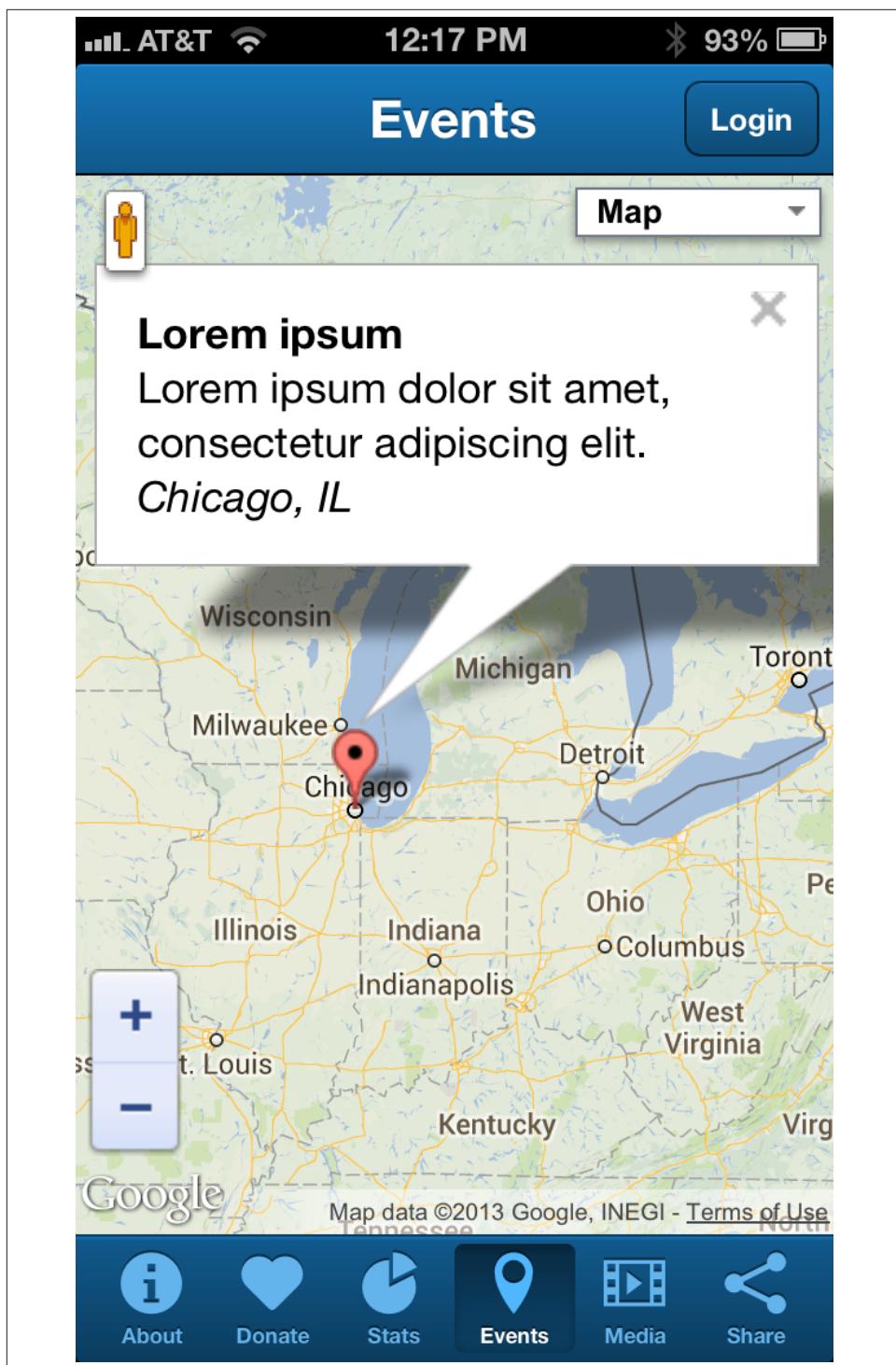


Figure 12-18. The Events page

Of course, some additional styling would be needed before offering this view in production environment, but our CampaignsMap.js that supports this screen is only ninety lines of code!

```
Ext.define('SSC.view.CampaignsMap', {
    extend: 'Ext.Map',
    xtype: 'campaignsmap',

    config: {                                     // ①
        listeners: {
            maprender: function () {           // ②

                if (navigator && navigator.onLine) {
                    try {
                        this.initMap();
                        this.addCampaignsOnTheMap(this.getMap());
                    } catch (e) {
                        this.displayGoogleMapError();
                    }
                } else {
                    this.displayGoogleMapError();
                }
            }
        },
        initMap: function () {

            // latitude = 39.8097343 longitude = -98.55561990000001
            // Lebanon, KS 66952, USA Geographic center
            // of the contiguous United States
            // the center point of the map

            var latMapCenter = 39.8097343,
                lonMapCenter = -98.55561990000001;

            var mapOptions = {
                zoom      : 3,
                center   : new google.maps.LatLng(latMapCenter, lonMapCenter),
                mapTypeId: google.maps.MapTypeId.ROADMAP,
                mapTypeControlOptions: {
                    style   : google.maps.MapTypeControlStyle.DROPDOWN_MENU,
                    position: google.maps.ControlPosition.TOP_RIGHT
                }
            };

            this.setMapOptions(mapOptions);
        },
        addCampaignsOnTheMap: function (map) {
            var marker,
                infowindow = new google.maps.InfoWindow(),

```

```

geocoder = new google.maps.Geocoder(),
campaigns = Ext.StoreMgr.get('Campaigns');

campaigns.each(function (campaign) {
    var title = campaign.get('title'),
        location = campaign.get('location'),
        description = campaign.get('description');

    geocoder.geocode({
        address: location,
        country: 'USA'
    }, function(results, status) {
        if (status == google.maps.GeocoderStatus.OK) {

            // getting coordinates
            var lat = results[0].geometry.location.lat(),
                lon = results[0].geometry.location.lng();

            // create marker
            marker = new google.maps.Marker({
                position: new google.maps.LatLng(lat, lon),
                map: map,
                title: location
            });

            // adding click event to the marker to show info-bubble with data from json
            google.maps.event.addListener(marker, 'click', (function(marker) {
                return function () {
                    var content = Ext.String.format(
                        '<p class="infowindow"><b>{0}</b><br/>{1}<br/><i>{2}</i></p>',
                        title, description, location);

                    infowindow.setContent(content);
                    infowindow.open(map, marker);
                };
            })(marker));
        } else {
            console.error('Error getting location data for address: ' + location);
        }
    });
},
displayGoogleMapError: function () {
    console.log("Sorry, Google Map service isn't available");
}
));

```

- ❶ We just use `listeners` config here, but `Ext.Map` has 60 of them. For example, if we wanted the mobile device to identify the current location of the device and put it in the center of the map, we'd add `useCurrentLocation: true`.

- ② This event is fired when the map is initially rendered. We are reusing the same code as in previous chapters for initializing the map (showing the central point of the USA) and adding the campaigns information. The code of the store Campaigns is shown in the section Stores and Models below.

Sencha Touch is a framework for mobile devices, which can be on the move. `Ext.util.Geolocation` is a handy class for applications that require to know the current position of the mobile device. When your program instantiates `Geolocation`, it starts tracking the location of the device by firing the `locationupdate` event periodically (you can turn auto updates off). The following code fragment shows how to get the current latitude of the mobile device.

```
var geo = Ext.create('Ext.util.Geolocation', {
    listeners: {
        locationupdate: function(geo) {
            console.log('New latitude: ' + geo.getLatitude());
        }
    }
});

geo.updateLocation(); // start the location updates
```

## Stores and Models

In the Sencha Touch version of the Save The Child application all the data is hard-coded. All store classes are located in the store directory (see Figure [Figure 12-11](#)), and each of them has the `data` property. For example, here's the code of the `Videos.js`.

```
Ext.define('SSC.store.Videos', {
    extend: 'Ext.data.Store',

    config: {
        fields: [
            { name: 'title', type: 'string' },
            { name: 'description', type: 'string' },
            { name: 'url', type: 'string' },
            { name: 'thumbnail', type: 'string' }
        ],
        data: [
            { title: 'The title of a video-clip 1', description: 'Short video description 1',
                url: 'intro.mp4', thumbnail: 'intro.jpg' },
            { title: 'The title of a video-clip 2', description: 'Short video description 2',
                url: 'intro.mp4', thumbnail: 'intro.jpg' },
            { title: 'The title of a video-clip 3', description: 'Short video description 3',
                url: 'intro.mp4', thumbnail: 'intro.jpg' }
        ]
    }
});
```

```
    }  
});
```



There is compatibility issue between Ext JS and Sencha Touch 2 stores and models. For example, in the above code `fields` and `data` are wrapped inside the `config` object, while in Ext JS store they are not. Until Sencha will offer a generic solution to resolve the compatibility issues, you have to come up with your own if you want to reuse the same stores.

The code of the Donors store supports the charts in the Stats page. It's self explanatory:

```
Ext.define('SSC.store.Donors', {  
    extend: 'Ext.data.Store',  
  
    config: {  
        fields: [  
            { name: 'donors', type: 'int' },  
            { name: 'location', type: 'string' }  
        ],  
  
        data: [  
            { donors: 48, location: 'Chicago, IL' },  
            { donors: 60, location: 'New York, NY' },  
            { donors: 90, location: 'Dallas, TX' },  
            { donors: 22, location: 'Miami, FL' },  
            { donors: 14, location: 'Fargo, ND' },  
            { donors: 44, location: 'Long Beach, NY' },  
            { donors: 24, location: 'Lynbrook, NY' }  
        ]  
    }  
});
```

The Campaigns store is used to display the markers on the map, where charity campaigns are active. Tapping on the marker will show the description of the selected campaign as shown on Figure [Figure 12-18](#) - we tapped on Chicago marker. The code of the store Campaigns.js is shown next.

```
Ext.define('SSC.store.Campaigns', {  
    extend: 'Ext.data.Store',  
  
    config: {  
        fields: [  
            { name: 'title', type: 'string' },  
            { name: 'description', type: 'string' },  
            { name: 'location', type: 'string' }  
        ],  
  
        data: [  
            {  
                title: 'Chicago Marathon',  
                description: 'A charity run organized by the Chicago Marathon Committee to raise funds for children in need.',  
                location: 'Chicago, IL'  
            },  
            {  
                title: 'New York City Marathon',  
                description: 'A charity run organized by the New York City Marathon Committee to raise funds for children in need.',  
                location: 'New York, NY'  
            },  
            {  
                title: 'Dallas Marathon',  
                description: 'A charity run organized by the Dallas Marathon Committee to raise funds for children in need.',  
                location: 'Dallas, TX'  
            },  
            {  
                title: 'Miami Marathon',  
                description: 'A charity run organized by the Miami Marathon Committee to raise funds for children in need.',  
                location: 'Miami, FL'  
            },  
            {  
                title: 'Fargo Marathon',  
                description: 'A charity run organized by the Fargo Marathon Committee to raise funds for children in need.',  
                location: 'Fargo, ND'  
            },  
            {  
                title: 'Long Beach Marathon',  
                description: 'A charity run organized by the Long Beach Marathon Committee to raise funds for children in need.',  
                location: 'Long Beach, NY'  
            },  
            {  
                title: 'Lynbrook Marathon',  
                description: 'A charity run organized by the Lynbrook Marathon Committee to raise funds for children in need.',  
                location: 'Lynbrook, NY'  
            }  
        ]  
    }  
});
```

```

        title: 'Mothers of Asthmatics',
        description: 'Mothers of Asthmatics - nationwide Asthma network',
        location: 'Chicago, IL'
    },
    {
        title: 'Lawyers for Children',
        description: 'Lawyers offering free services for The Children',
        location: 'New York, NY'
    },
    {
        title: 'Sed tincidunt magna',
        description: 'Donec ac ligula sit amet libero vehicula laoreet',
        location: 'Dallas, TX'
    },
    {
        title: 'Friends of Blind Kids',
        description: 'Semi-annual charity events for blind kids',
        location: 'Miami, FL'
    },
    {
        title: 'Place Called Home',
        description: 'Adoption of The Children',
        location: 'Fargo, ND'
    }
]
}
));

```

## Dealing With Landscape Mode

Handling the landscape mode with Sencha Touch is done differently depending on how you deploy your application. If you decide to [package this app as a native one](#), the landscape mode will be supported. Sencha CMD will generate the file packager.json, which will include the section dealing with orientation:

```

"orientations": [
    "portrait",
    "landscapeLeft",
    "landscapeRight",
    "portraitUpsideDown"
]

```

If you're not planning to package your app as the native one, you'll need to do some manual coding by processing the `orientationchange` event. For example,

```

Ext.Viewport.on('orientationchange', function() {
    // write the code to handle the landscape code here
});

```

This concludes the review of the Sencha Touch version of our sample application, which consists of six nice looking screens. The amount of manual coding to achieve this was

minimal. In the real world, you'd need to add business logic to this application, but comes down to inserting the JavaScript code to a well structured layers. The code to communicate with the server will go to the stores, the data will be placed in the models, the UI remains in the views, and the main glue of your application is controllers. Sencha Touch did a pretty good job for us, wouldn't you agree?

## Comparing jQuery Mobile and Sencha Touch

In chapters 11 and 12 you've learned about two different ways of developing a mobile application. So what's better jQuery Mobile or Sencha Touch? There is no answer to this question, and you will have to make a decision on your own. But here's a quick summary of pros and cons for each library or framework.

Use jQuery Mobile if:

- If you are afraid of being locked up with any one vendor. The effort to replace jQuery Mobile in your application with another framework (if you decide to do so) is a magnitude lower than switching from Sencha Touch to something else.
- If you need your application to work on most of the mobile platforms.
- If you prefer declarative UI and hate debugging JavaScript.

Use Sencha Touch if:

- If you like to have a rich library of pre-created UI.
- If your application needs smooth animation. Sencha Touch does automatic throttling based on the actual frames per seconds supported on the device.
- If splitting the application code into cleanly defined architectural layers (model-view-controller-service) is important.
- If you believe that using code generators add value to your project.
- If you want to be able customize and extend components to fit your application's needs perfectly. Yes, you'll be writing JavaScript, but it still may be simpler than trying to figure out the enhancements done to HTML component by jQuery Mobile under the hood.
- If you want to minimize the efforts required to package your application as a native one.
- If you want your application to look as close to the native ones as possible.
- If you prefer to use software that is covered by the commercial support offered by vendor.

While considering support options do not just assume that paid support translates into better quality. This is not to say that Sencha won't offer you quality support, but in many

cases having a large community of developers will lead to a faster solution to a problem that dealing with one assigned support engineer. Having said this, we'd like you to know that [Sencha forum](#) has about half a million registered users who are actively discussing problems and offering solutions to each other.

Even if you are a developer's manager, you don't have to make the framework choice on your own. Bring your team into a conference room, order pizza, and listen to what *your team members* have to say about these two or any other frameworks being considered. We offered you the information about two of many frameworks, but the final call is yours.

---

# Hybrid Mobile Applications

The word **hybrid** means something of mixed origin or composition. In the realm of mobile Web applications such a mix consists of the code written in HTML5, which access the APIs written in native languages. If an organization doesn't want or can't hire separate teams of software developers (e.g Objective-C developers for iPhone, Java for Android, C# for Windows Phone) there is a way to have one team of developers having HTML/JavaScript skills that will develop applications having the same code deployed on various mobile devices packaged as native applications. Let's do a quick comparison of native, Web and hybrid mobile applications.

## Native Applications

We call a mobile application *native* if it was written not in HTML/JavaScript, but in a programming language recommended for devices of this mobile platform. The manufacturer of mobile devices releases an SDK and describes a process of creating native applications. Such SDK provides an API for accessing all components (both hardware and software) of the mobile device such as phone, contact list, camera, microphone and others. Such SDKs include UI components that have a *native look and feel*, so applications developed by third parties look the same as those developed by the respective device manufacturer.

Native applications can seamlessly communicate with each other. They can use all available hardware and software components of the device to create convenient workflows that people quickly get accustomed too. For example, a person can take a picture with her mobile phone, which can figure out the current geographical location and allow to share the photo with other people from her Contacts list. To support such functionality a native application has to access the camera of the mobile device, use GPS to discover the device coordinates, and access the Contacts application.

If you are in the business of writing mobile flight simulators or games that heavily rely on graphics (not a Sudoku type of games) - select a programming language that can use the device hardware (e.g. graphic accelerators) to its fullest and works as fast as possible on this device. Faster applications use less battery power too.

For native applications, a device manufacturer usually offers an *application store*, which serves as an online market place where people would shop for applications. Apple has the *App Store* for iOS and MAC OS X applications. Google has *Google Play* market for Android applications. *BlackBerry World* is a store where you can find applications for mobile devices manufactured by RIM. Microsoft has their store too.

There are more application stores available, and having a one stop shop is a great way for distributing consumer-oriented applications. For enterprise applications having a public distribution channel may be less important, but enterprises still need a way to publish mobile applications for private use. Apple has **iOS Developer Enterprise Program**. For Android applications, there is a **Google Play Private Channel** for internal distribution channels. Microsoft has **their process** for business applications too.



HTML5 stack is not the only way to develop Hybrid applications using the same language for different mobile platforms. With **Xamarin** you can develop applications in C# for iOS, Android and Windows Phone.

## Native vs. Web Applications

Both Web and native applications have their pros and cons. The latter are usually faster than Web applications. Let's go through some of the examples of native mobile business applications that exist today.

Bank of America, Chase and other major banks have native mobile applications that allow you to deposit a check by taking a photo of its the front and back sides and entering the amount. At the time of this writing they support iPhones, Android, Windows phones, and iPads.

There are native applications implementing the Near Field Communication (NFC) technology allows NFC-enabled devices communicate with each other in close distance using radio frequencies. NFC can be used for payments (no need to enter passwords) and data sharing (contacts, photos, et al.) Proliferation of NFC in banking will seriously hurt the credit card industry. A number of smartphones already support NFC technology (see <http://www.nfcworld.com/nfc-phones-list>). Add one of the existing fingerprint biometrics solutions, and your mobile phone becomes your wallet.

While native applications have full access to all APIs of the mobile device (e.g. contacts, camera, microphone, et al.), they have drawbacks too. For instance, if you want to publish your application at Apple's App Store you have to submit your application in advance

and wait for its approval. If the users run into a crucial bug in your application, even if you fixed the same day, you can't put a new version in production until it goes through an approval process. Besides this inconvenience there can be other road blocks. For instance, back in 2011 Financial Times (FT) decided to stop using their native iOS application because Apple wouldn't agree to share the data about FT subscribers with FT - the owner of this application.

Mobile Web applications don't require any third-party involvement for the distribution. An enterprise can make them available at any time by simply adding a Download button on the corporate Web site. It's good to have an ability to quickly publish the latest versions of Web applications on your own servers without the need to ask for a permission. On the other hand, maintaining the presence on one of the popular app stores is a good channel for getting new customers.

The publisher of New York magazine is heavily investing into their native application for iPad, but the newer version of their Web application is as engaging as its native peer. If you want your application to be discoverable and visible by search engines, develop it as a Web application, not the native one.

## Hybrid Applications

*Hybrid applications* promise you to have the best of both worlds. Develop a Web application in HTML/JavaScript, but access the native API of the mobile device via third-party solutions such as [PhoneGap](#) from Adobe or [Titanium](#) from Appcelerator. Let's see what tools are available for creating hybrids.

### Cordova and PhoneGap

[Cordova](#) is a library (and a build tool) that serves as a bridge between JavaScript and native API. Cordova started from the code donated by Adobe to Apache Software Foundation. Cordova is an open source platform created for building mobile applications with HTML5, but packaged as native ones. PhoneGap is a brand owned by Adobe. Besides the Cordova library it offers developers a remote server, where they can package their applications for various mobile platforms. If the role of Cordova library in the PhoneGap product is not clear to you, think of a similar situation when the same software library is used in different products. For example, the rendering engine Webkit, is used in Chrome and Safari browsers. NOTE: Cordova can be used without PhoneGap. For example, Facebook and Salesforce use Cordova in their mobile SDK.

Figure [Figure 13-1](#) illustrates the interaction of PhoneGap, Cordova, and a Web application.

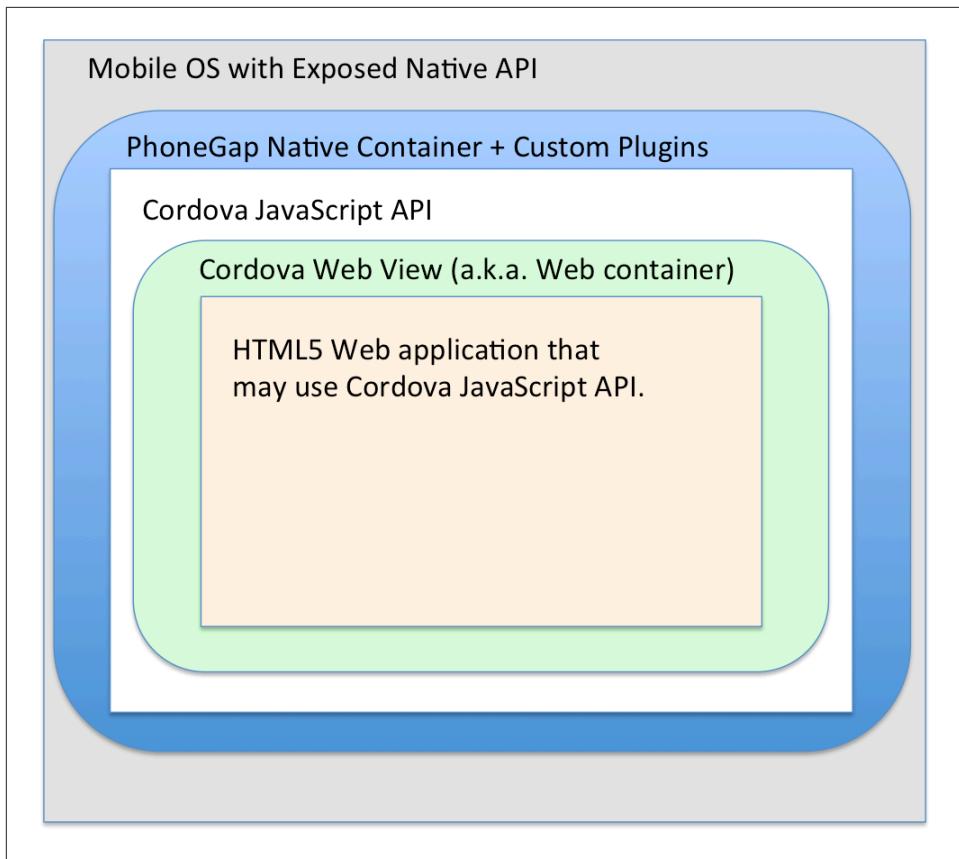


Figure 13-1. PhoneGap, Cordova and a Web Application

PhoneGap includes APIs, code generator, and a workflow for creating *native application containers* for Web applications written in HTML5 (with or without JavaScript frameworks). PhoneGap also allows making JavaScript calls for accessing native API offered by the mobile OS.

**PhoneGap Build** is a cloud service, where you upload your HTML/JavaScript/CSS code for packaging it for multiple mobile platforms. PhoneGap Build creates several native applications - one per mobile platform. Each application is a wrapper with embedded chrome-less Web browser ( a.k.a. Web View) that looks native to the mobile OS, and has access to various native APIs. Refer to [PhoneGap documentation](#) to see what APIs are supported on each mobile platform. The native wrapper serves as a messaging bus between the external native API and HTML-based applications running inside the Web view.

For iOS applications the PhoneGap Build server creates an *.ipa* file, for deployment on Android devices it generates an *.apk* file and so on. After that, if you want to submit your application to a public or private application store, follow the procedure that exists for native applications for the selected store. The PhoneGap Build service can package your application for iOS, Android, Windows Phone, Blackberry, and some other platforms.

PhoneGap applications may run slower comparing to the HTML-based applications running in the mobile Web browser, because there is yet another middle man - a Web View. In Android SDK the `WebView` control is used to embed HTML5 application into a native shell, and the iOS SDK has the `UIWebView` control for the same purpose. Both of these controls perform slower than respective mobile Web browsers.



To compare performance of an application that runs in a mobile browser vs `WebView` or `UIWebView` control use [Google's V8 Benchmark Suite](#) or [SunSpider benchmark utility](#).

The UI components of the HTML5 framework of your choice may not look native enough. But the main selling point is that PhoneGap (and Cordova) allows you to leverage existing HTML/JavaScript developers' skills for all major mobile platforms, and their bridge to native APIs is easy to learn.

## Titanium

Titanium offers its own set of tools and more extensive API. It has no relations to Cordova or PhoneGap. You'd be writing code in JavaScript (no HTML or CSS) and would need to learn lots of APIs. The compiled and deployed application is a JavaScript code embedded inside Java or Objective-C code plus the JavaScript interpreter plus the platform-specific Titanium API. An important difference between Phonegap and Titanium is that the latter doesn't use any Web view container for rendering. The business logic written in JavaScript is executed by the embedded interpreter, the final UI components are delivered by native to iOS or Android components from Titanium.

Titanium UI components can be extended to use native OS interface abilities to their fullest. Some components are cross-platform - Titanium has a compatibility layer, while others are platform-specific. But if you want to learn platform-specific components, you might rather invest time in learning to develop the entire application in the native language and APIs. Besides, as new platform are introduced, you'll depend on the willingness of the Titanium developers to create a new set of components in a timely fashion.

Don't not expect the top performance from the old Rhino JavaScript engine, which is used by Titanium for Android and Blackberry applications. Oracle has a new JavaScript

engine called Nashorn, but it's available only for Java 8, which doesn't run on Android, and won't run there in the foreseeable future. Nashorn is as fast as [Google's V8](#), but Rhino is slower. Does it mean that Titanium applications on Android and Blackberry will always run slower? This seems to be the case unless Oracle and Google will find a way to stop their quarrels around Java.

The learning curve of the Titanium API is steeper (they have over 5000 APIs) than with PhoneGap. At the time of this writing, Titanium supports iOS, Android, and older versions of Blackberry devices. They plan to support Windows phones by the end of 2014.



PhoneGap and Titanium are not the only solutions that allow building hybrid applications using HTML5. The framework [Kendo UI Mobile](#) can build hybrid applications for iOS, Android, Blackberry, and Window Phone 8. The [Mobile Conduit API](#) allows to build cross-platform mobile application with HTML5. [Convertigo Mobilizer](#) is a cross-platform enterprise mashup environment that incorporates PhoneGap and Sencha Touch for building mobile applications. [IBM Woklight](#) offers to enterprises a client/server/cloud to enterprises develop, test, run and manage HTML5, hybrid and native mobile applications.

## The Bottom Line

If a particular enterprise application is intended only for the internal use by people carrying a limited variety of mobile devices, and if making business users productive is your main goal - consider developing native applications, which may be fine-tuned to look and feel as best as a selected platform allows. You can start with developing and deploying the first application for the pilot mobile OS (typically for the latest iOS or Android OS), and then gradually add support for more platforms, budget permitting. If you are planning to develop a Web application with a relatively simple UI and have to support a wide variety of unknown consumer devices (e.g. you want to enable people to donate from any device) - develop an HTML5 Web application.

Consider developing a hybrid application for anything in between, and in this chapter we'll show you how to access the camera of the mobile device with [PhoneGap](#) framework. Such functionality can be pretty useful for our Save The Child application as kids who received donations may want to share their success stories and publish their photos after being cured.

Still, remain open-minded about in native vs hybrid discussions. Be prepared that going hybrid may not become your final choice. Picking a platform is a complex, business-specific decision that might change over the life of your app.

# Intro to the PhoneGap Workflows

In this section you'll go through the entire process of building a PhoneGap application. PhoneGap 3.1 offers two major workflows. Each of them allows you to build a mobile application, but the main difference is where you build it - either locally or remotely. Here they are:

1. Install all required mobile SDKs and tools for the mobile platforms you want to develop for (e.g. iOS and Android), generate the initial project using the Command Line Interface (CLI), write your HTML5 application code, build it locally, and test the application using IDE, simulators and physical devices.
2. Don't install any mobile SDK and tools. Just generate the initial project using CLI, add the application code, zip up the *www* folder and upload it to **Adobe PhoneGap Build** server, which will build the application for all supported mobile platforms. Then download and test the application on physical devices.

The second workflow requires running a trivial install of PhoneGap and then just let the Adobe's Build PhoneGap server do the build for various mobile platforms. The first workflow is more involved, and we'll illustrate it by showing how to use the local SDKs for iOS deployment.



For some platforms PhoneGap supports only local builds (e.g. BlackBerry 10, Windows Phone 8), while builds for WebOS and Symbian can only be done remotely.

In any case you'll need to install the PhoneGap software according to the instructions from [the command-line interface](#) documentation. Start with installing Node.js, which will also install its package manager *npm* used for installing Cordova (and PhoneGap library). We're developing on MAC OS X, and here's the command that will install PhoneGap:

```
sudo npm install -g phonegap
```

The above command installs the JavaScript file *phonegap* in */usr/local/bin* and the Cordova library with supporting files in the */usr/local/lib/node\_modules/phonegap* - Figure [Figure 13-2](#) shows the snapshot of some of the files and directories that come with PhoneGap. We've highlighted the *create.js* script, which will be used for generating Hello World and Save The Child projects.

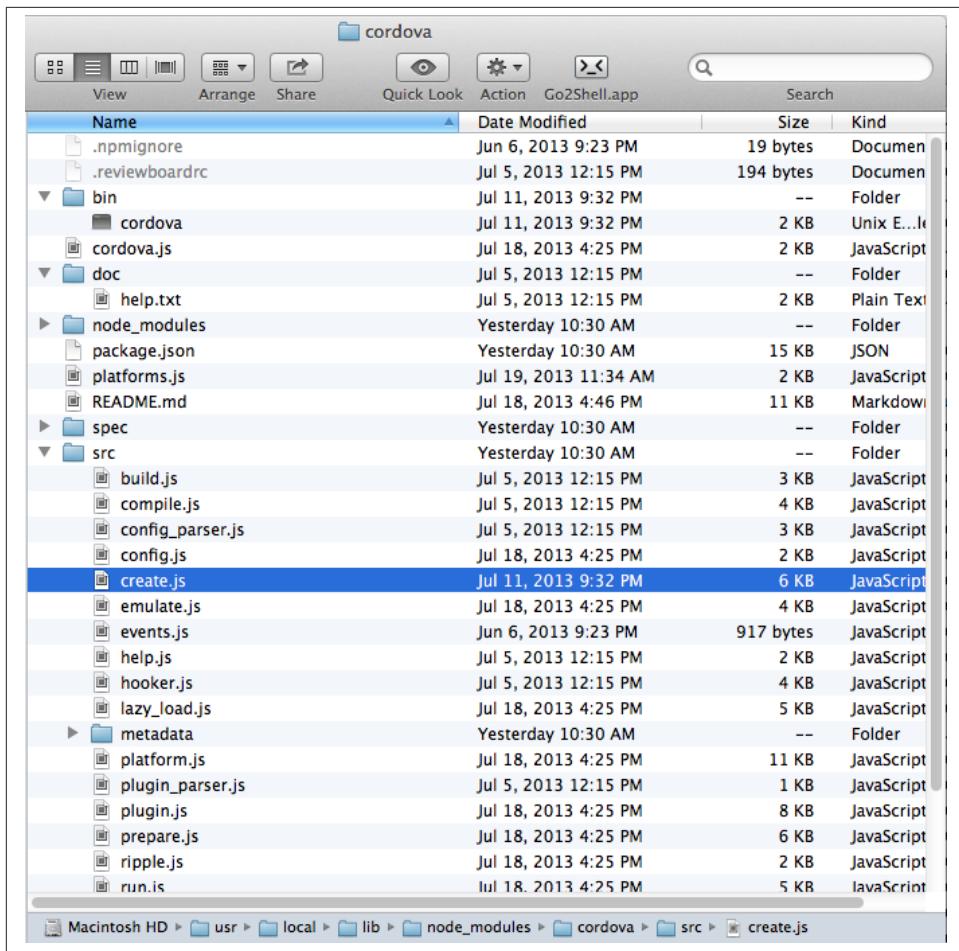


Figure 13-2. PhoneGap 3.1 Installed

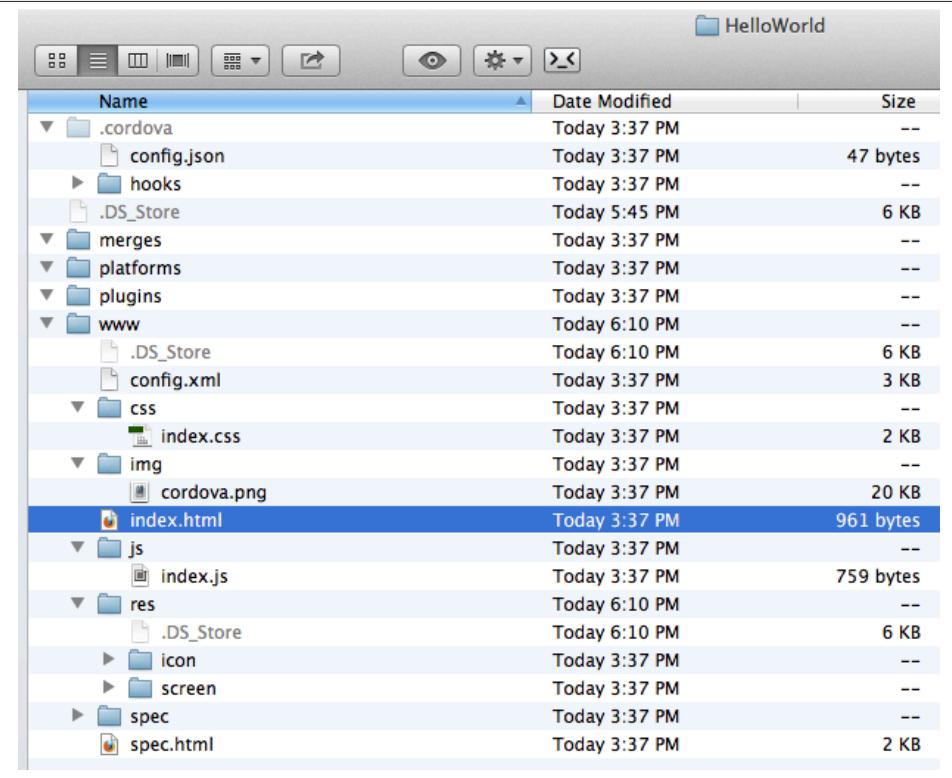
In this chapter we'll be developing a sample application for the iOS platform to illustrate the most involved deployment-deployment cycle. It requires [Xcode IDE](#), which is available at Apple's App Store at no charge. After installing Xcode open its menu Preferences and install Command Line Tools (CLT) from the Downloads panel. By default, Xcode comes with the latest iOS simulator (it's version 6.1 at the time of this writing).

## One More Hello World

The time has come for a PhoneGap version of Hello World. We are going to generate the initial project using CLI as described in the same [document](#) we used for installing PhoneGap in the section titled "Create the App". We'll be running the `phonegap` script:

```
phonegap create HelloWorld com.example.hello "Hello World"
```

After generating the Hello World code with the *phonegap create* command (might need to run it a superuser with *sudo*), you'll see the files and directories as on Figure Figure 13-3.



The screenshot shows a Mac OS X Finder window titled "HelloWorld". The contents are listed in a table:

Name	Date Modified	Size
.cordova	Today 3:37 PM	--
config.json	Today 3:37 PM	47 bytes
hooks	Today 3:37 PM	--
.DS_Store	Today 5:45 PM	6 KB
merges	Today 3:37 PM	--
platforms	Today 3:37 PM	--
plugins	Today 3:37 PM	--
www	Today 6:10 PM	--
.DS_Store	Today 6:10 PM	6 KB
config.xml	Today 3:37 PM	3 KB
css	Today 3:37 PM	--
index.css	Today 3:37 PM	2 KB
img	Today 3:37 PM	--
cordova.png	Today 3:37 PM	20 KB
index.html	Today 3:37 PM	961 bytes
js	Today 3:37 PM	--
index.js	Today 3:37 PM	759 bytes
res	Today 6:10 PM	--
.DS_Store	Today 6:10 PM	6 KB
icon	Today 3:37 PM	--
screen	Today 3:37 PM	--
spec	Today 3:37 PM	--
spec.html	Today 3:37 PM	2 KB

Figure 13-3. CLI-generated project Hello World



While using the above command *phonegap create HelloWorld com.example.hello "Hello World"*, keep in mind that in case of iOS you'll need to create a certificate, which has to be valid for applications packages located under *com.example*. For more details see the sidebar "Testing Application on iOS Devices" later in this chapter.

The content of the generated index.html is shown next. It includes several meta tags instructing the browser to use the entire screen of the mobile device without allowing scaling with user's gestures. Then it includes a couple of JavaScript files in the '*<script>*' tags.

```

<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <meta name = "format-detection" content = "telephone=no"/>
    <meta name="viewport" content="user-scalable=no, initial-scale=1, maximum-scale=1, minimum-scale=1" />
    <link rel="stylesheet" type="text/css" href="css/index.css" />
    <title>Hello World</title>
  </head>
  <body>
    <div class="app">
      <h1>Apache Cordova</h1>
      <div id="deviceready">
        <p class="status pending blink">Connecting to Device</p>
        <p class="status complete blink hide">Device is Ready</p>
      </div>
    </div>
    <script type="text/javascript" src="phonegap.js"></script>
    <script type="text/javascript" src="js/index.js"></script>
    <script type="text/javascript">
      app.initialize();
    </script>
  </body>
</html>

```

The screenshot of the running Hello World application is shown on Figure [Figure 13-5](#).

This HTML file includes the code to load the phonegap.js library and the initialization code from index.js. Then it calls `app.initialize()`. But if you look at [Figure 13-3](#) the file phonegap.js is missing. The CLI tool will add it to the project during the next phase of code generation when you'll run the command `phonegap platform add` to add specific mobile platforms to your project. Let's look at the code of the index.js.

```

var app = {
  initialize: function() { // ①
    this.bind();
  },
  bind: function() {
    document.addEventListener('deviceready', // ②
      this.deviceready, false);
  },
  deviceready: function() {
    app.report('deviceready');
  },
  report: function(id) { // ③
    console.log("report:" + id);
  }
};

```

```
document.querySelector('#' + id + '.pending').className += ' hide';
var completeElem = document.querySelector('#' + id + '.complete');
completeElem.className = completeElem.className.split('hide').join('');
}
};
```

- ❶ This function is being called when all scripts are loaded in index.html.
- ❷ The mobile OS sends the `deviceready` event to the PhoneGap application when it's ready to invoke native APIs.
- ❸ The function `report()` is called from the `deviceready` event handler. It hides the text `.pending` `<p>` and shows the `.complete` `<p>` in index.html. Technically, `split('hide')` followed by `join('')` performs the removal of the word *hide*.

It wouldn't be too difficult to prepare such simple HTML and JavaScript files manually, but we prefer using code generators - they are faster and less error prone.



Neither Cordova nor PhoneGap restrict you from using any HTML5 frameworks of your choice.

## Prerequisites for Local Builds

If you are planning to build your application locally, install the supporting files for the required platforms. For example, you can run the following commands from the command window (switch to the `HelloWorld` directory) to request the builds for iOS, Android, and Blackberry:

`phonegap install ios`

`phonegap install android`



The first command will run fine, because we have Xcode installed. The second command will fail until you install the latest Android SDK as described in the sidebar.

After running the above commands, the initially empty directory `platforms` will be filled with additional sub-directories specific to each platform. Technically, these commands generate separate Hello World projects - one per platform. Each of them will have its own `www` directory with `index.html` and `phonegap.js` that was missing during the initial project generation. Don't make any modifications in these `www` folders as they will be

regenerated each time when the *install* or *run* command are run. Make the required modification in the root *www* folder.

You can see on Figure [Figure 13-4](#) the content of the *ios* folder that was generated as a result of executing command *phonegap install ios*.

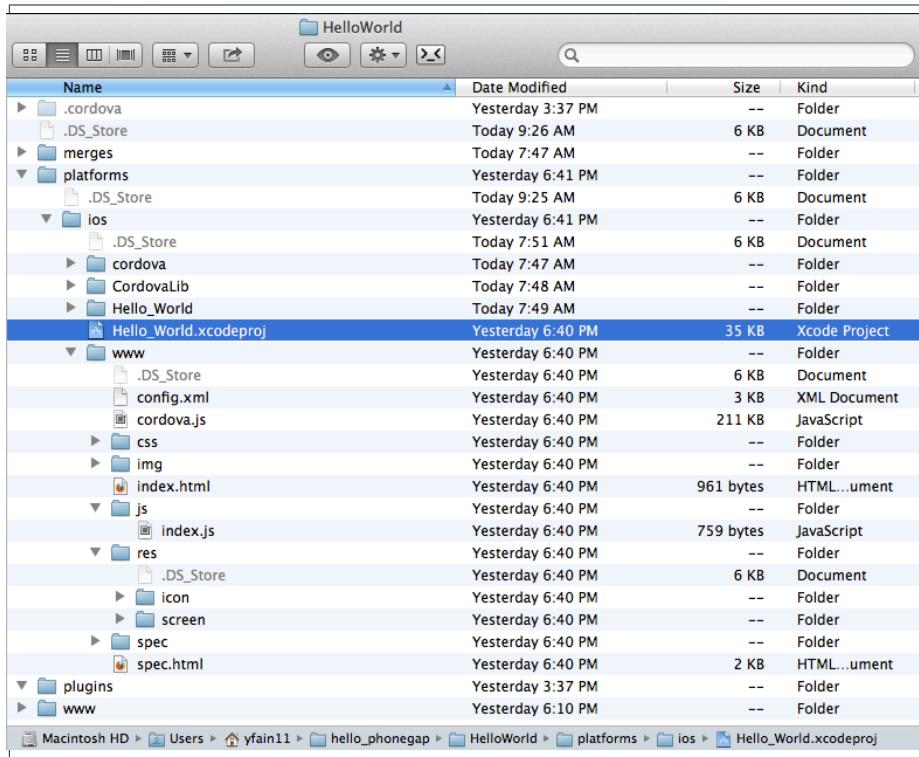


Figure 13-4. CLI-generated project for iOS platform

Double-click on the file *Hello\_World.xcodeproj*, and Xcode will open it as a project. Press the button Run on the top left corner of the toolbar to compile the project and start in the iOS simulator (see Figure [Figure 13-5](#)). Note the “Device is ready” text from *index.html* (as per *index.css* this text is blinking and is shown in the upper case).

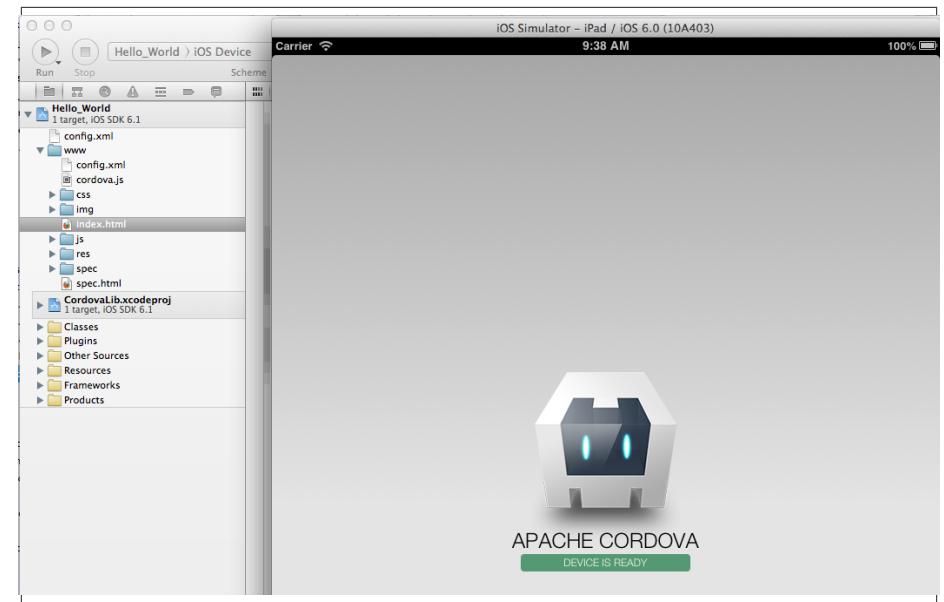


Figure 13-5. Running Hello World in XCode

The description of the workflow with the Build Phonegap server will follow.

## Testing Applications on iOS Devices

If you want to test your application not in the simulator, but on the physical iOS device, it has to be connected to your Mac computer, enabled for deployment and recognized by Apple. Details on *provisioning your devices for development* are described in the online [iOS Developer Library](#). If you prefer shorter instructions, here's what worked for us:

1. Open a Keychain Access application on your Mac computer and create a certificate request using the menus Keychain Access | Certificate Assistant | Request a Certificate from Certificate Authority. This will create a file with the name extension .certSigningRequest.
2. Log on to Member Center at [developer.apple.com](http://developer.apple.com) and create a certificate in there for iOS Development specifying the wildcard (an asterisk) in the Bulk name unless you want to restrict this certificate to be used only with application that start with a certain prefix. During this step you'll need to upload the .certSigningRequest file created in the previous step.

3. After this certificate is created, download this file (its name ends with .cer), and double-click on it to open in your local keychain. Find it in the list of certificates and expand it - it should include the private key.
4. Remain in the Member Center, and create a unique application ID.
5. Finally, in the same Member Center create a Provisioning Profile.
6. In Xcode, open the menu Window | Organizer, go to Provisioning profiles window, and refresh it. You should see the newly created provisioning profile marked with a green bullet. A physical file with the name extension .mobileprovision correspond to this profile.
7. Select your iOS device in the active scheme dropdown on top left and run your Hello World or other project on the connected device.



Read Apple's [App Distribution Guide](#) to learn how to distribute your iOS applications.

## Installing More Local SDKs

As we stated earlier, you don't have to install SDK's locally, but if you decided to do so, consult with instructions by the respective mobile platform vendor. For example, Blackberry developers can download their WebWorks SDK at [developer.blackberry.com/html5/download](http://developer.blackberry.com/html5/download) as well as Blackberry 10 Simulator. If you haven't downloaded the Ripple Emulator (see Chapter 12) you can get it there too.

Instructions for installing the Windows Phone SDK are available at the [Windows Phone Dev Center](#).

To get Android SDK, go to [android.com/sdk](http://android.com/sdk). We are going to do a simple install by pressing the button "Download the SDK ADT Bundle for Mac", which will download and install Eclipse IDE with ADT plugin, Android SDK tools, Android Platform tools, and Android platform. But if you already have Eclipse IDE and prefer to install and configure required tooling manually, follow the instructions published on this Web site under the section "Use an Existing IDE".

After downloading the bundle, unzip this file, and it'll create a folder with two subfolders:  *sdk* and *eclipse*. Start Eclipse IDE from *eclipse* folder accepting the location of the default workspace. Press the little plus-sign on the top toolbar and open perspective DDMS. There you can use Android emulator while developing Android applications.

# Using Adobe PhoneGap Build Service

Instead of installing multiple SDKs for different platforms you can use the cloud service **Adobe PhoneGap Build**, which already has installed and configured all supported SDK's and will do a build of your application for different platforms. For our example we're going to use iOS build.

Visit [build.phonegap.com](http://build.phonegap.com) and sign in with your Adobe or GitHub ID. If your project resides on GitHub, copy its URL to the text field shown on Figure Figure 13-6. The other way to do a build is to compress your project's *www* directory and upload this zip file there.



Starting from PhoneGap 3.0 all code modifications are done in the main *www* folder of your project. During local rebuilds all the changes get automatically replicated to each installed platform's *www* folder.

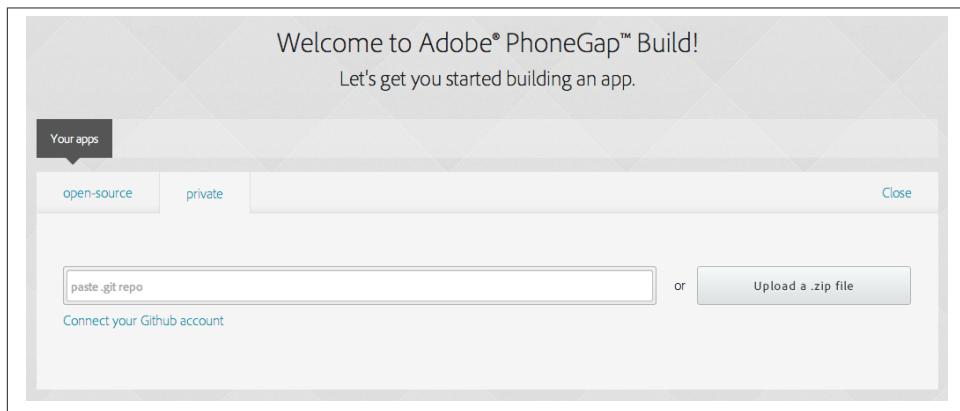


Figure 13-6. Submitting Application to PhoneGap Build Server

Before zipping up the Hello World's *www* directory, open and modify the file *config.xml*. The generated XML contains entries for every platform. Since we are doing a build for iOS, we'll remove all the lines that contain the words *android* or *blackberry*.

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="com.example.hello" version="2.0.0"
    xmlns="http://www.w3.org/ns/widgets"
    xmlns:cdv="http://cordova.apache.org/ns/1.0">

    <name>Hello World</name>

    <description>
```

```

A sample Apache Cordova application that responds to the deviceready event.
</description>

<author email="callback-dev@incubator.apache.org" href="http://cordova.io">
    Apache Cordova Team
</author>

<icon height="512" src="res/icon/cordova_512.png" width="512" />
<icon cdv:platform="ios" height="144" src="res/icon/cordova_ios_144.png" width="144" />
<cdv:splash cdv:platform="ios" height="748" src="res/screen/ipad_landscape.png" width="1024" />
<cdv:splash cdv:platform="ios" height="1004" src="res/screen/ipad_portrait.png" width="768" />
<cdv:splash cdv:platform="ios" height="1496" src="res/screen/ipad_retina_landscape.png" width="1024" />
<cdv:splash cdv:platform="ios" height="2088" src="res/screen/ipad_retina_portrait.png" width="768" />
<cdv:splash cdv:platform="ios" height="320" src="res/screen/iphone_landscape.png" width="480" />
<cdv:splash cdv:platform="ios" height="480" src="res/screen/iphone_portrait.png" width="320" />
<cdv:splash cdv:platform="ios" height="640" src="res/screen/iphone_retina_landscape.png" width="480" />
<cdv:splash cdv:platform="ios" height="960" src="res/screen/iphone_retina_portrait.png" width="320" />

<feature name="http://api.phonegap.com/1.0/device" />

<preference name="phonegap-version" value="3.1.0" />
<access origin="*" />
</widget>
```

Specify the latest *supported* PhoneGap version in the “phonegap-version” attribute. The online document [Using config.xml](#) has the current information about supported versions and other essential properties. We’ll change the phonegap-version value to 3.1.0, which was the latest supported by PhoneGap Build version at the time of this writing. You’ll see some other entries in config.xml of the Save The Child application.

Now select all the content inside the *www* folder and compress it into the zip file named *helloworld-build.zip*. Open the Web browser, go to [build.phonegap.com](#), press the button labeled “Upload a .zip file”, and select your local file *helloworld-build.zip*. When uploading is done, you’ll see the next screen shown at [Figure 13-7](#).

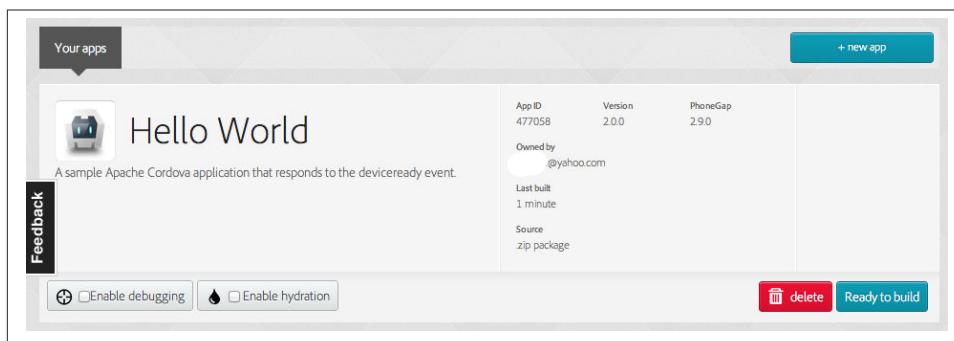


Figure 13-7. After helloworld-build.zip was uploaded

Click on the button “Ready to Build”, to start the build for all available platforms. In you did everything right, after watching the wait cursor above each icon, all the builds will successfully complete, and you’ll see a blue line under each button. Figure [Figure 13-8](#) illustrates the case when the build failed for iOS and BlackBerry platforms (the first and fourth buttons are underlined with in red).



You can create remote builds on with Adobe PhoneGap Build service from the command line too (*phonegap remote build*). Read the section “Build Applications Remotely” in the [PhoneGap CLI Guide](#).

Fixing the Blackberry version of the application is not on our agenda. Refer to the [Platform Guides](#) documentation that contains specific information on what has to be done to develop and deploy PhoneGap applications for each platform. We’ll just take care of the iOS issue.

The screenshot shows the Adobe PhoneGap Build service interface. At the top, there's a navigation bar with 'Your apps' and a '+ new app' button. Below it, the 'Hello World' app is listed. The app details include: App ID: 477058, Version: 2.0.0, PhoneGap: 2.9.0, Owned by: yfain11@yahoo.com, Last built: 38 minutes ago, Source: zip package, and a QR code. Below the details, there are icons for different platforms: iOS (red underline), Android, Windows, BlackBerry (red underline), HP, and i. At the bottom, there are buttons for 'Update code' and 'Rebuild all'.

*Figure 13-8. Two builds failed*

After clicking on the iOS button, it revealed the message in a dropdown box “No key selected”. Another error message reads “You must provide the signing key first”. The dropdown also offers an option to add the missing key. Selecting this option reveals a panel shown on Figure [Figure 13-9](#).

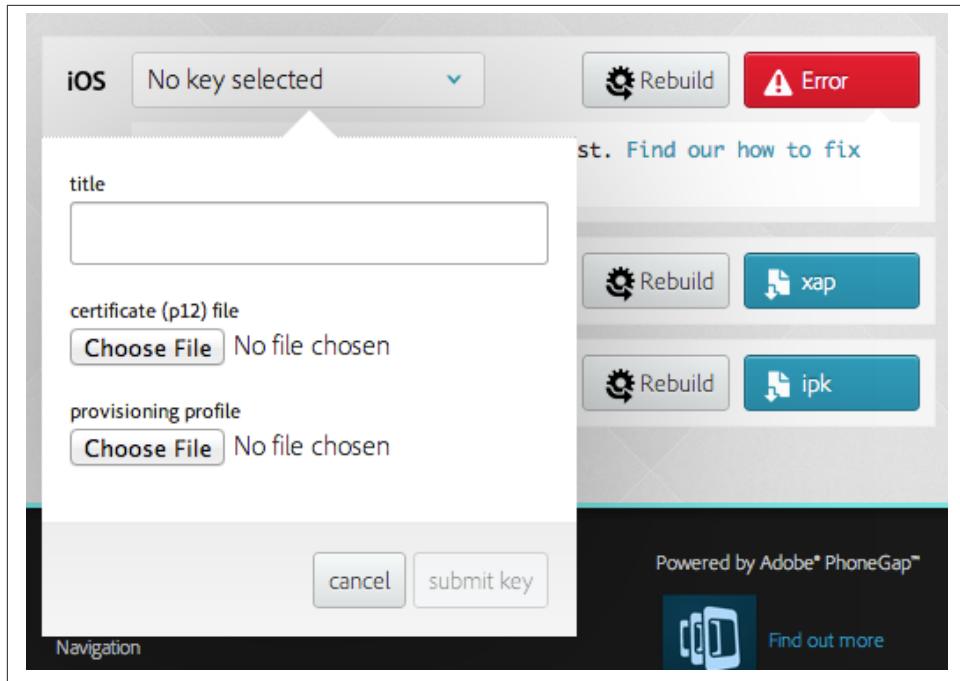


Figure 13-9. Uploading certificate and profile

The missing key message actually means that they need the provisioning profile and the certificate discussed in the section “Testing Applications on iOS Devices”. The certificate has to be in the P12 format, and you can export it into the .p12 file from the Keychain Access program under MAC OS X. During the export, you’ll assign a password to the certificate that will be required by the PhoneGap Build process. After uploading the .p12 and .mobileprovision files to PhoneGap Build and unlocking the little yellow lock, rebuild the Hello World for iOS and it should run without any errors.



If you forgot where the .mobileprovision file is located, open Xcode and go to the menu Window | Organize, open the panel Provisioning Profiles under Library, right-click on the profile record and select Reveal in Finder.

To complete the process, deploy the application on your mobile device, which can be done by one of the following methods:

1. Use the **QR Code** that was generated specifically for our application - it’s shown on the right side of Figure [Figure 13-8](#). Just install a QR Reader program on your device, scan this code and the Hello World application will be installed on your device.

2. Download the application file from build.phonegap.com to your computer and then copy it onto the mobile device. For example, to get the Android version of the Hello World, just click on the button with Android's logo and the file HelloWorld-debug.apk will be downloaded to your computer. Copy this file to your Android device and enjoy the application. For the iOS version, click on the button with the iOS logo, which will download the file HelloWorld.ipa on your Mac computer. Double click on this file in Finder, and it'll bring it into the Application section of iTunes. Synchronize the the content of iTunes with your iOS device, and Hello World will be installed there.



Using the PhoneGap Build service is free as long as you're building public applications, which have their source code hosted on a publicly accessible repository on Github or other hosting service. Our Hello World application is considered to be private because we submitted it to PhoneGap Build in a zip file (note the *private* tab in Figure [Figure 13-6](#)). Only one private application at a time can be built with PhoneGap Build for free. For building multiple private applications you'd need to purchase an inexpensive subscription from Adobe. To replace one application with another, click on its name, then press the buttons Settings and then delete this App.

Phew! This was the longest description of developing and deploying the Hello World application that we've ever written! We picked the deployment on the Apple's devices, which this the most complicated process among all mobile platforms. And we didn't even cover the process of submission the application in the App Store (you'll read more about it in the next section)! But developing and deploying an application that have to run natively on multiple platforms is expected to be more complicated than deploying an HTML5 application in a Web browser.



The HelloWorld application does not use any API to access the hardware of the mobile device, and it doesn't have to. PhoneGap build can be used simply to package any HTML5 app as a native one to be submitted to an app store.



Instead of using the JavaScript function `alert()`, display messages using `navigator.notification.alert()` and PhoneGap will show them using the native message box of the device. The `Notification` object also supports `confirm()`, `beep()`, and `vibrate()` methods.

## Distribution of Mobile Applications

Mobile device manufacturers set their own rules for application distribution. Apple has the most strict rules for the iOS developers.

Apple runs the [iOS Developer Program](#), and if you're an individual who wants to distribute iOS applications via the App Store, it'll cost you \$99 per year. Higher education institutions that teach iOS development can be enrolled into this program for free. The iOS Developer Enterprise program costs \$299 a year. To learn the differences between these programs and visit Apple's Web page <https://developer.apple.com/programs/which-program/>.

Besides being able to deploy the application in the App Store, developers can allow their beta-customers to test the application even before they were accepted in the App Store. Individual developers can share their application among up to 100 iOS devices identified by UUID (click on the serial number of your device in iTunes to see it). This is so-called "Ad Hoc distribution".

For example, after the PhoneGap Build service has built the .ipa file for iOS, you can make it available for installation right on the beta-tester's device using such services as [diawi](#) or [TestFlight](#). Upload the .ipa file and its provisioning profile to one of these services and you'll get the link (a URL) to be given to your testers - the UUID of their devices must be registered with your developer's profile. To do this, login to your account at [developer.apple.com](https://developer.apple.com), select the section "Certificates, Identifiers & Profiles", then go to Devices and add the UUID of the iOS device to the existing list of registered devices.

The owners of the enterprise license can distribute their applications right from their own Web sites.

Figure [Figure 13-10](#) shows the snapshot from the iPhone after the tester clicked on such a link from diawi. Pressing the button Install Application completes the install of the application on your iOS device.

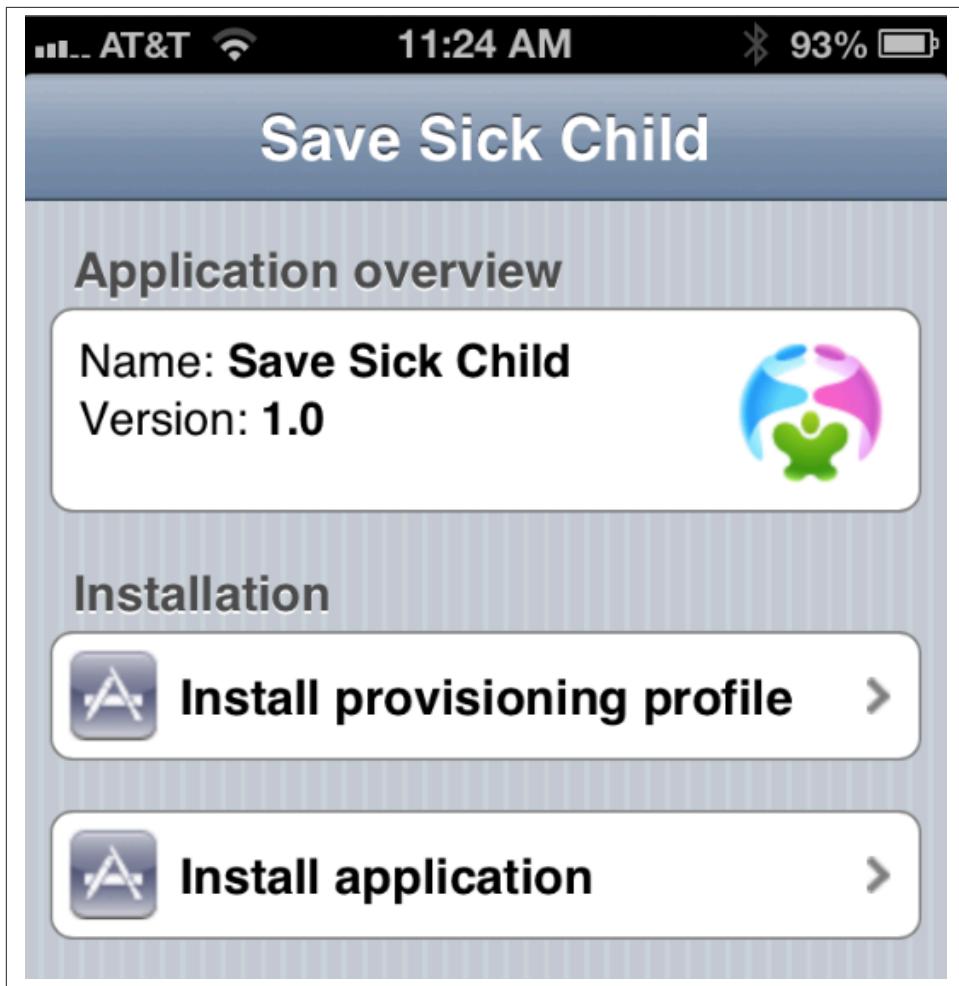


Figure 13-10. Ad hoc application install from diawi

Android developers are not restricted in distributing of their application - upload the application's APK package to your corporate Web site and send the URL to anyone who's interested. For example, the authors of this book are creating a software for insurance industry, where they offer to download both iOS and Android versions of the application right from their corporate Web site as shown at Figure [Figure 13-11](#).

The screenshot shows the 'SureLC Mobile Downloads' section of the surancebay.com website. At the top, the SuraceBay logo is visible along with a navigation bar containing links for Home, Solutions, Clients, Testimonials, Demos, Support, About, and Contact us. Below the navigation, the title 'SureLC Mobile Downloads' is centered above two download links. The left link, 'Install SureLC for iOS', features the SuraceBay logo next to an Apple icon. The right link, 'Install SureLC for Android', features the SuraceBay logo next to an Android icon. To the right of these links is a sidebar with several sections: 'AGENTS' (with a cloud icon), 'GENERAL AGENCIES' (with a person icon), 'CARRIERS' (with a briefcase icon), 'CONSUMER SOLUTIONS' (with a gear icon), 'CONSULTING SERVICES' (with a person icon), and a 'Our Blog' link at the bottom.

Figure 13-11. Distributing mobile applications at surancebay.com

While simulators and emulators can be very handy, nothing is better than testing on the real devices. There are several models of iPhones that vary by the CPU power and screen resolution. Ensuring that the application performs well on Android devices is a lot more challenging - this market is really fragmented in both hardware and OS use. Android emulators are not as good as the iOS ones. On the other hand, iOS emulator won't allow you to test the integration with the camera. Such features of the real devices like accelerometer or gyroscope simply can't be tested with emulators. The **PhoneGap emulator** is based on Ripple add-on (see Chapter 11) and allows to subscribe to the `deviceready` event and stub responses for your custom plugins.



You can use **TestFlight** as a way to test, distribute apps, and manage provisioning profiles for iOS. **HockeyApp** HockeyApp is a platform to collect live crash reports, get feedback from your users, distribute your betas, recruit new testers, and analyze your test coverage.

If you've architected your hybrid application in a modularized fashion as described in Chapter 6, you'll get an additional benefit. If the code of one of the loadable modules changes, but the main application shell remains the same, there is no need to resubmit the new version of the application to the App Store or another marketplace. This can be a serious time saver, especially on Apple devices - you eliminate the approval process of each new version of the application.

# Save The Child with PhoneGap

To demonstrate how to turn a Web application into a hybrid one, we'll take the code of the jQuery Mobile version of the Save The Child application from Chapter 12. Initially, we'll just turn it into a hybrid PhoneGap application as is without adding any native API calls. After that we'll add to it the ability to work with the photo camera using PhoneGap API and create two builds for iOS and Android platforms. In this exercise we'll use PhoneGap 3.3.



Usually, PhoneGap is mentioned in the context of building hybrid applications that need to access some native API. But PhoneGap can be used for packaging any HTML5 application as a native one even if it doesn't use native API.

## How to Package Any HTML5 App With PhoneGap

Let's go through the process of building and deploying the jQuery Mobile version of Save The Child in its existing form without changing even one line of code. Here's the step by step procedure:

1. Generate a new PhoneGap project using PhoneGap CLI as we did with Hello World. This time we won't add any specific mobile SDKs to the project though.
2. Copy the existing HTML, CSS, JavaScript and other resources from the jQuery Mobile Save The Child application into the directory *www* of the newly generated PhoneGap project.
3. Create platforms where we're planning to deploy our application:

```
$ sudo phonegap build ios $ sudo phonegap build android
```

1. Install PhoneGap plugins listed below that are necessary for supporting such functionality as Splashscreen, Camera, Inappbrowser, File, and File-transfer.

```
$ sudo phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-splashscreen.git $ sudo phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-camera.git $ sudo phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-inappbrowser.git $ sudo phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-file.git $ sudo phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-file-transfer.git
```

1. Test the Save The Child application on the Android, iOS or other mobile devices.



If you don't have some of the platforms SDK installed locally (as we did in Step 2 above), you can compress the entire content of the *www* directory into a ZIP file, upload it to PhoneGap Build server and generate the packages for several platforms there.

## Adding Camera Access to Save The Child

Charity Web sites help millions of people to get better. When this happens, they want to share their success stories, and maybe publish photos of themselves or their families and friends. These days everyone uses smartphones and tablets to take pictures, and adding the ability to access camera of the mobile device and uploading photos seems like a useful feature for our Save The Child application.

We'll add the camera access to the jQuery Mobile version of our application. The next code fragment is an extract from the file *app-main.js*.

Our next goal is to use PhoneGap to access the native API of the camera of the mobile device to take photos. After that, the user should be able to upload images to the server.

For starting the device's default camera application and taking photos, PhoneGap offers the function `navigator.camera.getPicture()`, which takes three arguments: the name of the function handler if the photo has been successfully taken, the handler for the error, and the object with the optional parameters describing the image. Details about the camera API are available in the [PhoneGap documentation](#).

```
var pictureSource;
var destinationType;
var uploadedImagesPage = "http://savesickchid.org/ssc-phonegap/uploaded-images.php";
var photo;

function capturePhoto() {

    navigator.camera.getPicture(
        onPhotoDataSuccess, onCapturePhotoFail,
        {
            quality : 49,
            destinationType: destinationType.FILE_URI
        });
}

function onCapturePhotoFail(message) {
    alert('Capture photo failed: ' + message);
}

function onPhotoDataSuccess(imageURL) {
    var smallImage = $('#smallImage');
    photo = imageURL;
    $('#photoUploader').css('display', 'block');
```

```

$( '#ssc-photo-app-description' ).css( 'display', 'none' );
smallImage.css('display', 'block');
smallImage.attr("src", imageURL);
$( '#largeImage' ).attr("src", imageURL);

$( '#uploadPhotoBtn' ).removeClass('ui-disabled');
$( '#done-msg-holder' ).css('display', 'none');

}

```

Depending on the options in the third argument of the `getPicture()`, the image will be returned as either base64-encoded String, or as in our case, the URI of the file where the image is saved. If the photo was taken successfully, the application will make the `#photoUploader` button visible.

This code sample uses `quality:49` for picture quality, which allows you to request the picture quality as a number on the scale of 1 to 100 (the larger number means better quality). Based on our experience, 49 gives reasonable quality/file size ratio. The taken image can be returned as either a base64 encoded string or as a file URI - we used the latter. For the current list of options refer to the [PhoneGap Camera API](#) documentation.



For illustration purposes the above code uses the JavaScript `alert()` function to report a failure. For a more robust solution consider creating some custom way of reporting errors like red borders, modal dialog boxes with images or status bars.

The above function `capturePhoto()` should be called when the user taps the button on the application's screen. Hence we need to register an event listener for this button. Below is a fragment of the `onDeviceReady` function that registers all required event listeners.

```

function onDeviceReady() {

    pictureSource = navigator.camera.PictureSourceType;
    destinationType = navigator.camera.DestinationType;

    $(document).on("pageshow", "#Photo-app",
        function() {

            $('#capturePhotoBtn').on('touchstart', function(e) {
                $(e.currentTarget).addClass('button-active');
            });

            $('#capturePhotoBtn').on('touchend', function(e) {
                $(e.currentTarget).removeClass('button-active');
                capturePhoto();
            });
        });
}

```

```

$( '#uploadPhotoBtn' ).on( 'touchstart', function(e) {
    $( e.currentTarget ).addClass( 'button-active' );
});

$( '#uploadPhotoBtn' ).on( 'touchend', function(e) {
    $( e.currentTarget ).removeClass( 'button-active' );
    uploadPhoto( photo );
});

$( '#viewGallerylBtn' ).on( 'touchend', function() {
    window.open( uploadedImagesPage, '_blank', 'location=no' );
});
}
);

```

If the user clicks on the Upload Photo button, we use the `FileTransfer` object to send the image to the server side script `upload.php` for further processing. The code to support file uploading on the client side is shown next:

```

function uploadPhoto( imageURI ) {

    var uploadOptions = new FileUploadOptions();
    uploadOptions.fileKey = "file";
    uploadOptions.fileName = imageURI.substr( imageURI.lastIndexOf( '/' ) + 1 );
    uploadOptions.mimeType = "image/jpeg";

    uploadOptions.chunkedMode = false;

    var fileTransfer = new FileTransfer();
    fileTransfer.upload( imageURI, "http://savesickchild.org/ssc-test/upload.php", onUploadSuccess );

    var uploadedPercentage = 0;
    var uploadedPercentageMsg = "Uploading...";

    fileTransfer.onprogress = function( progressEvent ) {
        if ( progressEvent.lengthComputable ) {
            uploadedPercentage = Math.floor( progressEvent.loaded / progressEvent.total );
            uploadedPercentageMsg = uploadedPercentage + "% uploaded...";
        } else {
            uploadedPercentageMsg = "Uploading...";
        }
        $.mobile.showPageLoadingMsg( "b", uploadedPercentageMsg );
    };
}

function onUploadSuccess( r ) {
    $.mobile.hidePageLoadingMsg();

    $('#done-msg-holder').css( 'display', 'block' );
    $('#uploadPhotoBtn').addClass( 'ui-disabled' );
}

```

```
function onUploadFail(error) {
    alert("An error has occurred: Code = " + error.code);
}
```

This sample code uses the PHP script located at the following URL: <http://savesickchild.org/ssc-test/upload.php>. You'll see this script in the next section. The "b" in the showPageLoadingMsg() function defines the **jQuery Mobile theme**. Figure 13-12 is a snapshot taken on iPhone while the Save The Child application was uploading a photo.

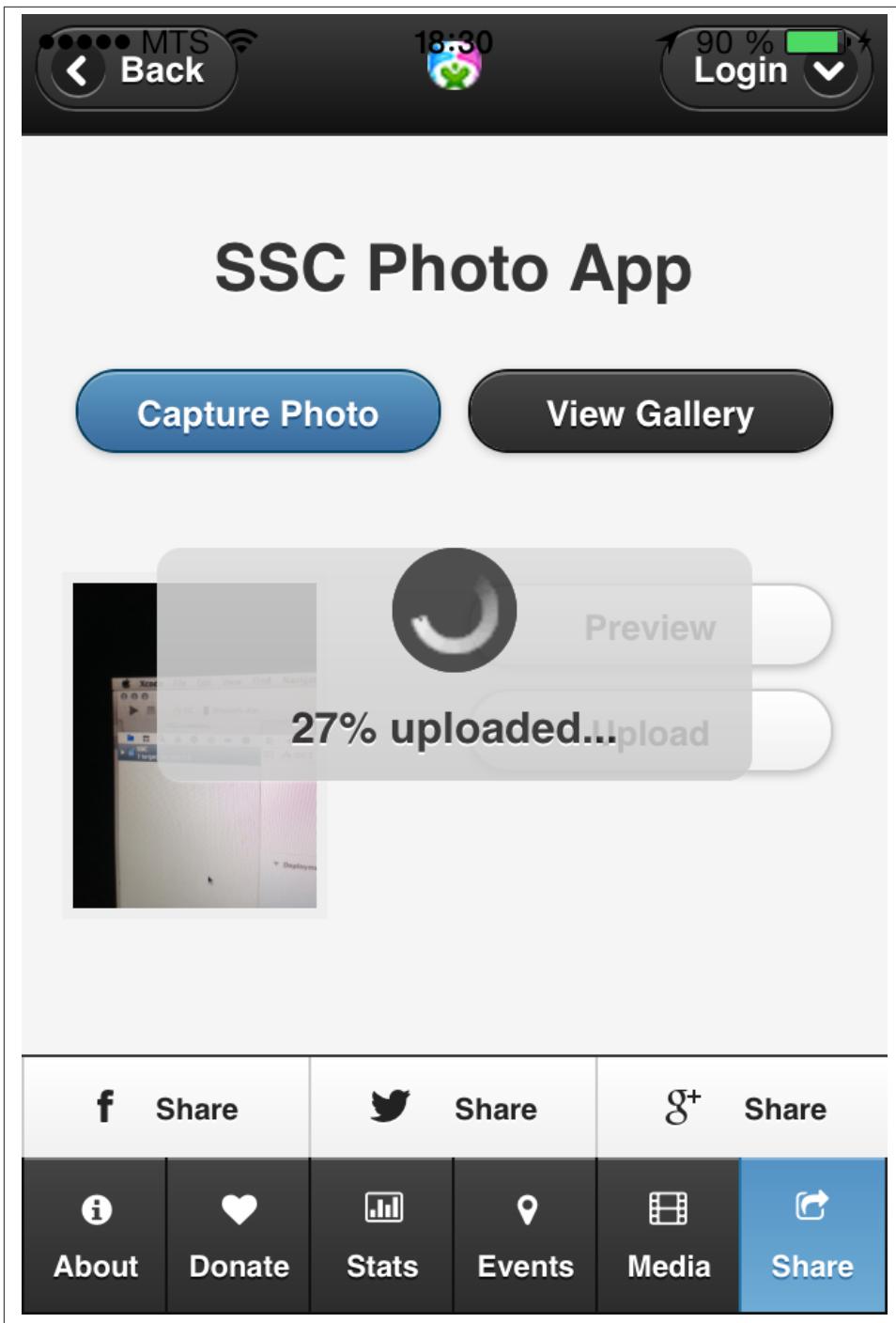


Figure 13-12. Uploading a Photo

## The Sever-Side Support for Photo Images

To support this application on the server side, we've created several PHP scripts. Of course, you can use the programming language of your choice instead of PHP.

The PHP script upload.php shown below uploads the image into a folder on the server and then creates two versions of this image: a thumbnail and optimized image. The thumbnail can be used for showing the image's preview in a grid. The optimized image file will have reduced dimensions for showing the image in the mobile browser. This script also moves and saves the thumbnail, optimal and original files in the corresponding folders on disk.

```
<?php

function resizeAndSave ($new_width, $new_height, $input, $output, $quality) {

    // Get new dimensions
    // assign variables as if they were an array
    list($width_orig, $height_orig) = getimagesize($input);
    $ratio_orig = $width_orig/$height_orig;

    if ($new_width/$new_height > $ratio_orig) {
        $new_width = $new_height*$ratio_orig;
    } else {
        $new_height = $new_width/$ratio_orig;
    }

    //using the GD library
    $original_image = imagecreatefromjpeg($input);

    // Resampling
    $image = imagecreatetruecolor($new_width, $new_height);
    imagecopyresampled($image, $original_image, 0, 0, 0, 0, $new_width, $new_height, $width_on);

    // Output
    imagejpeg($image, $output, $quality);
    imagedestroy($image);
}

$timestamp = time();
$image_name = $timestamp.'.jpg';
$path_to_original = 'upload/original/'.$image_name;

if(move_uploaded_file($_FILES["file"]["tmp_name"], $path_to_original)) {

    $thumb_width = 200;
    $thumb_height = 200;
    $thumb_output = 'upload/thumbs/'.$image_name;

    $optimum_width = 800;
    $optimum_height = 800;
```

```

$optimum_output = 'upload/optimum/'.$image_name;

$quality = 90;

resizeAndSave ($thumb_width, $thumb_height, $path_to_original, $thumb_output, $quality);
resizeAndSave ($optimum_width, $optimum_height, $path_to_original, $optimum_output, $quality
}

?>

```

The following script uploaded-images.php serves the web page with a list showing thumbnails of uploaded images.

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>SSC. Uploaded Images</title>
    <link rel="stylesheet" href="styles.css?<?php echo(time()); ?>">
</head>
<body>
    <ul>
        <?php
            $thumbs_dir = "upload/thumbs/";
            //get all image files with a .jpg and .png extension.
            $thumbs = glob($thumbs_dir."{*.jpg,*.*png}", GLOB_BRACE);

            foreach($thumbs as $thumb){
                $filename = basename($thumb);
                echo('<li><a href="show-img.php?p='.$filename.'"></a>');
            }
        ?>
    </ul>
</body>
</html>

```



During development, you might be often changing the CSS content. The `php echo(time());` in the above code is just a trick to prevent the Web browser from doing CSS caching during local tests. Newly generated time makes the CSS URL different on each load.

The script show-img.php shows an optimized single image in the user's browser window.

```

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,initial-scale=1">

```

```
<title>The Uploaded Image</title>
<link rel="stylesheet" href="styles.css?<?php echo(time()); ?>">
</head>
<body>
    <div id="wrapper"><?php $img=$_GET["p"]; echo(''); ?></div>
</body>
</html>
```

The complete source code of the PhoneGap version of the Save The Child with the camera support is available for download among other book's code samples.

## Summary

Hybrid applications allow you to take the HTML5-based Web application, connect it with the native API of the mobile device and package it as a native application. The selling point of using hybrids is that you can reuse the existing HTML5/JavaScript expertise. In the enterprise setup maintaining bugs in a one-language bug database is a lot more easier than if you had multiple versions of the application written in different languages. Maintaining a single set of images, videos, and CSS files is yet another advantage that lowers both time to market and cost of ownership of the application.

Thorough testing of hybrid applications is a must. With the BYOD policies even the enterprise applications must be tested on a variety of the mobile devices. The development manager and application owners have to agree on the list of mobile devices where your application will be deployed first. This has to be done in writing in the early stages of the project and be as detailed as possible. The statements like "The initial version of the application will run on iOS devices" is not good enough, because the difference between iPhone 3GS and iPhone 5 is huge. The former has 256MB of RAM, 600 Mhz CPU, and 480x320 pixels screen, while the latter champions 1GB of RAM, 3-core A6 CPU at 1.3Ghz, and 1135x640 pixels display.

Hybrid applications not only give the developers and users access to the native capabilities of the mobile devices, but allow distribute your HTML5 application through multiple App Stores or market places offered by device manufacturers.

Enterprise managers are always concerned with the availability of paid technical support. A substantial part of this chapter was about using PhoneGap, and Adobe offers **various support packages** for purchase.

Make no mistakes though - if you want to create the fastest possible application that looks exactly like other applications on the selected mobile platform, develop it in the native language prescribed by the device manufacturer. Faster applications take less CPU power, which translates to a longer battery life. If you can't hire experts in each mobile OS going hybrid can be a practical compromise.



---

## CHAPTER 14

# Epilogue

Even though this book is about HTML5, the authors would rather work with compiled languages that produce applications running in virtual machines. Such software platforms are more productive for development and more predictable for deployment.

## HTML5 is not a Rosy Place

While writing this book, we were often arguing about pros and cons of switching to HTML5, and so far we are concerned that the HTML/JavaScript/CSS platform is not overly productive for developing enterprise applications just yet. We live in the era when amateurs feel comfortable creating Web sites and HTML with a little JavaScript inserts provide the flexibility and customization the Microsoft Access and Excel provided in the good old PC times.

Till this day Microsoft Excel remains the most popular application among business users in the enterprises. They start Excel locally, it has a local storage that enables work in the occasionally-connected scenarios. Both the data and the code are physically located close to the user's heart. Microsoft Excel allows the users to have her own little pieces of data and amateurish-but-working-code (a.k.a. formulas) very close and personal. Right on the desktop. Fine print: until their computer crashes due to viruses or other problems. No need to ask these IT prima donnas for programming favors. Business users prefer not being dependent on the connectivity or some mysterious servers being slow or down. The most advanced business users even learn how to operate MS Access database to further lessen their dependency from the IT labor force.

But there is only so much you can do with primitive tools. Visual Basic was “JavaScript” of the nineties - it had similar problems, but nevertheless had huge followings. Now the same people are doing JavaScript. If we don't break this cycle by adopting a VM common to all browsers, we are doomed to go through generation after generation of under-powered crap.

Recently, one of our clients from Wall Street sent us a list of issues to be fixed in an Web application that we were developing using Adobe Flex framework (these applications run inside Flash Player virtual machine). One of the requested fixes was “remove a random blink while a widget moves in the window and snaps to another one”. We’ve fixed it. What if that fix had to be implemented in HTML5 and tested in a dozen of Web browsers? Dealing with a single VM is easier.

You may argue that browser’s plugin Flash Player (as well as Silverlight or browser’s Java runtime) is going away, it was crashing the browsers and had security holes. But the bar for the UI of Flash-based enterprise applications is set pretty high. Business users will ask for features or fixes they are accustomed to in the desktop or VM-based applications. We hope that future enterprise Web applications developed with supporting future HTML 6 or 7 specifications will be able to accommodate the user’s expectations in the UI area. The time will come when HTML widgets won’t blink in any of the major browsers.

We wrote this book to help people with understanding of what HTML5 applications are about. But make no mistakes - the world of HTML5 is not a peachy place in the future preached by educated and compassionate scientists, but rather a nasty past that is catching up trying to transform into a usable instrument in the Web app developer’s toolbox.

It’s the past and the future. The chances are slim that any particular vendor will win all or even 80% of the market of the mobile devices. In competitive business, being able to make an application available ONLY to 80% of the market is not good enough, hence the chances that any particular native platform will dominate in the Web developers are slim. HTML5 and related technologies will serve as a common denominator for mobile developers.

Check out one of the trading applications named **tradeMonster**. It has been developed using HTML5 and uses the same code base for all mobile devices. The desktop version was built using Adobe Flex framework and runs in Flash Player’s VM. Yes, they have created native wrappers to offer tradeMonster in Apple or Google’s application stores, but it’s still an HTML5 application nevertheless. Create a paper trading account (no money needed) and test their application. If you like it, consider developing in HTML5.

Enterprise IT managers need the cross platform development and deployment platform, which HTML5 is promising to become. But take with a grain of salt all the promises of being 100% cross-platform made by any HTML5 framework vendor. *“With our HTML5 framework you won’t need to worry about differences in Web browsers”*. Yeah, right! HTML5 is not a magic bullet, and don’t expect it to be. But HTML5 is for real and may become the most practical development platform for your organization today.

# Dart: A Promising Language

Unfortunately, developing an application in JavaScript is not overly productive. Some people use CoffeScript or TypeScript to be converted for JavaScript for deployment. We are closely watching the progress with Google's new programming language called **Dart**, which is a compiled language with an elegant and terse syntax. Dart's easy to understand to anyone who knows Java or C#. Although compiled version of the Dart code requires Dartium VM, which is currently available only in the Chromium browser, Google created *dart2js* compiler that turns your application code into JavaScript in seconds so it can run in all Web browsers today. Google also offers Dart IDE with debugger and autocomplete features. You can debug the Dart code in Dart Editor while running generated JavaScript in the browser.

Dart's VM can communicate with JavaScript's VM, so if you have a portion of your application written in JavaScript, it can peacefully coexist with the Dart code. You can literally have two buttons on the Web page: one written in JavaScript and the other in Dart.

W3C published a document called "[Introduction to Web Components](#)", which among other things defines recommendations on how to create custom HTML components. The existing implementation of Web UI package includes a number of UI components and allows defining new custom HTML elements in a declarative way. Here's an example we borrowed from the [Dart Web site](#):

```
<element name="x-click-counter" constructor="CounterComponent" extends="div">
  <template>
    <button on-click="increment()">Click me</button>
    <span>(click count: {{count}})</span>
  </template>
  <script type="application/dart">
    import 'package:web_ui/web_ui.dart';

    class CounterComponent extends WebComponent {
      int count = 0;
      void increment(e) { count++; }
    }
  </script>
</element>
```

This code extends the Web UI element `div` and includes a template, which uses binding. The value of the variable `count` is bound to `<span>` and as soon as a counter increases, the Web page immediately reflects its new value without the need to write any other code. The Web UI package will be replaced soon with the [Polymer Stack](#) built on top of Web components.

Google have ported their popular JavaScip framework Angular.js into [Angular.dart](#). Farata Systems is working on <https://github.com/Farata/dart-pint> [Pint], which is an

open source Pint is a library of Angular.Dart components built on top of **Semantic UI**, a library of rich UI components for developing responsive Web applications.

In 2014, the popularity of Dart should increase. In this case, we'll send a new proposal to O'Reilly Media for a book titled "Enterprise Web Development with Dart".

## HTML5 is in Demand Today

Having said that, we'd like you to know that at the time of this writing the popular job search engine Indeed.com reports that HTML5 is **the #1 job trend** - the fastest growing keyword found in online job postings - ahead of iOS in third place and Android in fourth place. We'll be happy if our book will help you in mastering HTML5 and finding an interesting and financially rewarding job!

---

# Appendix A. Advanced Introduction to JavaScript

*“Atwood’s Law: any application that can be written in JavaScript, will eventually be written in JavaScript.*

This Appendix is dedicated to the JavaScript programming language. While all chapters of this book show how JavaScript frameworks can greatly minimize the amount of the JavaScript code that you need to write manually, you still need to understand the language itself. We assume that you have some programming experience with any programming, understand the HTML syntax, and are familiar with general principal of communications between Web browsers and Web servers. We’ve included the word *advanced* in the title of this Appendix because of these assumptions. We’ll start with covering basics of the language, but then quickly progress to such advanced topics as *prototypal inheritance, callbacks, and closures*.



If you’re absolute beginner with Web development and had no previous exposure to JavaScript, consider reading one of the fundamental tutorials covering each and every detail of JavaScript. We can recommend you the book “JavaScript: The Definite Guide”, Sixth Edition by David Flanagan.

Besides the JavaScript coverage this Appendix includes a section on the tools (IDEs, debuggers, Web inspectors et al.) that will make your development process more productive.

## JavaScript: A Brief History

The JavaScript programming language was designed in 1995 by Brendan Eich, who was working for Netscape Communications Corporation at the time. His goal was to allow

developers create more interactive Web pages. Initially the name of this language was Mocha, then LiveScript, and finally Netscape agreed with Sun Microsystems, creator of Java to rename it to JavaScript.

A year later, the language was given to an international standards body called ECMA, which formalized the language into *ECMAScript standard* so that other Web browser vendors could create their implementation of this standard. JavaScript is not the only language created based on the ECMAScript specification - ActionScript is a good example of another popular dialect of ECMAScript.

To learn more about the history of JavaScript from the source watch the Brendan Eich's presentation "[JavaScript at 17](#)" at O'Reilly's conference Fluent 2012.

Vast majority of today's JavaScript code is being executed by the Web browsers, but there are JavaScript engines that can execute JavaScript code independently. For example, [Google's V8](#) JavaScript engine implements ECMAScript 5 and is used not only in the Chrome browser, but can run in a standalone mode and can be embedded in any C++ application. Using the same programming language on the client and the server is the main selling point of node.js, which runs on top of V8. Oracle's Java Development Kit (JDK) 8 will include the JavaScript engine Nashorn that not only can run on both the server and the client computers, but also allows to embed the fragments of JavaScript into Java programs.

In the ninetieth, JavaScript was considered a second class language used mainly for prettifying Web pages. In 2005 the techniques known as AJAX (see Chapter 2) made a significant impact to the way Web pages were built. With AJAX, the specific content inside the Web page could be updated without the need to make a full page refresh. For example, Google's gmail that inserts just one line at the top of your input box whenever the new email arrives - it doesn't usually re-retrieve the entire content of your Inbox from the server and definitely doesn't re-render the Web page.

AJAX gave a second birth to JavaScript. But the vendors of Web browsers were not eager to implement the latest specifications of ECMAScript. Browsers' incompatibility and lack of good development tools prevented JavaScript from becoming the language of choice for Web applications. Let's not forget about the ubiquitous Flash Player – a popular VM supported by all desktop Web browsers. Rich Internet Applications written in ActionScript were compiled into the byte code and executed by Flash Player on the user's machine inside the Web browser.

If AJAX saved JavaScript, then rapid proliferation of tablets and smartphones made it really hot. Today's mobile devices come equipped with modern Web browsers, and in the mobile world there is no need to make sure that your Web application will work in the 4-year old Internet Explorer 8. Adobe's decision to stop supporting Flash Player in the mobile Web browsers is yet another reason to turn to JavaScript if your Web application has to be accessed from smartphones or tablets.

ECMAScript, 5th Edition has been published in 2009 and is currently supported by all modern Web browsers. If you are interested in discovering if specific features of ECMAScript 5 are supported by a particular Web browser, check the latest version of the [ECMAScript 5 compatibility table](#). At the time of this writing the snapshot of the Chrome Browser v. 22 looks as in [Figure A-1](#) below:

THIS BROWSER	IE 8	IE 9	IE 10	FF 3.5, 3.6	FF 4-13	SF 5	SF 5.1	SF 6
Object.create	Yes	No	Yes	No	Yes	Yes	Yes	Yes
Object.defineProperty	Yes	Yes [1]	Yes	No	Yes	Yes [6]	Yes	Yes
Object.defineProperties	Yes	No	Yes	No	Yes	Yes	Yes	Yes
Object.getPrototypeOf	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Object.keys	Yes	No	Yes	No	Yes	Yes	Yes	Yes
Object.seal	Yes	No	Yes	No	Yes	No	Yes	Yes
Object.freeze	Yes	No	Yes	No	Yes	No	Yes	Yes
Object.preventExtensions	Yes	No	Yes	No	Yes	No	Yes	Yes
Object.isSealed	Yes	No	Yes	No	Yes	No	Yes	Yes
Object.isFrozen	Yes	No	Yes	No	Yes	No	Yes	Yes
Object.isExtensible	Yes	No	Yes	No	Yes	No	Yes	Yes
Object.getOwnPropertyDescriptor	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Object.getOwnPropertyNames	Yes	No	Yes	No	Yes	Yes	Yes	Yes
Date.prototype.toISOString	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Date.now	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Array.isArray	Yes	No	Yes	No	Yes	Yes	Yes	Yes
JSON	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Function.prototype.bind	Yes	No	Yes	No	Yes	No	No [8]	Yes
String.prototype.trim	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes

Figure A-1. ECMAScript 5 Compatibility Sample Chart

JavaScript became the lowest common denominator available on thousands of different devices. Yes, the JavaScript engines are not exactly the same on thousands devices that people use to login to Facebook, but they are pretty close, and using some of the JavaScript frameworks spare you from worrying about their incompatibilities.

JavaScript is an interpreted language that arrives to the Web browser as text. The JavaScript engine optimizes and compiles the code before the execution. The JavaScript engine is a part of the Web browser, which will load and execute the JavaScript code embedded or referenced between the HTML tags `<script>` and `</script>`. JavaScript was originally created for Web browsers, which were supposed to display whatever content has successfully arrived. What if an image has not arrived from the server? You'll see a broken image icon. What if erroneous JavaScript code with syntax errors has arrived to the browser? Well, the engine will try to execute whatever code has arrived. The end users may appreciate such browser's forgiveness when at least some content is available, but the software developers should be ready to spend more time debugging (in multiple browsers) the errors that could have been caught by compilers in other programming languages.

# JavaScript Variables

JavaScript is a weakly typed language hence the developers don't have a luxury of strong compiler's help that Java or C# developers enjoy. For those unfamiliar with weakly typed languages, let us explain. Imagine that if in Java or C# instead of declaring variables of specific data types everything would be of type `Object`, and you could assign to it any value – a string, a number, or a custom object `Person`. This would substantially complicate the ability of the compiler to weed out all possible errors. You don't need to declare variables in JavaScript – just assign a value and the JavaScript engine will figure out the proper type during the execution of your code. For example, the variable named `girlfriend` will have a data type of `String`:

```
girlfriendName="Mary";
```

Since we haven't used the keyword `var` in front of `girlfriend`, this variable will have the global scope, which is a big no-no. Creating your global variables in an Web application that often includes multiple libraries from different vendors can easily create situations when the variable's value has been accidentally replaced by someone else's code. Besides, if the application uses concurrent execution (read about Web Workers in Appendix B) using global variable can lead to race conditions. Pretty soon you'll see how to limit the scope of application's variables to avoid polluting global space.

Variables declared with the `var` keyword inside functions are have local scope and are not visible from outside of the function. Consider the following function declaration:

```
function addPersonalInfo(){
    var address ="123 Main Street";           // local String variable
    age=25;                                    // global Number variable
    var isMarried = true;                      // local boolean variable
    isMarried = "don't remember";              // now it's of String type
}
```

The variables `address` and `isMarried` are visible only inside the function `addPersonalInfo()`. The variable `age` becomes global because of omission of the keyword `var`.

The variable `isMarried` changes its type from `Boolean` to `String` during the execution of the above script, and JavaScript engine won't complain assuming that the programmer knows what she's doing. So be ready for the run-time surprises and allocate a lot more time for testing than with programs written in compiled languages.

# Adding JavaScript to HTML

Software developers either directly include the JavaScript code to the HTML document by placing it between the tags `<script>` and `</script>` or include a reference to the external location of the code (e.g. a local file name or a URL) in the `src` attribute of the `<script>` tag. We usually place the `<script>` tags at the end of HTML file. The

reason is simple - your JavaScript code may be manipulating with HTML elements, and you want them to exist by the time the script runs. The other way to ensure that the code will run only after the Web page has loaded is by catching window's 'load' event (you'll see such example later in this chapter in the section on browser's events). Some JavaScript frameworks may have their own approach to dealing with HTML content and in Chapter 4 you'll see that the main HTML file of the Web application written with Ext JS framework has <script> tags followed by the empty <body> tags. But let's keep things simple for now.



The code samples for this book are included at the location mentioned in the Preface. The authors of this book use WebStorm IDE from JetBrains (see Appendix C for details).

Create a new HTML file in WebStorm IDE called new\_file.html and add the following fragment at the very end (right above the closing </body> tag):

```
<h1>Hello World</h1>

<script>
    alert("Hello from JavaScript");
</script>
```

Right-click on the new\_file.html in WebStorm, select Open in Browser, and you'll see the following output in your Web browser:

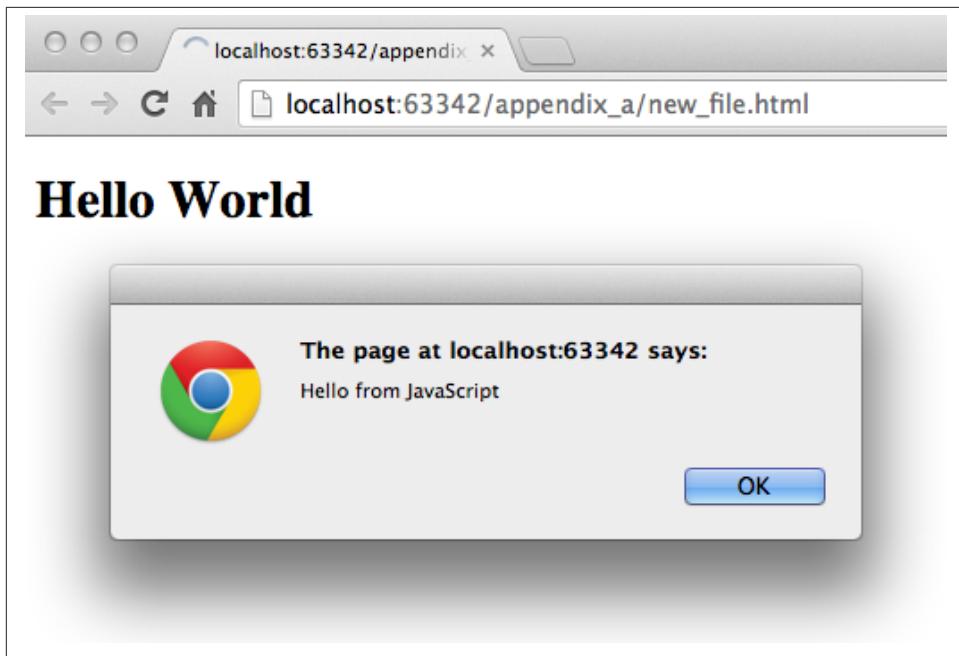


Figure A-2. Running MyFirstProject with JavaScript at the bottom

Note that the Alert popup box is shown on top of the Web page that already rendered its HTML component `<h1>`. Now move the above code from the `<body>` up to the end of the `<head>` section and re-open new\_file.html. The picture is different now - the Alert box is shown before the HTML rendering is complete (see [Figure A-3](#)).

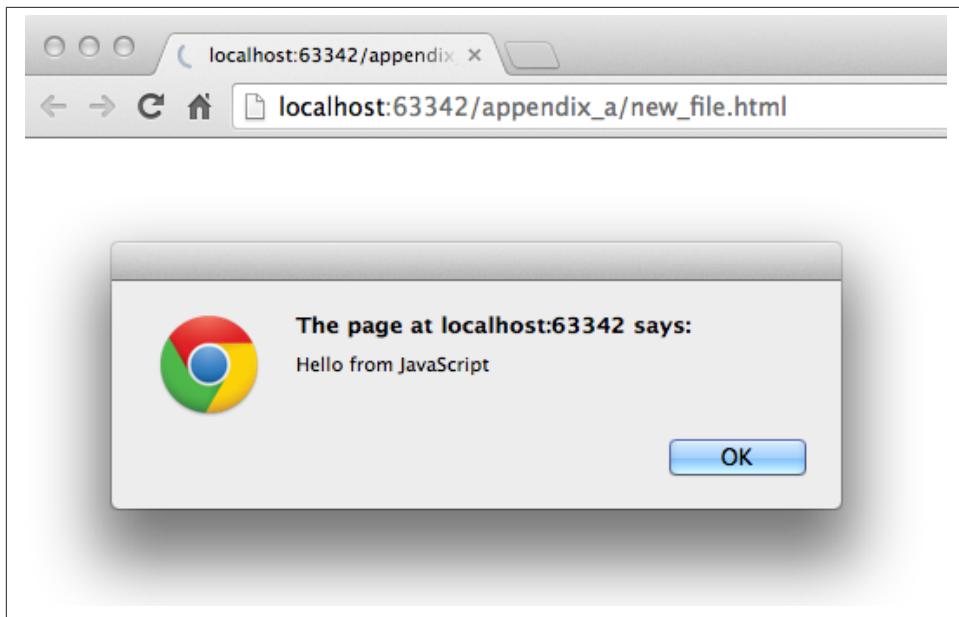


Figure A-3. Running HTML with JavaScript in the `<head>` section

This code sample doesn't cause any malfunctioning of the code, but if our JavaScript would need to manipulate with HTML elements, we'd run into issues of accessing non-existent components. Beside simple Alert box, JavaScript has Confirm and Prompt boxes, which allow asking OK/Cancel type of questions or request some input from the user.

## Debugging JavaScript in Web Browsers

The best way to learn any program is to run it step by step through a debugger. While some people appreciate using debuggers offered by the IDE, we prefer to debug using great tools offered by the major Web browsers:

- Firefox: Firebug add-on
- Chrome: Developer Tools
- Internet Explorer: F12 Developer Tools
- Safari: the menu Develop
- Opera: Dragonfly

We'll be doing most of the debugging either in Firebug or Chrome Developer Tools. Both of them provide valuable information about your code and are easy to use. To get

Firebug go to [www.getfirebug.com](http://www.getfirebug.com) and press the red button Install Firebug and follow the instructions. In Firefox, open the Firebug panel from the menu View.

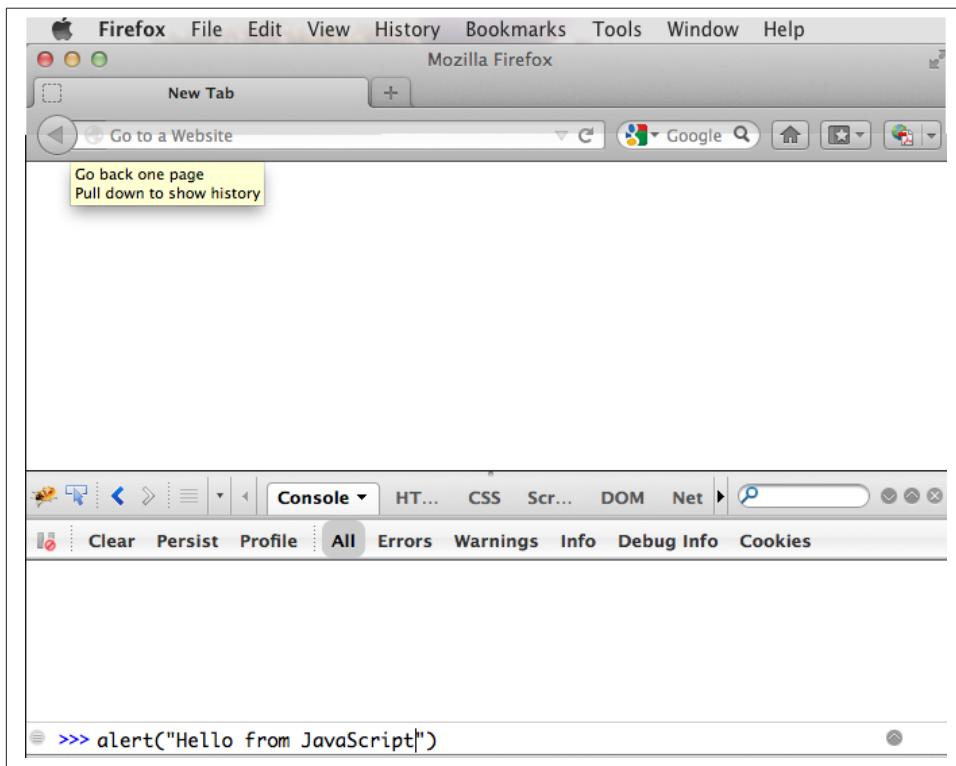


Figure A-4. FireBug Console

Select the Console option on the Firebug toolbar, [enable the console](#) and enter `alert("Hello from JavaScript")` after the `>>>` sign and you'll see the Alert box. To enter multi-line JavaScript code press the little circle with a caret at the bottom right corner and Firebug will open a panel on the right, where you can enter and run your JavaScript code.

This was probably the last example where we used the `Alert()` popup box for debugging purposes. All JavaScript debuggers support the `console.log()` for printing debug information. Consider the following example that illustrate strict equality operator `==`. Yes, it's three equal signs in a row. This operator evaluates to true if the values are equal and the data types are the same.

```
var age=25;  
  
var ageStr="25";
```

```

if (age==ageStr){
    console.log("The values of age and ageStr are equal");
}

if (age==ageStr){
    console.log("The values of age and ageStr are strictly equal");
} else{
    console.log ("The values of age and ageStr are not strictly equal");
}

```

Running this code in the Firebug console produces the following output:

*Figure A-5. Using `console.log()` for the debug output*



You can also use `console.info()`, `console.debug()`, and `console.error()` so the debuggers may highlight the output with different colors or mark with different icons.



For more information about debugging JavaScript refer to the code samples illustrated in [Figure A-6](#) and [Figure A-7](#).

# JavaScript Functions. Gentle Introduction

JavaScript can be called an object-oriented language cause it allows an object to inherit existing functionality from another object, and you can encapsulate the data and restrict the data access. It's done not as simple as in classical object-oriented languages, but it's possible. Now comes the chicken or the egg dilemma. What should be explained first - the syntax of functions or creation of objects? Understanding of objects is needed for some of the function code samples and visa versa. We'll start with simple function use cases, but will be switching to objects as needed.

Many of the readers can have experience with object-oriented languages like Java or C#, where classes can include *methods* implementing required functionality. Then these methods can be invoked with or without instantiation of the objects. If a JavaScript object includes functions they are called *methods*. But JavaScript functions don't have to belong to an object. You can just declare a function and invoke it. Just like this:

```
//Function declaration
function calcTax (income, dependents){
    var tax;
    // Do stuff here
    return tax;
}

//Function invocation
calcTax(50000, 2);
var myTax = calcTax(50000,2);
```



Please note that the data types of the function parameters `income` and `dependents` are not specified. We can only guess that they are numbers based on their names. If a software developer won't bother giving meaningful names to function parameters, the code becomes difficult to read.

After the function `calcTax()` is invoked and complete, the variable `myTax` will have the value returned by the function.

Another important thing to notice is that our function has a name `calcTax`. But this is not always the case - JavaScript allows functions to be *anonymous* - you'll see an example of anonymous functions in the function expressions below (note the absence of a name after the keyword `function`).



If you see the line of code where the keyword `function` is preceded by any other character this is not a function declaration, but a function expression.

Consider the following variation of the tax calculation sample:

```
//Function expression
var doTax=function (income, dependents){
    //do stuff here
    return tax;
}

//Function invocation
var myTax=doTax(50000,2);
```

In the code above the `function` keyword is being used in the expression - we assign the anonymous function to the variable `doTax`. After this assignment just the text of the function is assigned to the variable `doTax` - the anonymous function is not being invoked just yet. It's important to understand that even though the code of this anonymous function ends with `return tax;` actually, the tax calculation and return of its value is not happening until the `doTax()` is invoked. Only then the function is evaluated and the variable `myTax` will get whatever value this function returns.

Yet another example of a function expression is when it's placed inside the *grouping operator* - parentheses as shown below. As in an arithmetic expressions, this means that the the content inside the expressions has to be evaluated first, and then used in an expression:

```
(function calcTax (income, dependents){
    // Do stuff here
});
```

The outermost parentheses hide its internal code from the outside world creating a scope or a closed ecosystem, where the function's code will operate. Try to add a line invoking this function after the last line in the above code sample, e.g. `calcTax(50000,2)`, and you'll get an error - "calcTax is not defined". There is a way to expose some of the internal content of such a *closure* and you'll see how to do it later in this appendix.

If you'll take away the outermost parentheses and the closing semicolon, you'll get the function declaration, which will be subject to *hoisting* (we'll explain it explained soon). Function expressions are usually a part of a larger expression. For example, if you'll add a couple of parentheses at the end of this expression, you'll get a *self-invoked* function. This extra pair of parentheses will cause the function expression located in the first set of parentheses to be executed right away.

```
(function calcTax (income, dependents){
    // Do stuff here
})();
```



The topic “Function declaration vs. function expressions” is one of those fuzzy JavaScript areas that can cause unexpected behavior of your code. Angus Croll published a [well-written article](#) on this subject.

## JavaScript Objects. Gentle Introduction

JavaScript objects are simply unordered collections of properties. You can assign new or delete existing properties from the objects during the runtime whenever you please. In classical object oriented languages there are *classes* and there are *objects*. However JavaScript doesn't have classes.



The ECMAScript 6 specification will include classes too, but since it's a work in progress we won't consider them as something useful in the today's world. If you'd like to experiment with the upcoming features of JavaScript, download the [Chrome Canary browser](#), go to `chrome:flags` and enable experimental JavaScript. Chrome Canary should be installed on the computer of any HTML5 developers - you can use today those features that will be officially released in Chrome Developer Tools in about three months.

In JavaScript you can create objects using one of the following methods:

- Using object literals
- Using `new Object()` notation
- Using `Object.create()`
- Using *constructor functions* and a `new` operator.

Technically, there can be other APIs that implicitly create objects, e.g. `JSON.parse()`, but let's keep things simple.



In JavaScript everything is an `Object`. Think of `Object` as of a root of the hierarchy of all objects used in your program. All your custom objects are descendants from `Object`.

## Object Literals

The easiest way to create a JavaScript object is by using the object literal notation. The code sample below starts with a creation of an empty object.

```
var t = {} // create an instance of an empty object
```

The following line of code creates an object with one property `salary` and assigns the value of 50000 to it.

```
var a = {salary: 50000}; // an instance with one property
```

Below, the instance of one more object is created and the variable `person` points at it.

```
// Store the data about Julia Roberts
var person = { lastName: "Roberts",
               firstName: "Julia",
               age: 42
             };
```

This object has three properties: `lastName`, `firstName`, and `age`. Note that in object literal notation the values of these properties are specified using colon. You can access the properties of this person using the dot notation, e.g. `person.LastName`. But JavaScript allows yet another way of accessing the object properties by using square bracket syntax, for example `person["lastName"]`. In the next code sample you'll see that using the square brackets is the only way to access the property.

```
var person = {
  "last name": "Roberts",
  firstName: "Julia",
  age: 42};

var herName=person.lastName;           // ①

console.error("Hello " + herName);    // ②

herName=person["last name"];          // ③

person.salutation="Mrs. ";

console.log("Hello "+ person.salutation + person["last name"]); // ④
```

- ① The object `person` doesn't have a property `lastName`, but no error is thrown
- ② This will print "Hello undefined"
- ③ Using an alternative way of referring to an object property
- ④ This will print "Hello Mrs. Roberts"



It's a good idea to keep handy a style guide of any programming language, and we know two of such documents for JavaScript. Google has published their version of JavaScript Style Guide at <http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml>. A more detailed Airbnb JavaScript Style Guide is available as a github project at <https://github.com/airbnb/javascript>. And the github version of the JavaScript style guide is located at <https://github.com/styleguide/javascript>.

## Nesting Object Literals

Objects can contain other objects. If a property of an object literal is also an object, you just need to specify the value of this property in an extra pair of curly braces. For example, you can represent the telephone of a person as an object having two properties: the type and the number. The following code snippet adds a nested object to store a work phone as a *nested object* inside the person's object. Run this code in the Firebug's console and it'll print "Call Julia at work 212-555-1212".

```
var p = { lastName: "Roberts",
           firstName: "Julia",
           age: 42,
           phone:{ type: "work",
                     numb: "212-555-1212"
                 }
       };
console.log("Call " + p.firstName + " at " + p.phone.type + " " + p.phone.numb );
```

What if a person has more than one phone? We can change the name of the property phone to phones and instead store an array of objects. JavaScript arrays are surrounded by square brackets, and they are zero based. The following code snippet will print "Call Julia at home 718-211-8987".

```
var p = { lastName: "Roberts",
           firstName: "Julia",
           age: 42,
           phones:[{ type: "work",
                      numb: "212-555-1212"
                  },
                  {
                      type: "home",
                      numb: "799-211-8987"
                  }
                ];
console.log("Call " + p.firstName + " at " + p.phones[1].type + " " + p.phones[1].numb );
```

## Methods in Object Literals

Functions defined inside objects are called *methods*. Defining methods in object literals is similar to defining properties - provide a method name followed by a colon and the function declaration. The code snippet below declares a method `makeAppointment()` to our object literal. Finally, the line `p.makeAppointment();` invokes this new method, which will print the message that Steven wants to see Julia and will call at so-and-so number.

```
var p = { lastName: "Roberts",
          firstName: "Julia",
          age: 42,
          phones:[{
            type: "work",
            numb: "212-555-1212"
          },
          {
            type: "home",
            numb: "718-211-8987"
          }],
          makeAppointment: function(){
            console.log("Steven wants to see " + this.firstName +
                        ". He'll call at " + this.phones[0].numb);
          }
        };

p.makeAppointment();
```



Since we already started using arrays, it's worth mentioning that arrays can store any objects. You don't have to declare the size of the array upfront and can create new arrays as easy as `var myArray=[]` or `var myArray=new Array()`. You can even store function declarations as regular strings, but they will be evaluated on the array initialization. For example, during the `greetArray` initialization the user will see a prompt asking to enter her name, and, when it's done, the `greetArray` will store two strings. The output of the code fragment below can look like "Hello, Mary".

```
var greetArray=[
  "Hello",
  prompt("Enter your name", "Type your name here")
];

console.log(greetArray.join(", "));
```

We've briefly covered object literals, and you to start using them. Chapter 2 covers JSON - a popular data format used as replacement for XML in the JavaScript world. You can see there that the syntax of JSON and JavaScript object literals are similar. Now we'll

spend a little bit of time delving into JavaScript functions, and then - back to objects again.

## Constructor Functions

JavaScript functions are more than just some named pieces of code that implements certain behavior. They also can become objects themselves by a magic of the `new` operator. To make things even more intriguing, the function calls can have memories, which will be explained in the section about closures.

If a function is meant to be instantiated with the `new` operator it's called a *constructor function*. If you are familiar with Java or C# you understand the concept of a class constructor that is being executed only once during the instantiation of a class. Now imagine that there is only a constructor without any class declaration that still can be instantiated with the `new` operator as in the following example.

```
function Person(lname, fname, age){  
    this.lastName=lname;  
    this.firstName=fname;  
    this.age=age;  
}  
  
// Creating 2 instances of Person  
var p1 = new Person("Roberts", "Julia", 42);  
  
var p2 = new Person("Smith", "Steven", 34);
```

This code declares the function `Person` and after that, with the help of the `new operator` it creates two instances of the `Person` object referred by the variables `p1` and `p2` accordingly.

According to common naming conventions the names of the constructor functions are capitalized.



The JavaScript language doesn't support classes, and a constructor function is the closest concept to the classes of the languages like Java or C#. Chapter 4 is about the Ext JS framework that extends JavaScript and introduces constructs similar to classes and classical inheritance.

## Adding Methods and Properties to Functions

Objects can have methods and properties, right? On the other hand, functions are objects. Hence functions can have methods and properties too. If you declare a function `marryMe()` inside the constructor function `Person`, `marryMe()` becomes a method of `Person`. This is exactly what we'll do next. But this time we'll create an HTML file that

includes the <script> section referring to the JavaScript code sample located in a separate file.

If you want to try it hands-on, create a new file in your Aptana project by selecting the menu File | New | File and give it a name `marryme.js`. When prompted, accept the suggested default JavaScript template, and key in the following content into this file:

```
function Person(lname, fname, age){  
    this.lastName=lname;  
    this.firstName=fname;  
    this.age=age;  
  
    this.marryMe=function(person){  
        console.log("Will you marry me, " + person.firstName);  
    };  
  
};  
  
var p1= new Person("Smith", "Steven");  
var p2= new Person("Roberts", "Julia");  
  
p1.marryMe(p2);
```

The code above uses the keyword `this` that refers to the object where the code will execute. If you are familiar with the meaning of `this` in Java or C#, it's similar, but not exactly the same, and we'll illustrate it in the section titled "Who's this". The method `marryMe()` of one `Person` object takes an instance of another `Person` object and makes an interesting proposition: "Will you marry me, Julia".

This time we won't run this code in the Firebug's console, but rather will include it in the HTML file. In WebStorm, create a new HTML file `marryme.html`. Modify it to include the JavaScript file `marryme.js`:

```
<!DOCTYPE html>  
<html>  
    <head>  
        <meta charset="utf-8" />  
    </head>  
  
    <body>  
        <h1>Making Proposals</h1>  
  
        <script src="marryme.js"></script>  
    </body>  
</html>
```

## Debugging JavaScript in Firebug

Right-click on the file `marryme.html` and select Open in Browser. In Firefox and you'll see it open a new Web page that reads "Making Proposals". Open the Firebug using the

View menu of Firefox, refresh the page and switch to the Firebug's tab Script. You'll see the split panel with the JavaScript code from `marryme.js` on the left.

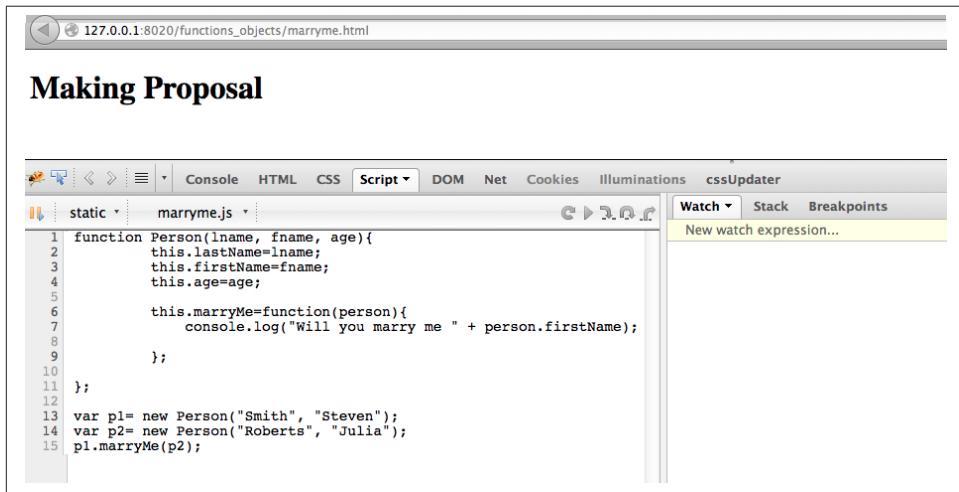


Figure A-6. Firebug's Script panel

Let's set a breakpoint inside the method `marryMe()` by clicking in the Firebug's gray area to the left of the line 7. You'll see a red circle that will reveal a yellow triangle as soon as your code execution will hit this line. Refresh the content of the browser to re-run the script with a breakpoint. Now the execution stopped at line 7, the right panel contains the runtime information about the objects and variables used by your program.

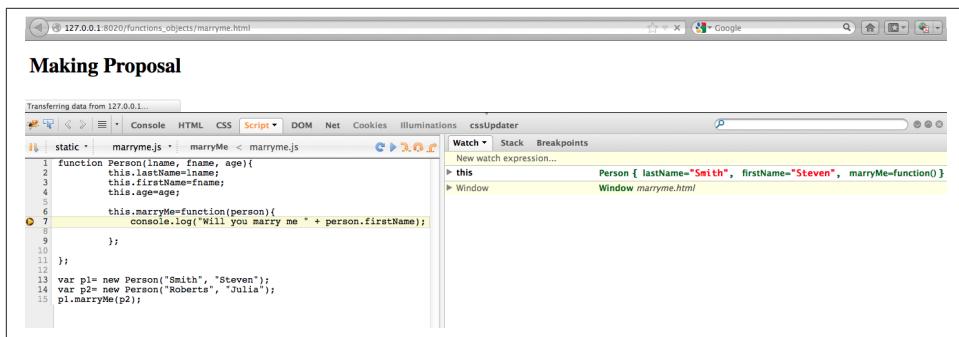


Figure A-7. Firebug's Script panel at a breakpoint

On the top of the left panel you'll see standard debugger buttons with curved arrows (Step Into, Step Over, Step Out) as well as triangular button to continue code execution. The right panel depicts the information related to `this` and global `Window` objects. In

**Figure A-7** this represents the instance of the Person object represented by the variable p1 (Steven Smith). To see the content of the object, received by the method marryMe() you can add the watch variable by clicking on the text “New watch expression...” and entering person - the name of the parameter of marryMe(). **Figure A-8** shows the watch variable person (Julia Roberts) that was used during the invocation of the method marryMe().

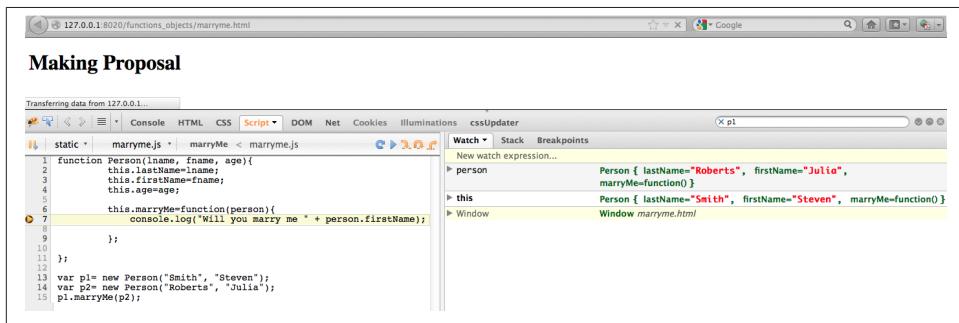


Figure A-8. Firebug’s Script panel at a breakpoint

Now click on the Firebug’s Net panel, which shows what goes over the network during communication between the Web browser and Web server. Figure 2-11 shows a screen shot of the Net panel where we clicked on the Headers tab for marryme.html and the Response tab of marryme.js. The code 200 for both files means that they arrived successfully to the browser. It also shows the IP address of the Web server they came from, their sizes, and plenty of other useful information. Both Script and Net panels of Firebug or any other developers tools are your best friends of any Web developer.

The screenshot shows the Firebug Net panel with two requests listed:

- GET marryme.html**: Status 200 OK, 127.0.0.1:8020, 159 B, 127.0.0.1:8020, 1ms. Headers show Connection: keep-alive, Content-Length: 159, Content-Type: text/html, Date: Sun, 18 Nov 2012 14:28:17 GMT, Server: HttpComponents/4.1.3.
- GET marryme.js**: Status 200 OK, 127.0.0.1:8020, 357 B, 127.0.0.1:8020, 1ms. Headers show Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8, Accept-Encoding: gzip, deflate, Accept-Language: en-US,en;q=0.5, Cache-Control: max-age=0, Connection: keep-alive, Host: 127.0.0.1:8020, User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:16.0) Gecko/20100101 Firefox/16.0.

The Response tab for 'marryme.js' displays the following JavaScript code:

```

function Person(lname, fname, age){
    this.lastName=lname;
    this.firstName=fname;
    this.age=age;

    this.marryMe=function(person){
        console.log("Will you marry me " + person.firstName);
    };
}

var p1= new Person("Smith", "Steven");
var p2= new Person("Roberts", "Julia");
n1.marryMe(n2);

```

Figure A-9. Firebug's Net panel

We like Firebug, but testing and debugging should be done in several Web browsers. Besides Firebug, we'll be using excellent Google Chrome developers tools. Their menus and panels are similar and we won't be including such mini-tutorials on using such tools - you can easily learn them on your own.



You can find a tutorial on using Google Chrome Developer Tools at <https://developers.google.com/chrome-developer-tools/>. The cheat-sheet of Chrome developer Tools is located at <http://anti-code.com/devtools-cheatsheet/>. Finally, Google offers an online video course titled “Explore and Master Chrome DevTools”.

## Notes on Arrays

A JavaScript array is a grab bag of any objects. You don't have to specify in advance the number of elements to store, and there is more than one way to create and initialize array instances. The following code samples are self-explanatory.

```

var myArray=[];
myArray[0]="Mary";
myArray[2]="John";

// prints undefined John
console.log(myArray[1] + " " + myArray[2]);

var states1 = ["NJ", "NY", "CT", "FL"];

var states = new Array(4); // size is optional

states[0]="NJ";

states[1]="NY";

states[2]="CT";

states[3]="FL";

// remove one array element
delete states[1];

// prints undefined CT length=4
console.log(states[1] + " " + states[2] + " Array length=" + states.length);

// remove one element starting from index 2
states.splice(2,1);

// prints undefined FL length=3
console.log(states[1] + " " + states[2] + " Array length=" + states.length);

```

Removing elements with `delete` creates gaps in the arrays while using the array's method `splice()` allows to remove or replace the specified range of elements closing gaps.

The next code sample illustrates an interesting use case when we assign a string and a function text as array elements to `mixedArray`. During array initialization the function `prompt()` will be invoked, the user will be prompted to enter name, and after that, two strings will be stored in `mixedArray`, for example “Hello” and “Mary”.

```

var mixedArray=[
  "Hello",
  prompt("Enter your name", "Type your name here")
];

```

## Prototypal Inheritance

JavaScript doesn't support classes, at least till the ECMAScript 6 will become a reality. But JavaScript allows you to create objects that inherit properties and methods of other objects. By default, all JavaScript objects are inherited from `Object`. Each JavaScript

construction function has a special property called `prototype`, which points at this object's ancestor. If you want to create an inheritance chain where instances of constructor function `ObjectB` extends `ObjectA` (similar to classical object-oriented languages) write one line of code like `ObjectB.prototype=ObjectA;`.

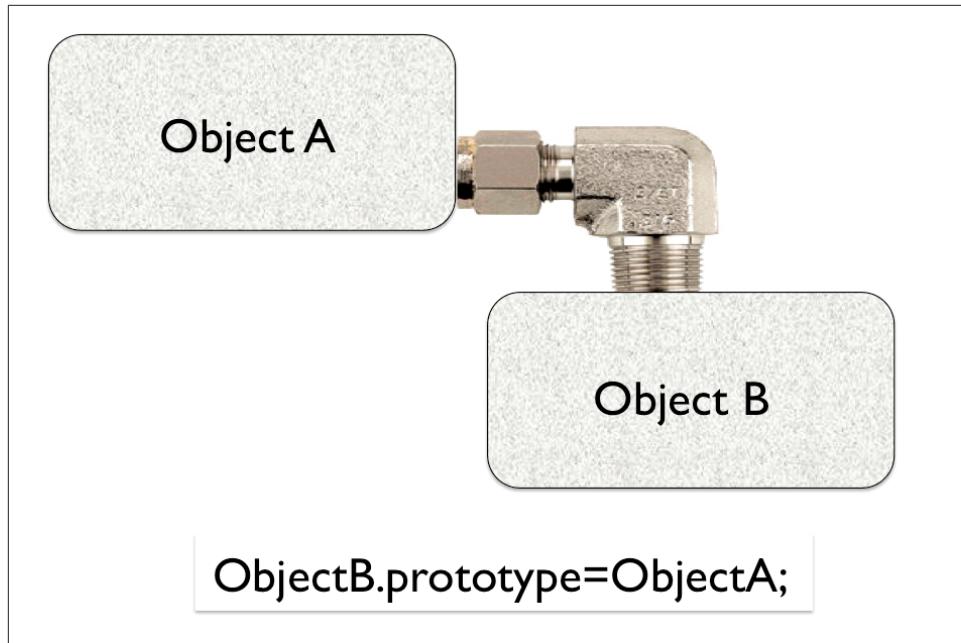


Figure A-10. Prototypal Inheritance

Consider two constructor functions `Employee` and `Person` shown in the code snippet below. They represent two unrelated objects. But assigning the `Person` object to the `prototype` property of `Employee` creates an inheritance chain, and now the object `emp` will have all properties defined in both `Employee` and `Person`.

```
function Person(name, title){  
    this.name=name;  
    this.title=title;  
    this.subordinates=[];  
}  
  
function Employee(name, title){  
    this.name=name;  
    this.title=title;  
}  
  
// All instances of Employee will extend Person  
Employee.prototype = new Person(); // ①
```

```
var emp = new Employee("Mary", "Specialist"); // ②  
console.log(emp); // ③
```

- ① Assigning an ancestor of type person
- ② Instantiating Employee
- ③ Printing the object referred by `emp` will output [object Object]. It happens because each object has a method `toString()`, and if you want it to output useful information - override it. You'll see how to do it later in this section.



The above code results in code duplication as the object referred by the variable `emp` will have a pair of the `name` and a pair of the `title` properties. You'll see how to avoid such duplication a bit later in the section "Avoiding Declaration Redundancy".

The property `prototype` exists on constructor functions. After creating specific instances of such objects you may see that these instances have another property called `proto`. At the time of this writing this property is not a standard yet and won't be supported in some older browsers, but ECMAScript 6 will make it official. To illustrate the difference between `prototype` and `proto` let's add the following piece of code to the previous code sample:

```
//Create an instance of Person and add property dependents  
var p=new Person();  
p.dependents=1; // ①  
  
var emp2=new Employee("Joe", "Father");  
  
//This employee will have property dependents  
emp2.__proto__=p; // ②  
  
console.log("The number of Employee's dependents " + emp2.dependents); // ③
```

- ① Creating an instance of `Person` and adding an extra property `dependents` just for this instance
- ② Assigning this instance to the `__proto__` property of one instance
- ③ The code will properly print 1 as a number of dependents of the `Employee` instance represented by the variable `emp2`. The variable `emp` from the previous code snippet won't have the property `dependents`.

To try it hands-on, open the file WhoIsYourDaddy.html (it's included in book code samples). Just for a change, this time we'll use Google Chrome Developer Tools by opening the browser's menu View | Developer | Developer Tools. Select the tab Sources and expand the panel on the left to select the file WhoIsYourDaddy.js. Set the breakpoint at the last line of the JavaScript, refresh the Web page content, and add the watch expressions (hit the + sign on top right) for the variables `p`, `emp`, and `emp2`. When the JavaScript code engine runs into `emp2.dependents` it tries to find this property in property on the `Employee` object. If not found, the engine checks all the objects in the prototypal chain (in our case it'll find it in the object `p`) all the way up to the `Object` if need be. Examine the values of these variable shown on [Figure A-11](#).



If your program need to work only with those properties that are defined on the specific object (not in its ancestors in the prototypal chain) use the method `hasOwnProperty()`.

```

Developer Tools - http://127.0.0.1:8020/functions_objects/WhoIsYourDaddy.html
Elements Resources Network Sources Timeline Profiles Audits Console
Paused
Watch Expressions
p: Person
dependents: 1
name: undefined
subordinates: Array[0]
title: undefined
__proto__: Person
emp: Employee
name: "Mary"
title: "Specialist"
__proto__: Person
name: undefined
subordinates: Array[0]
title: undefined
__proto__: Person
emp2: Employee
name: "Joe"
title: "Father"
__proto__: Person
dependents: 1
name: undefined
subordinates: Array[0]
title: undefined
__proto__: Person
Call Stack
(anonymous function)
Scope Variables
Global Window
Breakpoints

```

```

1 function Person(name, title){
2   this.name=name;
3   this.title=title;
4   this.subordinates=[];
5 }
6
7
8 function Employee(name, title){
9   this.name=name;
10  this.title=title;
11 }
12
13 // All instances of Employee will extend Person
14 Employee.prototype=new Person();
15
16 var emp=new Employee("Mary", "Specialist");
17
18 console.log(emp);
19
20 //Create an instance of Person and add property dependents
21 var p=new Person();
22 p.dependents=1;
23
24
25 var emp2=new Employee("Joe", "Father");
26 //This employee will have property dependents
27 emp2.__proto__=p;
28
29 console.log("The number of Employee's dependents "+ emp2.dependents);

```

*Figure A-11. The instance-specific `__proto__` variable*

Please note the difference in the content of the variables `__proto__` of the instances represented by `emp` and `emp2`. These two employees are inherited from two *different* objects `Person`. Isn't it scary? Not really.

## Avoiding Declaration Redundancy

Prototypical inheritance allows you to inherit one object from another, but it can lead to issues of redundancy and code duplication. If you take a closer look at the screenshot from [Figure A-11](#) you'll see that the `Person` and `Employee` objects have redundant properties `name` and `title`. We'll deal with this redundancy in the section titled "Call and Apply". But first let's introduce and cure the redundancy in method declarations when the prototypal inheritance is used.

Let's add a method to `addSubordinate()` to the ancestor object `Person` that will populate its array `subordinates`. Who knows, maybe an object `Contractor` (descendant of a `Person`) will need to be introduced to the application in the future, so the ancestor's method `addSubordinate()` can be reused. **First, we'll do it the wrong way to illustrate the redundancy problem**, and then we'll do it right. Consider the following code:

```
// Constructor function Person
function Person(name, title){
    this.name=name;
    this.title=title;
    this.subordinates=[];

    // Declaring method inside the constructor function
    this.addSubordinate=function (person){
        this.subordinates.push(person)
    }
}

// Constructor function Employee
function Employee(name, title){
    this.name=name;
    this.title=title;
}

// Changing the inheritance of Employee
Employee.prototype = new Person();

var mgr = new Person("Alex", "Director");
var emp1 = new Employee("Mary", "Specialist");
var emp2 = new Employee("Joe", "VP");

mgr.addSubordinate(emp1);
mgr.addSubordinate(emp2);
console.log("mgr.subordinates.length is " + mgr.subordinates.length);
```

The method `addSubordinate()` here is declared inside the constructor function `Person`, which becomes an ancestor of the `Employee`. After instantiation of two `Employee` objects the method `addSubordinate()` is duplicated for each instance.

Let's use Google Chrome Developer Tools profiler to see the sizes of the objects allocated on the Heap memory. But first we'll set up two breakpoints - one before, and one after creating our instances as shown on [Figure A-12](#).

The screenshot shows the Google Chrome Developer Tools interface. The title bar reads "Developer Tools – http://127.0.0.1:8020/functions\_objects/WhereToDeclareMethods.html". The top navigation bar includes tabs for Elements, Resources, Network, Sources, Timeline, Profiles, Audits, and Console. The Sources tab is active, displaying the code from "WhereToDeclareMethods.js". A red dot marks a breakpoint on line 20, where `Employee.prototype = new Person();` is written. The right side of the interface features the Call Stack panel, which is currently empty, showing only "(anonymous function)". Other panels like Watch Expressions, Breakpoints, and DOM Breakpoints are also visible.

```
// Constructor function Person
function Person(name, title){
    this.name=name;
    this.title=title;
    this.subordinates=[];
}

this.addSubordinate=function (person){
    this.subordinates.push(person)
}

// Constructor function Employee
function Employee(name, title){
    this.name=name;
    this.title=title;
}
// Changing the inheritance of Employee
Employee.prototype = new Person();

var mgr = new Person("Alex", "Director");
var emp1 = new Employee("Mary", "Specialist");
var emp2 = new Employee("Joe", "VP");

mgr.addSubordinate(emp1);
mgr.addSubordinate(emp2);
console.log("mgr.subordinates.length is " + mgr.subordinates.length);
```

*Figure A-12. Preparing Breakpoints Take 1.*

When the execution of the code will stop at the first breakpoint, we'll switch to the Profiler tab and take the first Heap snapshot. Upon reaching the second breakpoint we'll take another Heap snapshot. The dropdown at the status bar allows to view the objects allocated between the snapshots 1 and 2. [Figure A-13](#) depicts this view of the profiler. Note that the total size (the Shallow Size column) for the `Person` instances is 132 bytes. `Employee` instances weigh 104 bytes.

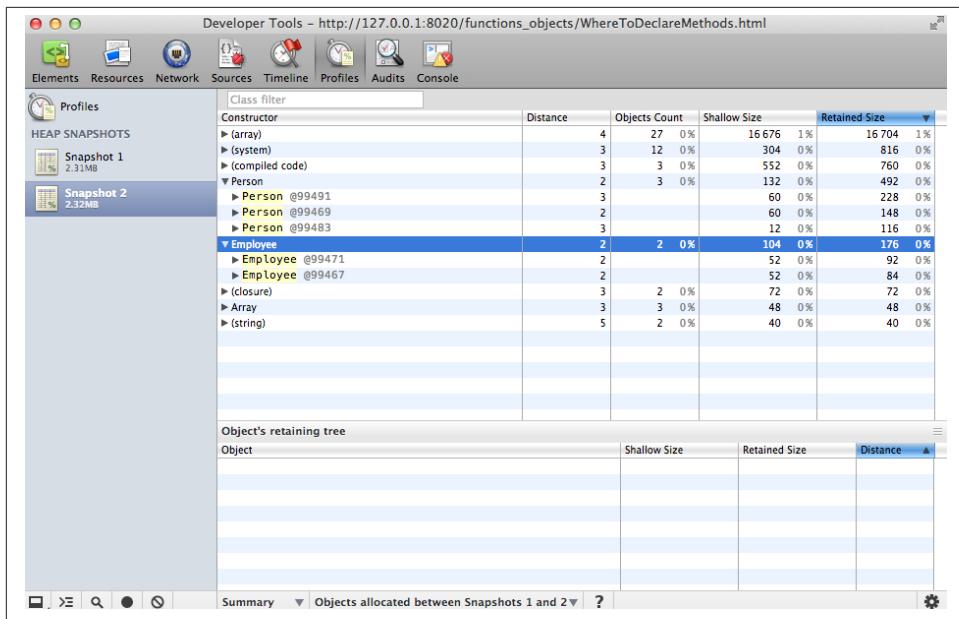


Figure A-13. Objects allocated between snapshots 1 and 2

Now we'll change the code to declare the method not inside the Person constructor function, but on its prototype - and **this is the right way to declare methods in functions to avoid code duplication**.

```
// Constructor function Person
function Person(name, title){
    this.name=name;
    this.title=title;
    this.subordinates=[];
}

//Declaring method on the object prototype
Person.prototype.addSubordinate=function(subordinate){
    this.subordinates.push(subordinate);
    return subordinate;
}

// Constructor function Employee
function Employee(name, title){
    this.name=name;
    this.title=title;
}

// Changing the inheritance of Employee
Employee.prototype = new Person();
```

```

var mgr = new Person("Alex", "Director");
var emp1 = new Employee("Mary", "Specialist");
var emp2 = new Employee("Joe", "VP");

mgr.addSubordinate(emp1);
mgr.addSubordinate(emp2);
console.log("mgr.subordinates.length is " + mgr.subordinates.length);

```

Similarly, we'll set up two breakpoints before and after object instantiation as shown in <>>FIG2-16>.

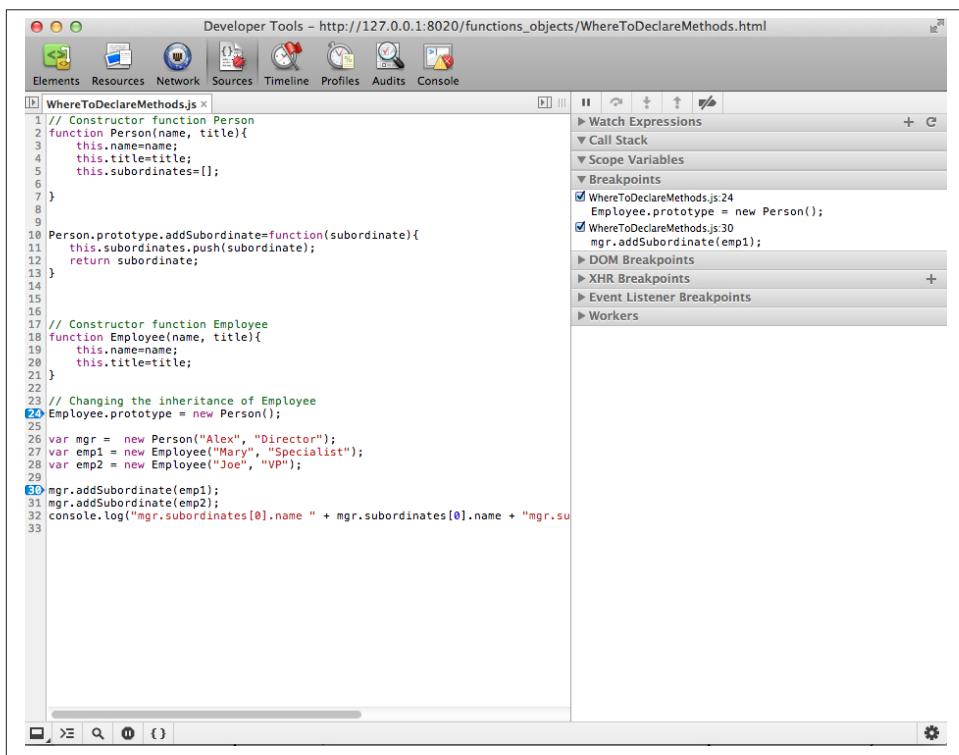
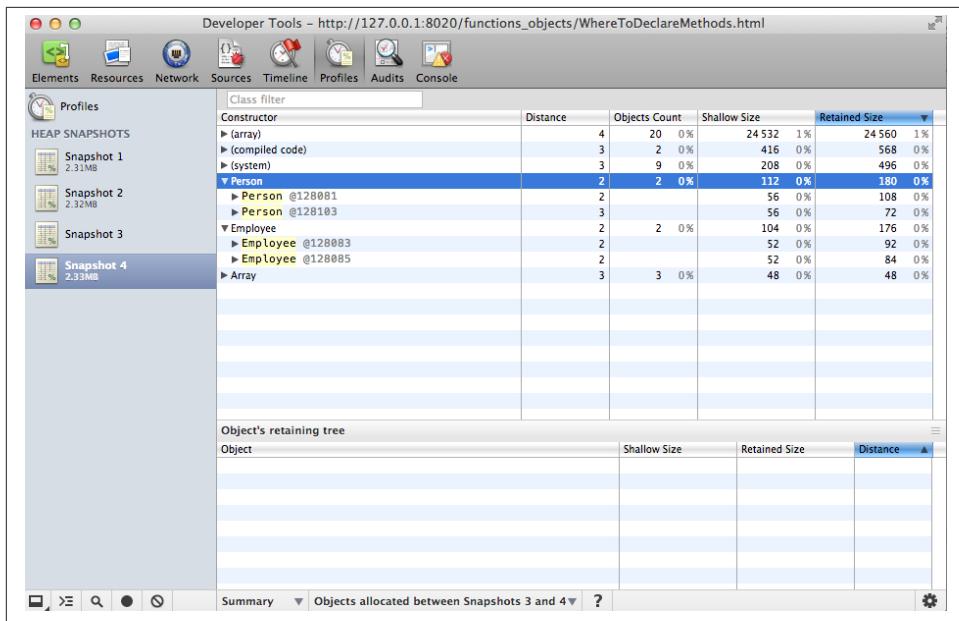


Figure A-14. Preparing Breakpoints Take 2.

Let's take two more profiler snapshots upon reaching each of the breakpoint. While the weight of the `Employee` instances remained the same (104 bytes), the `Person` instances became lighter: 112 bytes. While 20 bytes may not seem like a big deal, if you'll need to create hundreds or thousands of object instances it adds up.



*Figure A-15. Objects allocated between snapshots 3 and 4*

So if you need to declare a method on the object that will play a role of the ancestor, do it on the prototype level. The only exception to this rule is the case when such method needs to use some object specific variable that's different for each instance - in case declare methods inside the constructors (see the section on closures for details).



All modern Web browsers support the function `Object.create()`, which creates a new object based on another prototype object creates a new object and sets that new object's prototype to be the object passed in. For example, `var objectB=Object.create(objectA);`. What if you must support an older browser and need such “create by example” functionality? Of course, you can always create a custom arbitrarily named function with the similar functionality as the latest implementation of `Object.create()`. But the future-proof approach is to create the missing methods with the same signatures and on the same objects as the latest ECMAScript specification prescribes. In case of `Object.create()` you can use the implementation offered by Douglas Crockford:

```
if (typeof Object.create !== 'function') {  
    Object.create = function (o) {  
        function F() {}  
        F.prototype = o;  
        return new F();  
    };  
}  
newObject = Object.create(oldObject);
```

Such approach of custom implementation of missing pieces according to the latest ECMAScript specifications or W3C drafts is known as *polyfills*. People who can't wait till the browser vendors will implement the newest functionality create cross-browser polyfills and some of them submit their source code to the public domain. You can find a number of polyfills in the git repository of the [Modernizr project](#). The Web site <http://caniuse.com/> contains the current information about browser's support of the latest HTML5, JavaScript, and CSS features.



In the Ext JS chapter you'll see how this frameworks offers its own class system that supports inheritance.

## Method overriding

Method overriding allows a subclass to replace (*override*) the functionality of a method defined in a superclass. Since JavaScript allows declaring methods on an object as well as on its prototype, overriding a method becomes really simple. The following code sample (see the file `overriding.js`) declares the method `addSubordinate()` on the prototype of the `Person` object, but then the object `p1` overrides this method.

```
function Person(name, title){
```

```

    this.name=name;
    this.title=title;
    this.subordinates=[];
}

Person.prototype.addSubordinate=function(person){

    this.subordinates.push(person);
    console.log("I'm in addSubordinate on prototype " + this);
}

var p1=new Person("Joe", "President");

p1.addSubordinate=function(person){

    this.subordinates.push(person);
    console.log("I'm in addSubordinate in object " + this);
}

var p2 = new Person("Mary", "Manager")

p1.addSubordinate(p2);

```

Running the above code prints only one line: “I’m in addSubordinate in object [object Object]”. This proves that the method `addSubordinate()` on the prototype level is overridden. We can also improve this example by overriding the method `toString()` on the `Person` object. Just add the following fragment to the prior to instantiating `p1`.

```

Person.prototype.toString=function(){
    return "name:" + this.name +" title:" + this.title;
}

```

Now the code prints “I’m in addSubordinate in object name:Joe, title:President”. Overriding the method `toString()` on objects is a common practice as it gives a textual representation of your objects.

## Scope or who’s this?

You are about to read one of the most confusing sections in this book. The confusion is caused by some inconsistencies in JavaScript design and implementations by various browsers. Do you know what will happen if you’ll remove the keywords `this` from the `toString()` method from previous section? You’ll get an error - the variable `title` is not defined. Without the keyword `this` the JavaScript engine tries to find the variable `title` in the global namespace. Declaring and initializing the variable `title` outside of the `Person` declaration get rid of this error, but this is not what we want to do. Misunderstanding of the current scope can lead to difficult to debug errors.



Interestingly enough replacing `this.name` with `name` doesn't generate an error, but rather initializes the variable `name` with an empty string. Although `name` is not an officially reserved JavaScript keyword, there are articles in the blogosphere that don't recommend using the word `name` as a variable name. Keep [this list of reserved words](#) handy to avoid running into an unpredictable behavior.

Let's consider several examples that will illustrate the meaning of `this` variable in JavaScript. The code sample below defines an object `myTaxObject` and calls its method `doTaxes()`. Notice two variables with the same name `taxDeduction` - one of them has global scope and another belongs to `myTaxObject`. This little script is titled `ThisMafia.js`, and it was written for mafia and will apply some under the table deduction for the people who belong to Cosa Nostra.

```
var taxDeduction=300;      // global variable

var myTaxObject = {

    taxDeduction: 400,      // object property

    doTaxes: function() {
        this.taxDeduction += 100;

        var mafiaSpecial= function(){
            console.log( "Will deduct " + this.taxDeduction);
        }

        mafiaSpecial(); // invoking as a function
    }
}

myTaxObject.doTaxes(); //invoking method doTaxes
```

This code fragment illustrates the use of *nested functions*. The object method `doTaxes()` has a nested function `mafiaSpecial()`, which is not visible from outside of the `myTaxObject`, but it can be certainly invoked inside `doTaxes()`. What number do you think this code will print after the words "Will deduct "? Will it print three, four, or five hundred? Run this code in Firebug, Chrome Developer Tools or any other way and you'll see that it'll print 300!

But this doesn't sound right, does it? The problem is that in JavaScript the context where the function executes depends on the way it was invoked. In this case the function `mafiaSpecial()` was invoked as a function (not a method) without specifying the object it should apply to, and JavaScript makes it operate in the global object, hence the global variable `taxDeduction` having the value of 300 is being used. So in expression `this.taxDeduction` the variable `this` means global unless the code is operated in the strict mode.



ECMAScript 5 introduced a restricted version of JavaScript called *strict mode*, which among other things places stricter requirements to variable declarations and scope identification. Adding “use strict” as the first statement of the method `doTax()` will make the context *undefined*, and it’ll print the error “this is undefined” and not 300. You can read about the strict mode at [Mozilla’s developers site](#).

Let’s make a slight change to this example and take to control what `this` represents. When the object `myTaxObject` was instantiated its own `this` reference was created. The following code fragment stores this reference in additional variable `thisOfMyTaxObject` changes the game and the expression `thisOfMyTaxObject.taxDeduction` evaluates to 500.

```
var taxDeduction=300;      // global variable

var myTaxObject = {

    taxDeduction: 400,    // object property

    doTaxes: function() {
        var thisOfMyTaxObject=this;
        this.taxDeduction += 100;

        var mafiaSpecial= function(){
            console.log( "Will deduct " + thisOfMyTaxObject.taxDeduction);
        }

        mafiaSpecial(); // invoking as a function
    }
}

myTaxObject.doTaxes(); //invoking method doTaxes
```

You’ll see a different way of running a function in the context of the specified object using special functions `call()` and `apply()`. But for now consider one more attempt to invoke `mafiaSpecial()` shown in the following example that uses `this.mafiaSpecial()` notation.

```
var taxDeduction=300;      // global variable

var myTaxObject = {

    taxDeduction: 400,    // object property

    doTaxes: function() {
        this.taxDeduction += 100;

        var mafiaSpecial= function(){
            console.log( "Will deduct " + this.taxDeduction);
        }

        mafiaSpecial(); // invoking as a function
    }
}
```

```

        }
    }

    this.mafiaSpecial(); // trying to apply object's scope
}

myTaxObject.doTaxes(); //invoking method doTaxes

```

Run the above code and it'll give you the error "TypeError: this.mafiaSpecial is not a function" and rightly so. Take a closer look at the object `myTaxObject` represented by the variable `this`. The `myTaxObject` has only two properties: `taxDeduction` and `doTaxes`. The function `mafiaSpecial` is hidden inside the method `doTaxes` and can't be accessed via `this`.

After learning how to hide a function inside an object, let's see how to do something quite opposite - allowing an external method to run inside the context of an object.

## Call and Apply

Visualize the International Space Station, and add to the picture an image of a approaching space shuttle. After attaching to the docking bay of the station the shuttle's crew performs some functions on the station (a.k.a. object) and then flies to another object or back to Earth. What it has to do with JavaScript? It can serve as an analogy for creating a JavaScript function that can operate in the scope of any arbitrary object. For this purpose JavaScript offers two special functions: `call()` or `apply()`. Both `call()` and `apply()` can invoke any function on any object. The only difference between them is that `apply()` passes parameters to a function as an array, while `call()` uses a comma-separated list.



Every function in JavaScript is an instance of the `Function` object. Both `call()` and `apply()` are defined in `Function`.

For example, a function `calcStudentDeduction(income, numStudents)` can be invoked in a context of a given object using either `call()` or `apply()`. Note that with `call()` parameters have to be listed explicitly, while with `apply` parameters are given as an array:

```

calcStudentDeduction.call(myTaxObject, 50000, 2);

calcStudentDeduction.apply(myTaxObject, [50000, 2]);

```

In the above example the instance of 'myTaxObject' can be referred as `this` from within the function `calcStudentDeduction()` even though this is a function and not a method. The last example from the previous section can be re-written to invoke `mafiaSpe`

`cial()`. The following code will ensure that `mafiaSpecial()` has `this` pointing to 'myTaxObject' and will print on the console "Will deduct 500".

```
var taxDeduction=300;      // global variable

var myTaxObject = {

    taxDeduction: 400,

    doTaxes: function() {
        this.taxDeduction += 100;

        var mafiaSpecial = function(){
            console.log( "Will deduct " + this.taxDeduction);
        }

        mafiaSpecial.call(this); // passing context to a function
    }
}

myTaxObject.doTaxes();
```

## Callbacks

Can you program without using `call()` and `apply()`? Sure you can, but JavaScript allows you to easily create callbacks. The callback mechanism lets you pass the code of one function as a parameter to another function for execution in the latter function's context. This is a very useful feature of the language. Imagine an object with a method `processData()`. Depending on the business logic you can pass to this method (as an argument) different functions that will do actual data processing - these are callbacks.

Another example of callbacks is event handlers. If a user clicks on this button here's the name of the handler function to call:

```
'myButton.addEventListener("click", myFunctionHandler);'
```

It's important to understand that *you don't not immediately call* the function `myFunctionHandler` here - you are just registering it as the function argument. If the user clicks on `myButton` then the code of the callback `myFunctionHandler` will be given to the object `myButton` and will be invoked in the context of the `myButton` object. The functions `call()` and `apply()` exist exactly for this purpose.

Let's consider an example when you need to write a function that will take two arguments: an array containing preliminary tax data and a callback function, which will be applied to each element of this array. The following code sample (`Callback.js`) creates `myTaxObject` that has two properties: `taxDeduction` and the `applyDeduction`. The latter is a method with two parameters:

```

var myTaxObject = {
    taxDeduction: 400, // state-specific deduction

    // this function takes an array and callback as parameters
    applyDeduction: function(someArray, someCallBackFunction){

        for (var i = 0; i < someArray.length; i++){

            // Invoke the callback
            someCallBackFunction.call(this, someArray[i]);
        }
    }

    // array
    var preliminaryTaxes=[1000, 2000, 3000];

    // tax handler function
    var taxHandler=function(currentTax){
        console.log("Hello from callback. Your final tax is " +
                    (currentTax - this.taxDeduction));
    }

    // invoking applyDeduction passing an array and callback
    myTaxObject.applyDeduction(preliminaryTaxes, taxHandler);
}

```

The above code invokes `applyDeduction()` passing it the array `preliminaryTaxes` and the callback function `taxHandler` that takes the `currentTax` and subtracts `this.taxDeduction`. By the time this callback will be applied to each element of the array the value of `this` will be known and this code will print the following:

```

Hello from callback. Your final tax is 600
Hello from callback. Your final tax is 1600
Hello from callback. Your final tax is 2600

```

You may be wondering, why passing the function to another object if we could take an array, subtract 400 from each of its elements and be done with it? The solution with callbacks gives you an ability to make the decision on what function to call during the runtime and call it only when a certain event happens. Callbacks allow you to do *asynchronous processing*. For example, you make an asynchronous request to a server and register the callback to be invoked if a result comes back. The code is not blocked and doesn't wait until the server response is ready. Here's an example from AJAX: `request.onreadystatechange=myHandler`. You register `myHandler` callback but not immediately call it. JavaScript functions are objects, so get used to the fact that you can pass them around as you'd be passing any objects.

# Hoisting

A variable scope depends on where it was declared. You already had a chance to see that a variable declared inside a function with the keyword `var` is visible only inside this function and any function declared within it. Some programming languages allow to narrow down the scope even further. For example, in Java declaring a variable inside any block of code surrounded with curly braces makes it visible only inside such a block. In JavaScript it works differently. No matter where in the function you declared the variable its declaration will be *hoisted* to the top of the function, and you can use this variable anywhere inside the function.

## Hoisting Variables

The following code snippet will print 5 even though the variable `b` has been declared inside the `if`-statement. Its declaration has been hoisted to the top:

```
function test () {
    var a=1;

    if(a>0) {
        var b = 5;
    }
    console.log(b);

}

test();
```

Let's make a slight change to the above code to separate the variable declaration and initialization. The following code has two `console.log(b)` statements. The first one will output `undefined` and the second will print 5 just as in the previous example.

```
function test () {
    var a=1;

    console.log(b); // b is visible, but not initialized

    if(a>0) {
        var b;
    }

    b=5;

    console.log(b); // b is visible and initialized
}

test();
```

Due to hoisting, JavaScript doesn't complain when the first `console.log(b)` is invoked. It knows about the variable `b`, but its value is `undefined` just yet. By the time the second

`console.log(b)` is called, the variable `b` was initialized with the value of 5. Just remember that hoisting just applies to variable declaration and doesn't interfere with your code when it comes to initialization.

## Hoisting Functions

JavaScript function declarations are hoisted too, and this is illustrated in the following code sample.

```
function test () {
    var a=1;

    if(a>0) {
        var b;
    }

    b=5;

    printB();

    function printB(){
        console.log(b);
    }
}

test();
```

This code will print 5. We can call the function `printB()` here because its declaration was hoisted to the top. But the situation changes if instead of function declaration we'll use the function expression. The following code will give you an error "PrintB is not a function".

```
function test () {
    var a=1;

    if(a>0) {
        var b;
    }

    b=5;

    printB();

    var printB = function(){
        console.log(b);
    }
}

test();
```

Notice that it the error doesn't complain about `printB` being undefined cause the variable declaration was hoisted, but since the function expression wasn't the JavaScript engine doesn't know yet that `printB` will become a function really soon. Anyway, moving the invocation line `printB()` to the bottom of the function `test()` cures this issue.



Function expressions are not being hoisted, but the variables they are assigned to (if any) are being hoisted.

## Function properties

Functions as any other objects can have properties. You can attach any properties to a `Function` object and their values can be used by all instances of this object. Static variables in programming languages with the classical inheritance is the closest analogy to function properties in JavaScript.

Let's consider an example of a constructor function `Tax`. An accounting program can create multiple instances if `Tax` - one per person. Say this program will be used in a Florida neighborhood with predominantly Spanish speaking people. The following code (see the file `FunctionProperties.js`) illustrates the case when the method `doTax()` can be called with or without parameters.

```
function Tax(income, dependents){
    this.income=income;           // instance variable
    this.dependents=dependents;   // instance variable

    this.doTax = function calcTax(state, language){
        if(!(state && language)){ // ①
            console.log("Income: " + this.income + " Dependents: " + this.dependents
            + " State: " + Tax.defaults.state + " language:" + Tax.defaults.language);
        } else{                   // ②
            console.log("Income: " + this.income + " Dependents: " + this.dependents
            + " State: " + state + " language:" + language);
        }
    }
}

Tax.defaults={                // ③
    state:"FL",
    language:"Spanish"
};

// Creating 2 Tax objects
var t1 = new Tax(50000, 3);
t1.doTax();                  // ④
```

```
var t2 = new Tax(68000, 1);
t2.doTax("NY", "English"); // 5
```

- ① No state and language were given to the method `doTax()`
- ② The state and language were provided as `doTax()` parameters
- ③ Assigning the object with two properties as a `defaults` property on `Tax`. The property `default` is not instance specific, which makes it static.
- ④ Invoking `doTax()` without parameters - use `defaults`
- ⑤ Invoking `doTax()` with parameters

This program will produce the following output:

```
Income: 50000 Dependents: 3 State: FL language:Spanish
Income: 68000 Dependents: 1 State: NY language:English
```

You can add as many properties to the constructor function as needed. For example, to count the number of instances of the `Tax` object just add one more property `Tax.counter=0;`. Now add to the `Tax` function something like `console.log(Tax.counter++);` and you'll see that the counter increments on each instance creation.



If multiple instances of a function object need to access certain HTML elements of the DOM, add references to these elements as function properties so objects can reuse them instead of traversing the DOM (it's slow) from each instance.

## Closures

### A Formal Definition Attempt

A *closure* is one of those terms that are easier explained by examples. Closures have formal definitions, which are not very helpful for the first timers. Here's the [definition of a closure from Wikipedia](#):

*"In programming languages, a closure (also lexical closure or function closure) is a function or reference to a function together with a referencing environment—a table storing a reference to each of the non-local variables (also called free variables or upvalues) of that function. A closure—unlike a plain function pointer—allows a function to access those non-local variables even when invoked outside its immediate lexical scope."*

It's not a very helpful definition, is it? Let's try to give a better one. Imagine a function that contains a private variable, and a nested function. Is it possible to invoke the nested

function from the outside of the outer one? And if it's possible, what this inner function knows about its surroundings?

Larry Ullman, gives the following definition in his book "Modern Java Script": "Closure is a function call with memory". We can offer you our version of what a closure is: "Closure is a function call with strings attached".

## Why do we Need Closures?

In classical object-oriented languages you create an object with a certain state and behavior and can pass it to a method of another object for further processing. In JavaScript you can pass even a function to some object's method for further processing. But what if a function also need to remember the state (the values of external variables) of the context where the function was defined?

Think of a closure as a function that remembers state - it's just a special type of object that can be passed between objects and use certain variables that didn't seem to be defined in the function's code. But there existed in the context where the function was defined.

==Closures by Example

Now it's time for the explanation of these mysterious definitions, and we'll do it by example. Consider the following code (see closure1.js) that is yet another example of implementing the tax collection functionality.

```
(function (){                                // this is an anonymous function expression
    var taxDeduction = 500; // private context to remember
    //exposed closure
    this.doTaxes=function(income, customerName) {
        var yourTax;
        if (customerName !== "Tony Soprano"){
            yourTax = income*0.05 - taxDeduction;
        } else{
            yourTax = mafiaSpecial(income);
        }
        console.log("Dear " + customerName + ", your tax is "+ yourTax);
        return yourTax;
    }
    //private function
    function mafiaSpecial(income){
        return income*0.05 - taxDeduction*2;
    }
}
```

```
})(); // Self-invoked function

// The closure remembers its context with taxDeduction=500
doTaxes(100000, "John Smith");
doTaxes(100000, "Tony Soprano");

mafiaSpecial(); // throws an error - this function is private
```

First, a self-invoking function will create an anonymous instance of an object in the global scope. It contains a private variable `taxDeduction`, a public method `doTaxes()`, and a private method `mafiaSpecial()`. Just by the virtue of declaring `doTaxes` on this object, this method becomes exposed to the current scope, which is global in this example.

After that we call the method `doTaxes()` twice. Note that the function `doTaxes()` uses the variable `taxDeduction` that was never declared there. But when `doTaxes` was initially declared, the variable `taxDeduction` with a value of 500 was already there. So the internal function “remembers” the context (the neighborhood) where it was declared and can use it for its calculations.

The algorithm of tax calculations makes `doTaxes()` calls the function `mafiaSpecial()` if the customer’s name is “Tony Soprano”. The function `mafiaSpecial()` is not visible from outside, but for insiders like `doTaxes()` it’s available. Here’s what the above code example will print on the console:

```
Dear John Smith, your tax is 4500
Dear Tony Soprano, your tax is 4000
Uncaught ReferenceError: mafiaSpecial is not defined
```

The [Figure A-16](#) shows the screenshot taken when `doTaxes()` hit the breakpoint inside `doTaxes` - note the right panel that shows what’s visible in the Closure scope.

```

closure1.js
1 (function (){           // this is an anonymous function expression
2   var taxDeduction = 500; // private context to remember
3
4   //exposed closure
5   this.doTaxes=function(income, customerName) {
6
7     var yourTax;
8
9     if (customerName != "Tony Soprano"){
10       yourTax = income*0.05 - taxDeduction;
11     } else{
12       yourTax = mafiaSpecial(income);
13     }
14
15     console.log(" Dear " + customerName + ", your tax is "+ yourTax);
16     return yourTax;
17   }
18
19   //private function
20   function mafiaSpecial(income){
21     return income*0.05 - taxDeduction*2;
22   }
23
24 })();
25 // Self-invoked function
26
27 // calling doTaxes() in the global scope.
28 doTaxes(100000, "John Smith"); // The closure remembers its context: taxDeduction=500
29
30 doTaxes(100000, "Tony Soprano");
31 mafiaSpecial();           // an error - this function is private
32

```

Figure A-16. Closure view in Chrome's Developer Tools.



JavaScript doesn't give you an explicit way to mark a variable as private. By using closures you can get the same level of data hiding that you get from private variables in other languages. In the example above the variable `taxDeduction` is local for the object enclosed in the outermost parentheses and can't be accessed from outside. But `taxDeduction` can be visible from the object's functions `doTaxes` and `mafiaSpecial`.

Figure A-17 gives yet another visual representation of our above code sample. The self-invoked anonymous function is shown as a cloud that exposes only one thing to the rest of the world: the closure `doTaxes`.

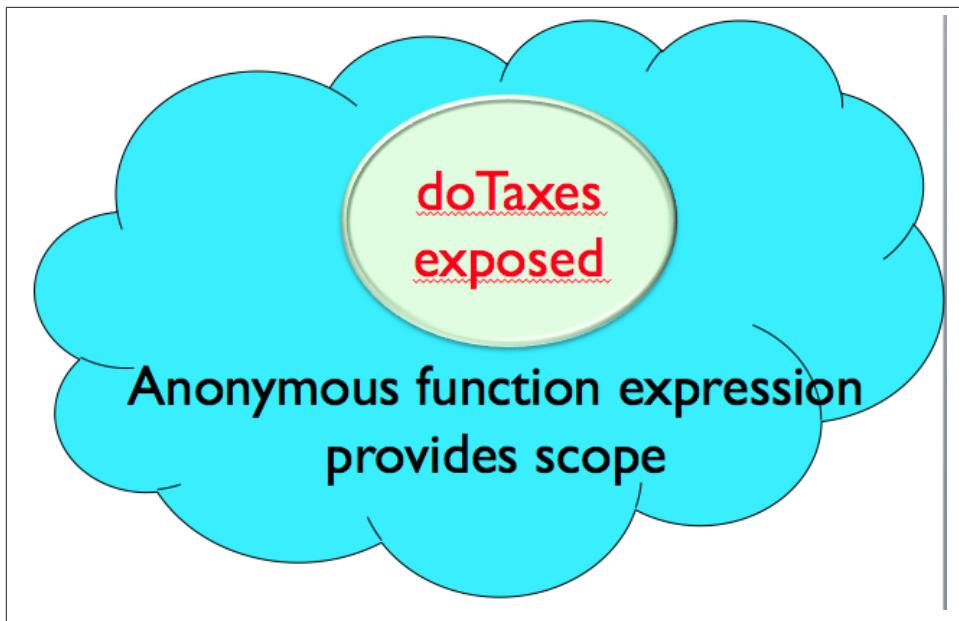


Figure A-17. Closure doTaxes

Let's consider a couple of more cases of returning a closure to the outside world so it can be invoked later. If the previous code sample was exposing the closure by using `this.taxes` notation, the next two examples will simply return the code of the closure using the `return` statement. The code below declares (see `closure3.js`) a constructor function `Person`, adds a function `doTaxes()` to its prototype, and finally creates two instances of the `Person` calling the method `doTaxes()` on each of them.

```
// Constructor function
function Person(name){

    this.name = name;

}

// Declaring a method that returns closure
Person.prototype.doTaxes= function(){

    var taxDeduction = 500;

    //private function
    function mafiaSpecial(income){
        return income*0.05 - taxDeduction*2;
    }

    //the code of this function is returned to the caller
}
```

```

    return function(income) {

        var yourTax;

        if (this.name !== "Tony Soprano"){
            yourTax = income*0.05 - taxDeduction;
        } else{
            yourTax = mafiaSpecial(income);
        }

        console.log( "My dear " + this.name + ", your tax is "+ yourTax);
        return yourTax;
    }
}(); // important parentheses!

//Using closure
var p1 = new Person("John Smith");
var result1 = p1.doTaxes(100000);

var p2 = new Person("Tony Soprano");
var result2 = p2.doTaxes(100000);

```

The calculated taxes in this example are the same as in the previous one: John Smith has to pay \$4500, while Tony Soprano only \$4000. But we used different technique for exposing the closure. We want to make sure that you didn't overlooked the parentheses at the very end of the function expression for `doTaxes`. These parenthesis force the anonymous function to self-invoke itself, it'll run into a `return` statement and will assign the code of the anonymous inner function that takes parameter `income` to the property `doTaxes`. So when the line `var result1 = p1.doTaxes(100000);` calls the closure the variable `result1` will have the value 4500. Remove these important parentheses, and the value of `result1` is not the tax amount, but the the code of the closure itself - the invocation of the closure is not happening.

The following code fragment (see the file `closure2.js`) is yet another example of returning the closure that remembers its context.

```

function prepareTaxes(studentDeductionAmount) {

    return function (income) {           // ①
        return income*0.05 - studentDeductionAmount;
    };
}

var doTaxes = prepareTaxes(300);      // ②
var yourTaxIs = doTaxes(10000);       // ③
console.log("Your tax is " + yourTaxIs); // ④

```

- ➊ When the function `prepareTaxes` is called, it immediately hits the `return` statement and returns the code of the closure to the caller.

- ② After this line is executed, the variable `doTaxes` has the code of the closure, which remembers that `studentDeductionAmount` is equal to 300.
- ③ This is actual invocation of the closure
- ④ the console output is “your tax is 200”

First, the closure is returned to the caller of `prepareTaxes()`, and when the closure will be invoked it'll remember the values defined in its outer context. After looking at this code you may say that there is nothing declared in the closure's outside context! There is - by the time when the closure is created the value of the `studentDeductionAmount` will be known.



Check the quality of your code with the help of the JavaScript code quality tools like [JSLint](#) or [JSHint](#).

## Closures as callbacks

Let's revisit the code from the section Callbacks above. That code has shown how to pass an arbitrary function to another one and invoke it there using `call()`. But if that version of the function `taxHandler` was not aware of the context it was created in, the version below will. If in classical object-oriented languages you'd need to pass a method that knows about its context, you'd need to create an instance of an object that contains the method and the required object-level properties, and then you'd be passing this wrapper-object to another object for processing. But since the closure remembers its context anyway, we can just pass a closure as an object. Compare the code below (see the file `callbackWithClosure.js`) with the code from the Callbacks section.

```
var myTaxObject = {

    // this function takes an array and callback as parameters
    applyDeduction: function(someArray, someCallBackFunction){

        for (var i = 0; i < someArray.length; i++){

            // Invoke the callback
            someCallBackFunction.call(this, someArray[i]);
        }
    }

    // array
    var preliminaryTaxes=[1000, 2000, 3000];
```

```

var taxHandler = function (taxDeduction){

    // tax handler closure
    return function(currentTax){
        console.log("Hello from callback. Your final tax is " +
        (currentTax - taxDeduction));
    };
}

// invoking applyDeduction passing an array and callback-closure
myTaxObject.applyDeduction(preliminaryTaxes, taxHandler(200));

```

The last line of the above example calls `taxHandler(200)`, which creates a closure that's being passed as a callback to the method `applyDeduction()`. Even though this closure is executed in the context of `myTaxObject`, it remembers that tax deduction is 200.

## Mixins

The need to extend capabilities of objects can be fulfilled by inheritance, but this is not the only way of adding behavior to objects. In this section you'll see an example of something that would not be possible in the object-oriented languages like Java or C#, which don't support multiple inheritance. JavaScript allows taking a piece of code and *mix it into any object* regardless of what its inheritance chain is. *Mixin* is a reusable code fragment that an object can borrow without the need to use inheritance. We'll illustrate this concept by example.

In the next code fragment we'll define a function expression and will assign it to a variable named `Tax`. This is a closure that includes the function `calcTax()` that knows the values of `income` and `state`. There is also an independent mixin `TaxMixin` with a couple of functions `mafiaSpecial()` and `drugCartelSpecial()`. We want to blend in this mixin into `Tax`. After this is done, the `Tax` object will have its original functionality, e.g. `calcTax()`, as well as a new "mafia and drug cartel flavors". The code below is located in the file `mixins.js`.

```

// Defining a function expression
var Tax = function(income, state){
    this.income=income;
    this.state=state;

    this.calcTax=function(){
        var tax=income*0.05;
        console.log("Your calculated tax is " + tax)
        return tax;
    }
};

```

```

// Defining a mixin
var TaxMixin = function () {};

TaxMixin.prototype = {

    mafiaSpecial: function(originalTax){
        console.log("Mafia special:" + (originalTax - 1000));
    },

    drugCartelSpecial: function(originalTax){
        console.log("Drug Cartel special:" + (originalTax - 3000));
    }

};

// this function can blend TaxMixin into Tax
function blend( mainDish, spices ) {

    for ( var methodName in spices.prototype ) {
        mainDish.prototype[methodName] = spices.prototype[methodName];
    }
}

// Blend the spices with the main dish
blend( Tax, TaxMixin );

// Create an instant of Tax
var t = new Tax(50000, "NY");

var rawTax = t.calcTax();

// invoke a freshly blended method
t.mafiaSpecial(rawTax);

```

The function `blend()` loops through the code of the `TaxMixin` and copies all its properties into `Tax`. After the function `blend()` is finished, you can call on the `Tax` instance newly acquired methods `mafiaSpecial()` and `drugCartelSpecial()`.

Mixins can be useful if you want to provide a specific feature to a number of different objects without changing their inheritance chains. The other use case is if you want to prepare a bunch of small code fragments (think `spices`) and add any combination of them to the various objects (`dishes`) as needed. Mixins give you a lot of flexibility in what you can achieve with the minimum code, but they may decrease the readability of your code.

If you've read this far, you should have a good understanding of the syntax of the JavaScript language. Studying the code samples provided in this appendix has one extra benefit: now you can apply for a job as a tax accountant in a mafia near you.

# JavaScript in the Web Browser

After learning all these facts and techniques about the language you might be eager to see “the real-world use of JavaScript”. Slowly but surely a Web browser becomes the leading platform for development of the user interface. The vast majority today’s JavaScript programs primarily manipulate HTML elements of Web pages. In this section we’ll be doing exactly this – applying JavaScript code to modify the content or style of HTML elements.

## The Document Object Model (DOM)

DOM stands for Document Object Model. It’s an object representing the hierarchy of HTML elements of a Web page. Every element of the HTML document is loaded into DOM. Each DOM element has a reference to its children and siblings. When DOM was invented, the Web pages were simple and static. DOM was not meant to be an object actively accessed by the code. This is the reason that on some of the heavily populated Web pages manipulating of DOM elements can be slow. Most likely DOM is the main target for anyone who’s trying to optimize the performance of a Web page.



If your Web page is slow, analyze it with [YSlow](#), the tool built based on the Yahoo! rules for high performance Web sites. Also, you can minimize and obfuscate your JavaScript code with the help of [JavaScript Compressor](#).

When a Web Browser is receiving the content it performs the following activities:

- Adding arriving HTML elements to DOM and laying out the content of the Web pages
- Rendering of the UI
- Running JavaScript that was included in the HTML
- Processing events

The amount of time spent on each of these activities varies depending the content of the page.



If you are interested in learning how the browsers work in detail, read an excellent writeup titled “How Browsers Work: Behind The Scenes of Modern Web Browsers” at <http://bit.ly/how-browsers-work> [<http://bit.ly/how-browsers-work>].

Let's consider the operations your application needs to be able to perform inside the Web page:

- Programmatically finding the required element by id, type, or a CSS class
- Changing styles of the elements (show, hide, apply fonts and colors et al.)
- Processing events that may happen to HTML elements (click, mouseover and the like)
- Dynamically adding or removing HTML elements from the page or changing their contents
- Communicating with the server side, e.g. submitting forms or making AJAX requests for some data from the server

Now you'll see some code samples illustrating the use of JavaScript for the operations listed above. Even if you'll be using one of the popular JavaScript frameworks, your program will be performing similar operations applying the syntax prescribed by your framework of choice. So let's learn how it can be done.

## Working with DOM

If you want to change the appearance of an HTML page, you need to manipulate with the DOM elements. Older Web applications were preparing the HTML content on the server side. For example, a server-side Java servlet would compose and send to the client HTML whenever the application logic required to change the appearance of the UI. The current trend is different - the client's code takes care of the UI rendering, and only the data go back and forth between the client and the server. You'll see how this works in more detail in Chapter 2 that explains the use of AJAX and JSON.

Earlier in this appendix we were talking about the global namespace where all JavaScript objects live unless they were declared with `var` inside the functions. If the JavaScript code is running in a Web browser, this global namespace is represented by a special variable `window`. It's an implicit variable and you don't have to use it in your code, but whenever we say that a variable is global, we mean that it's exists on the `window` object. For example, the code below will print "123 Main Street" twice:

```
var address = "123 Main Street";  
  
console.log(address);  
console.log(window.address);
```

The `window` object has a number of useful properties like `cookie`, `location`, `parent`, `document` and others. The variable `document` points at the root of the DOM hierarchy. Pretty often your JavaScript code would find an element in the DOM first, and then it could read or modify its content. [Figure A-18](#) is a snapshot from Firebug showing the fragment of a DOM of a simple Web page mixins.html.

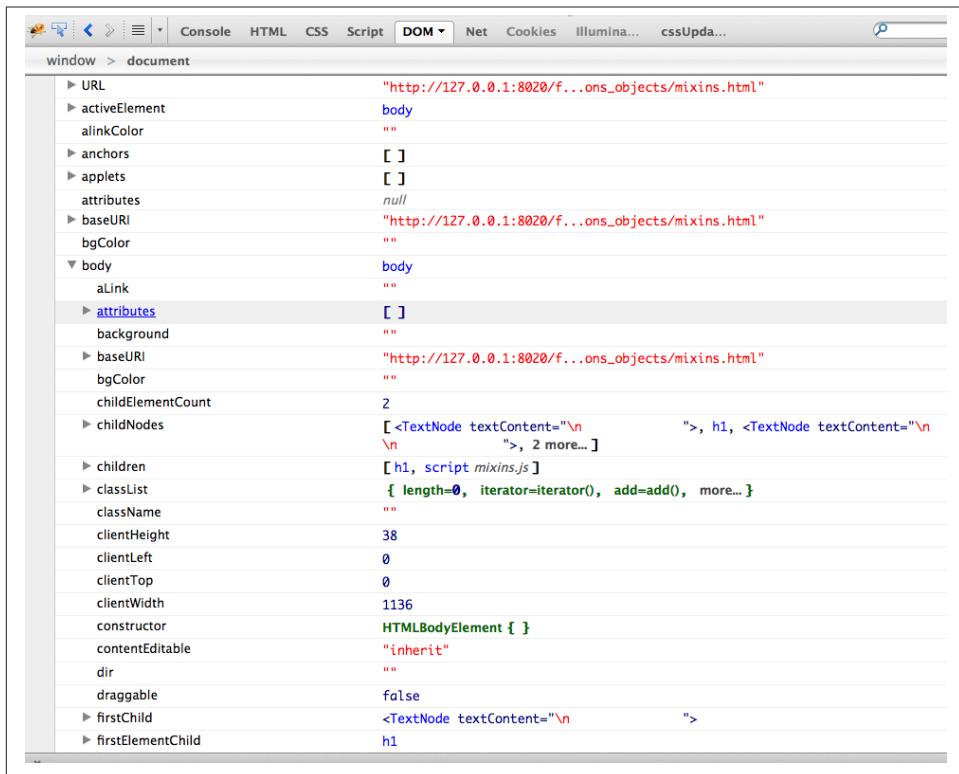


Figure A-18. Firebug's representation of DOM

Below are some of the methods that exist on the `document` object:

- `document.write(text)` – adds the specified text to the DOM. Careless usage of the method `write()` can result in unpredictable results if after changing the DOM the HTML content is still arriving.
- `document.getElementById(id)` – get a reference to the HTML element by its unique identifier
- `document.getElementsByTagName(tname)` - get a reference to one or more elements by tag names, e.g. get a reference to all `<div>` elements.
- `document.getElementsByName(name)` - get a reference to all elements that have requested value in their `name` attribute.
- `document.getElementsByClassName(className)` – get a reference to all elements to use specified CSS class(es), like `document.getElementsByClassName('red text-left')`.

- `document.querySelector(cssSelector)` – Find the first element that matches provided CSS selector string. It comes handy if you want to specify more complex queries than just a class name, e.g. `document.querySelector("style[type='text-left'])");`
- `document.querySelectorAll(cssSelector)` – Find all elements that match provided CSS selector string.

The next code sample contains the HTML `<span>` element that has an id `emp`. Initially it contains ellipsis, but when the user enters the name in the input text field, the JavaScript code will find the reference to this `<span>` element and will replace the ellipsis with the content of the input text field.

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
    </head>

    <body>
        <h2>Selecting DOM elements</h2>

        <p>
            The employee of the month is <span id="emp">...</span>
        <br>
        <input type="button" value="Change the span value"
               onclick="setEmployeeOfTheMonth()"/>
        Enter your name <input type="text" id="theName" />
    </p>

    <script>
        function setEmployeeOfTheMonth(){

            var mySpan = document.getElementById("emp");

            var empName = document.getElementsByTagName("input")[1];

            mySpan.firstChild.nodeValue = empName.value;

        }
    </script>

    </body>
</html>

```

Note the input field of type `button`, which includes the `onclick` property that corresponds to the `click` event. When the user clicks on the button, the browser dispatched `click` event, and calls the JavaScript function `setEmployeeOfTheMonth()`. The latter queries the DOM and finds the reference to the `emp` by calling the method `getElementById()`. After that, the method `getElementsByName()` is called trying to find all the

references to the HTML `<input>` elements. This methods returns an array cause there could be more than one element with the same tag name on a page, which explains the use of array notation. The first `<input>` element is a button and the second is the text field we're interested in. Remember that arrays in JavaScript have zero-based indexes. **Figure A-19** shows the Web page after the user entered the name *Mary* and pressed the button.

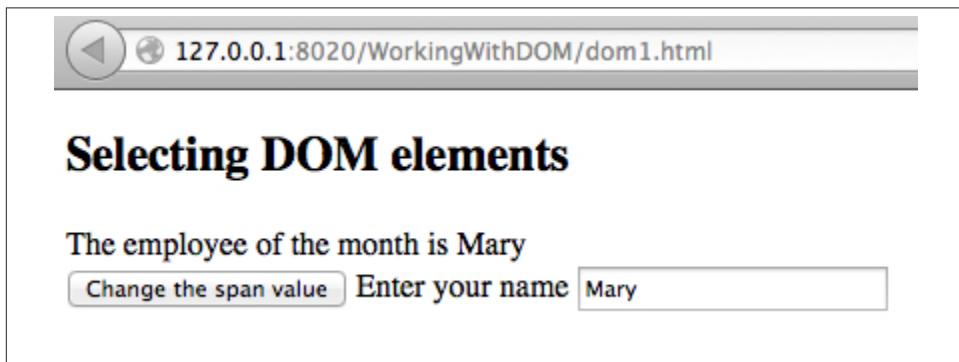


Figure A-19. Changing the content of the HTML `<span>` element

While manipulating the content of your Web page you may need to traverse the DOM tree. The code example below shows you an HTML document that includes JavaScript that walks the DOM and prints the name of each node. If a node has children, the recursive function `walkTheDOM()` will visit each child.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
  </head>

  <body>
    <h1>WalkTheDom.html</h1>

    <p>
      Enter your name: <input type="text"
        name="customerName" id="custName" />
    </p>

    <input type="button" value="Walk the DOM"
      onclick="walkTheDOM(document.body, processNode)" />

    <script>
      function walkTheDOM(node, processNode){

        processNode(node)
      }
    </script>
  </body>
</html>
```

```

        node = node.firstChild;

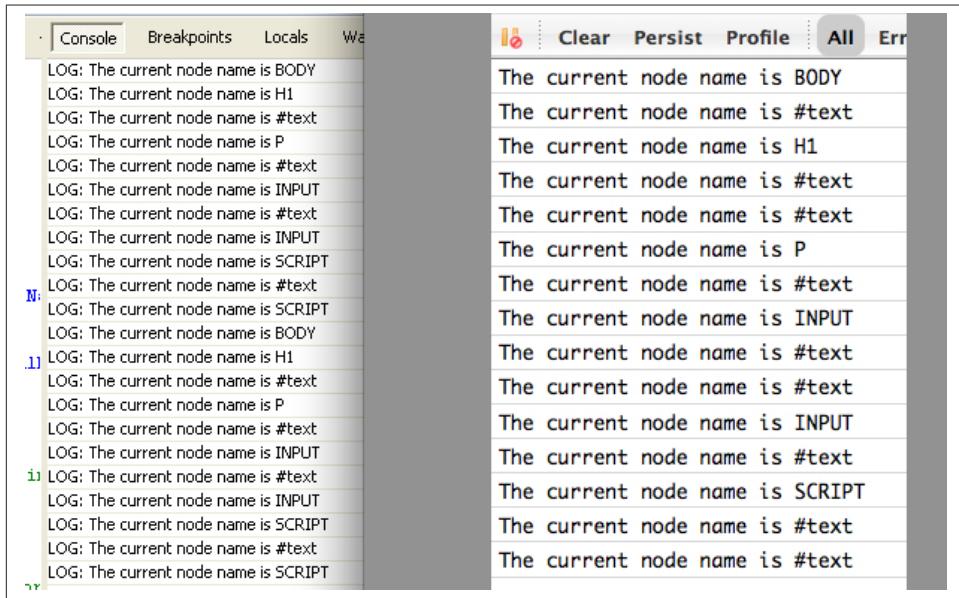
        while(node){
            // call walkTheDOM recursively for each child
            walkTheDOM(node,processNode);
            node = node.nextSibling;
        }
    }

    function processNode(node){
        // the real code for node processing goes here

        console.log("The current node name is "+ node.nodeName);
    }
</script>
</body>
</html>

```

Our function `processNode()` just prints the name of the current node, but you could implement any code that your Web application requires. Run this code in different browsers and check the output on the JavaScript console. [Figure A-20](#) depicts two snapshots taken in the F12 Developer Tools in Internet Explorer (left) and Firebug running in Firefox (right).



*Figure A-20. Traversing the DOM in Firefox*

While some of the output is self-explanatory, there is a number of #text nodes that you won't find in the code sample above. Unfortunately, Web browsers treat white-spaces in differently - some will ignore them, while others will report them as DOM elements. Accordingly, different browsers insert different number of text nodes in the DOM representing whitespaces found in the HTML document. So you'll be better off using one of the JavaScript frameworks for traversing the DOM cross-browser way. For example, jQuery framework's API for DOM traversing is listed at <http://bit.ly/WXj2r2>.

## Styling Web Pages with CSS

CSS stands for Cascading Style Sheets. During the last 15 years several CSS specifications reached the level of Recommendation by W3C: CSS Level 1, 2, and 2.1. The latest CSS Level 3 (a.k.a. CSS3) adds new features to CSS 2.1 module by module, which are listed at <http://www.w3.org/Style/CSS/current-work>.



You can find CSS tutorial as well as tons of other learning resources at [webplatform.org](http://webplatform.org).

You can include CSS into a Web page by linking to separate files using the HTML tag `<link>`, or by in-lining the styles with the tag `<style>`, or using the `style` attribute in HTML element (not recommended). For example, if CSS is located in the file `mystyles.css` in the folder `css` add the following tag to HTML:

```
<link rel="stylesheet" type="text/css" href="css/mystyles.css" media="all">
```

The `<link>` tag allows specifying the media where specific css file has to be used. For example, you can have one CSS file for smartphones and another one for tablets. We'll discuss this in detail in the section on media queries in Chapter 10.

You should put this tag in the section of your HTML before any JavaScript code to make sure that styles are loaded before the content of the Web page.

Placing the `@import` attribute inside the `<style>` tag allows to include styles located elsewhere:

```
<style>
  @import url (css/contactus.css)
</style>
```

What's the best way of including CSS in HTML? We recommend using CSS files. Keeping CSS in files separately from HTML and JavaScript makes the code more readable and reusable. You may argue that if your Web site consists of many files, the Web browser will have to make multiple round trips to your server just to load all resources required by the HTML document, which can worsen the responsiveness of your Web application.

But usually all files are merged into one before deploying Web application in QA or production servers.

HTML documents are often prettyfied by using CSS class selectors, and you can switch them programmatically with JavaScript. Imagine that a `<style>` section has the following definition of two class selectors `badStyle` and `niceStyle`:

```
<style>
    .badStyle{
        font-family: Verdana;
        font-size:small;
        color:navy;
        background-color:red;
    }

    .niceStyle{
        font-family: Verdana;
        font-size:large;
        font-style:italic;
        color:gray;
        background-color:green;
    }
</style>
```

Any of these class selectors can be used by one or more HTML elements, for example

```
<div id="header" class="badStyle">
    <h1>This is my header</h1>
</div>
```

Imagine that some important event has happened and the appearance the `<div>` styled as `badStyle` should programmatically change to `<niceStyle>`. In this case we need to find the `badStyle` element(s) first and change their style. The method `getElementsByClassName()` returns a set of elements that have the specified class name, and since our HTML has only one such element, the JavaScript will use the element zero from such set:

```
document.getElementsByClassName("badStyle")[0].className="niceStyle";
```

The next example will illustrate adding a new element to the DOM. On click of a button the code below dynamically creates an instance of type `img` and then assigns the location of the image to its `src` element. In a similar way we could have assigned values to any other attributes of the `img` element like `width`, `height`, or `alt`. The method `appendChild()` is applied to the `<body>` container, but it could be any other container that exists on the DOM.

```
<!DOCTYPE html>
<html>
    <head>
```

```

<meta charset="utf-8" />
</head>

<body>
<h2>Employee of the month</h2>
<p>
    <input type="button" value="Show me"
           onclick="setEmployeeOfTheMonth()"/>
</p>

<script>

function setEmployeeOfTheMonth(){

    // Create an image and add it to the <body> element
    var empImage=document.createElement("img");
        empImage.setAttribute('src','resources/images/employee.jpg');
        document.body.appendChild(empImage);
}

</script>
</body>
</html>

```



Some HTML elements like `<div>` or `<span>` contain other elements (children), and if you need to change their content use their property `innerHTML`. For example, to delete the entire content of the document body just do this: `document.body.innerHTML=""`. You can also use the method `appendChild()` as shown in the code sample above.

If you run this example and click on the button “Show me” you’ll see an image of the employee of the month added to the `<body>` section of the HTML document as shown on [Figure A-21](#).

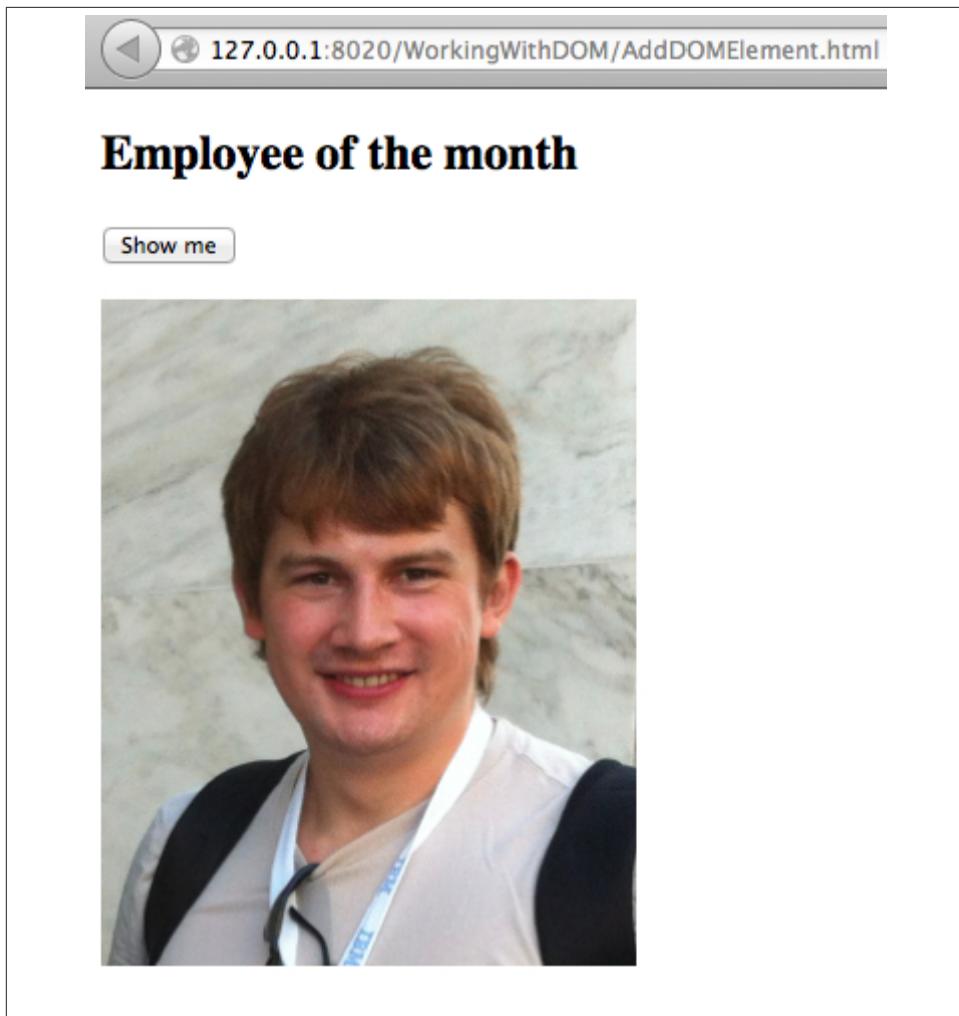


Figure A-21. After clicking the button “Show me”

## DOM Events

Web browser will notify your application when some changes or interactions occur. In such cases the browser will dispatch an appropriate event, for example `load`, `unload`, `mousemove`, `click`, `keydown` etc. When the Web page finished loading the browser will dispatch the `load` event. When the user will click on the button on a Web page the `click` event will be dispatched. A Web developer needs to provide JavaScript code that will react on the events important to the application. The browser events will occur regardless of if you provided the code to handle them or not. It's important to understand some terms related to event processing.

An *event handler* (a.k.a. *event listener*) is a JavaScript code you want to be called as a response to this event. The last code sample from the previous section was processing the `click` event on the button “Show me” as follows: `onclick="setEmployeeOfTheMonth()"`.



Each HTML element has a certain number of predefined *event attributes*, which start with the prefix `on` followed by the name of the event. For example `onclick` is an event attribute that you can use for specifying the handler for the `click` event. You can find out what event attributes are available in the online document titled [Document Object Model Events](#).

The preferred way of adding event listener was introduced in the DOM Level 2 specification back in 2000. You should find the HTML element in the DOM, and then assign the event listener to it by calling the method `addEventListener()` (this is done differently in Internet Explorer below version 9). For example:

```
document.getElementById("myButton").addEventListener("click", setEmployeeOfTheMonth);
```

The advantage of using such programmatic assignment of event listeners is that this can be done for all controls in a central place, for example in a JavaScript function that runs immediately after the Web page completes loading. Another advantage is that you can programmatically remove the event listener if it's not needed any longer by invoking `removeEventListener()`. The following example is a re-write of the last example from the previous section.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
  </head>

  <body>
    <h2>Employee of the month</h2>
    <p>
      <input type="button" value="Show me" id="myButton" /> ❶
    </p>

    <script>
      window.onload=function(){           // ❷
        document.getElementById("myButton").addEventListener("click",
          setEmployeeOfTheMonth);
      }

      function setEmployeeOfTheMonth(){

        // Create an image and add it to the <body> element
        var empImage=document.createElement("img");
```

```

        empImage.setAttribute('src','resources/images/employee.jpg');
        document.body.appendChild(empImage);

        document.getElementById("myButton").removeEventListener("click",
            setEmployeeOfTheMonth); // ❸
    }

</script>
</body>
</html>

```

- ❶ Compare this button with the one from the previous section: the event handler is removed, but it has an ID now.
- ❷ When the Web page completes loading, a `load` event is dispatched and the function attached to the event attribute `onload` assigns the event handler for the button `click` event. Note that we are passing the callback `setEmployeeOfTheMonth` as the second argument of the `addEventListener()`
- ❸ Removing the event listener after the image of the employee of the month has been added. Without this line each click on the button would add to the Web page yet another copy of the same image.

Each event goes through three different phases: *Capture*, *Target*, and *Bubble*. It's easier to explain this concept by example. Imagine that a button is located inside the `<div>`, which is located inside the `<body>` container. When you click on the button, the event travels to the button through all enclosing containers, and this is the capture phase. You can intercept the event at one of these containers even before it reached the button if need be. For example, your application logic may need to prevent the button from being clicked if certain condition occurs.

Then event reaches the button, and it's a target phase. After the event is handled by the button's `click` handler, the event bubbles up through the enclosing containers, and this is the bubble phase. You can create listeners and handle this event after the button finished its processing at the target phase. The next code sample is based on the previous one, but it demonstrates the event processing in all three phases.

Note that if your event handler function is declared with the `event` parameter, it'll receive the `Event` object (not in IE 8), which contains a number of useful parameters. For more information refer to the “Document Object Model Events” online.

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
    </head>

    <body>
        <h2>Employee of the month</h2>

```

```

<div id="myDiv">
    <input type="button" value="Show me" id="myButton"/>
</div>

<script>
    window.onload=function(){
        document.getElementById("myButton").addEventListener("click",
            setEmployeeOfTheMonth);

        document.getElementById("myDiv").addEventListener("click",
            processDivBefore, true); // ❶
        document.getElementById("myButton").addEventListener("click",
            processDivAfter);
    }

    function setEmployeeOfTheMonth(){

        console.log("Got the click event in target phase");

        // Create an image and add it to the <body> element
        var empImage=document.createElement("img");
        empImage.setAttribute('src','resources/images/employee.jpg');
        document.body.appendChild(empImage);

        document.getElementById("myButton").removeEventListener("click",
            setEmployeeOfTheMonth);
    }

    function processDivBefore(evt){
        console.log("Intercepted the click event in capture phase");

        // Cancel the click event so the button won't get it

        // if (evt.preventDefault) evt.preventDefault();
        // if (evt.stopPropagation) evt.stopPropagation(); ❷
    }

    function processDivAfter(){
        console.log("Got the click event in bubble phase");
    }
</script>
</body>
</html>

```

- ❶ We've added two event handler on the `<div>` level. The first one intercepts the event on the capture phase. When the third argument of `addEventListener()` is true, this handler will kick in during capture phase.

- ② If you uncomment these two lines, the default behavior if the `click` event will be cancelled and it won't reach the button at all. Unfortunately, browsers may have different method implementing `preventDefault` functionality hence additional if-statements are needed.

Running the above example will cause the following output in the JavaScript console:

```
Intercepted the click event in capture phase
Got the click event in target phase
Got the click event in bubble phase
```

You can see another example of using intercepting the event during the capture phase in the Donate Section of Chapter 2.



The Microsoft's Web browsers Internet Explorer 8 and below didn't implement the W3C DOM Level 3 event model - they handled events differently. You can read more on the subject at this MSDN article <http://bit.ly/anZZgZ>.

## Summary

This appendix covered the JavaScript language constructs that any professional Web developer should know. A smaller portion of this chapter was illustrating how to combine JavaScript, HTML, and CSS. There are lots of online resources and books that cover just the HTML markup and CSS, and you'll definitely need to spend more time mastering details of the Web tools like Firebug or Google Developer Tools.

Software developers who are coming from strongly-typed compiled languages may have a feeling that their productivity drops with JavaScript. We can recommend several medications for this. First, get familiar with the language called CoffeeScript. As a respected Java developer James Ward put it, "CoffeeScript is *the way to write JavaScript*". This language is very similar to JavaScript and is very easy to learn if you understand the JavaScript syntax, it supports classes and is compiled into JavaScript. Visit [coffeescript.org](http://coffeescript.org) to see the CoffeeScript code snippets and their equivalents in JavaScript.

Another interesting language to learn is Microsoft's TypeScript (it's an open source project). This language is also an extension of JavaScript with added classes, interfaces and inheritance. It also gets compiled into JavaScript and allows developers write strongly-typed code. TypeScript increases productivity of developers because it helps identify lots of error related with incorrect types during the compilation phase. TypeScript implements many constructs from EcmaScript 6 and can be serve as an example of the JavaScript of the future.

Probably the most interesting new programming language is Google's **Dart**. This is a compiled language with all object-oriented features - classes, objects, abstract classes,

inheritance. The compiled code runs inside the VM, and Google supports it in Chrome browser. What about the other browsers? The Web application is deployed as a script that automatically checks if the browser supports Dart. If it does, the compiled code will be sent to the client, otherwise the Dart code will be automatically compiled into JavaScript, and from the browser perspective nothing but a JavaScript engine is required. You can do a server-side programming in Dart too. JetBrains WebStorm, our IDE of choice, supports Coffeescript, Dart, and Typescript.



---

## Appendix B. Selected HTML5 APIs

This appendix is a brief review of selected HTML5 APIs. HTML5 is just a commonly used term for a combination of HTML, JavaScript, CSS, and several new APIs that appeared during the last several years. Five years ago people were using the term *Web 2.0* to define modern looking applications. These days HTML5 is almost a household name, and we'll go along with it. But HTML5 is about the same old development in JavaScript plus latest advances in HTML and CSS.

This appendix is more of an overview of selected APIs that are included in HTML5 specification, namely Web Storage, Application Cache, IndexedDB, localStorage, Web Workers, and History API.



To understand code samples included in this appendix you have to be familiar with JavaScript and some monitoring tools like Chrome Developer Tools. We assume that you are familiar with the materials covered in Appendix A.

### Does your Browser Support HTML5?

The majority of the modern Web browsers already support the current version of **HTML5 specification**, which will become a W3C standard in 2014. The question is if the users of your Web application have a modern browser installed on their device? There are two groups of users that will stick to the outdated browsers for some time:

1. Less technically savvy people may be afraid of installing any new software on their PCs, especially, people of the older generation. *“John, after the last visit of our grandson our computer works even slower than before. Please don’t let him install these new fancy browsers here. I just need my old Internet Explorer, access to Hotmail and Facebook”.*

2. Business users working for large corporations, where all the installations of the software on their PCs is done by a dedicated technical support team. They say, “*We have 50000 PCs in our firm. An upgrade from Internet Explorer version 8 to version 9 is a major undertaking. Internal users work with hundreds of Web applications on a regular basis. They can install whatever browser they want, but if some of these applications won’t work as expected, the users will flood us with support requests we’re not qualified to resolve.* Hence the strategy of using the lowest denominator browser often wins.

Often Web developers need to make both of the above groups of users happy. Take for example online banking - an old couple has to be able to use your Web application from their old PCs otherwise they will transfer their lifetime savings to a different bank which doesn't require, say the latest version of Firefox installed.

Does it mean that enterprise Web developers shouldn't even bother using HTML5 that's not 100% supported? Not at all. This means that a substantial portion of their application's code will be bloated with if-statements figuring out what this specific Web browser supports and providing several solutions that keep your application float in any Web browser. This what makes the job of DHTML developers a lot more difficult than that of, say Java or .Net developers who know exactly the VM where their code will work. If you don't install the Java Runtime of version 1.6 our application won't work. As simple as that. How about asking Java developers writing applications that will work in any runtime released during the last 10 years? No, we're not that nasty.

Do you believe it would be a good idea for Amazon or Facebook to re-write their UI in Java? Of course not unless they want to loose most of their customers who will be scared to death after seeing the message of the 20-step Java installer asking for the access to the internals of their computer. Each author of this book is a Java developer, and we love using Java... on the server side. But when it comes to the consumer facing Web applications there are better than Java choices.

The bottom line is that we have to learn how to develop Web applications that won't require installing any new software on the user's machines. In the Web browsers it's DHTML or in the modern terminology it's HTML5 stack.

In the unfortunate event of needing to support both new and old HTML and CSS implementations you can use [HTML5 Boilerplate](#) that is not a framework, but a template for creating a new HTML project that will support HTML5 and CSS3 elements but will work even in the hostile environments of the older browsers. It's like broadcasting a TV show in HD, but letting the cavemen with the 50-year old black-and-white tubes watching it too.

HTML Boilerplate comes with a simple way to start your project pre-packaged with solutions and workarounds offered by well known gurus in the industry. Make no mis-

take, your code base may be larger than you wanted - for example, the initial CSS starts with 500 lines accommodating the old and new browsers, but it may be your safety net.



Watch [this screencast by Paul Irish](#), a co-creator of HTML5 Boilerplate. You can also read the current version of the [Getting started with HTML5 Boilerplate](#) on Github.

## Handling Differences in Browsers

This appendix is about selected HTML APIs that we find important to understand in Web applications. But before using any of the API's listed here you want to check if the versions of the Web browsers you have to user support these APIs. The Web site <http://caniuse.com> will give you the up-to-date information about all major browsers and their versions that do (or don't) support the API in question. For example, to see which browsers support Worker API visit [caniuse.com](http://caniuse.com).

It's a good practice to include in your code a line that tests if a specific API is supported. For example, if the following if-statement returns false, the Web Worker is not supported and the code should fallback to a single-threaded processing mode:

```
if (window.Worker) {
    // create a Worker instance to execute your
    // script in a separate thread
} else{
    // tough luck, fallback to a single-threaded mode
}
```

In Chapter 1 you'll learn about the feature-detection tool *Modernizr* that allows to programmatically check if any particular HTML5 API is supported by the browser being used.

```
if (Modernizr.Worker) {
    // create a Worker instance to execute your
    // script in a separate thread
}
```

## HTML5 Web Messaging API

[HTML5 Web Messaging](#) allows you to arrange for communication between different Web pages of the same Web application. More officially, it's about "communicating between browsing contexts in HTML documents". Web messaging also allows you to work around the "same domain" policy that would result in security error if a browser's page A has one origin (the combination of URL scheme, host name and port, e.g. <http://myserver.com:8080>) and tries to access property of a page B that was downloaded from

another origin. But with messaging API windows downloaded from different origins can send messages to each other.

## Sending and Receiving Messages

The API is pretty straightforward: if a script in the page `WindowA` has a reference to `WindowB` where you want to send a message, invoke the following method:

```
myWindowB.postMessage(someData, targetOrigin);
```

The object referenced by `myWindowB` will receive an event object with the content of payload `someData` in the event's property `data`. The `targetOrigin` specifies the origin where `myWindowB` was downloaded from.

Specifying a concrete URI of the destination window in `targetOrigin` is the right way to do messaging. This way if a malicious site will try to intercept the message it won't be delivered since the URI specified in `targetOrigin` would be different from the malicious site's URI. But if you're absolutely sure that your application is operating in absolutely safe environment, you can specify "\*" as `targetOrigin`.

Accordingly, `myWindowB` has to define an event handler for processing of this external event `message`, for example:

```
window.addEventListener('message', myEventHandler, false);

function myEventHandler(event){
    console.log(`Received something: ` + event.data);
}
```

## Communicating with an iFrame

Let's consider an example where an HTML Window creates an iFrame and needs to communicate with it. In particular, the iFrame will notify the main window that it has loaded, and the main window will acknowledge receiving of this message.

The iFrame will have two button emulating the case of some trading system with two buttons: Buy and Sell. When the user clicks on one of these iFrame's buttons the main window has to confirm receiving of the buy or sell request. [Figure B-1](#) is a snapshot from a Chrome browser where Developers Tools panel shows the output on the console after the iFrame was loaded and the user clicked on the Buy and Sell buttons.

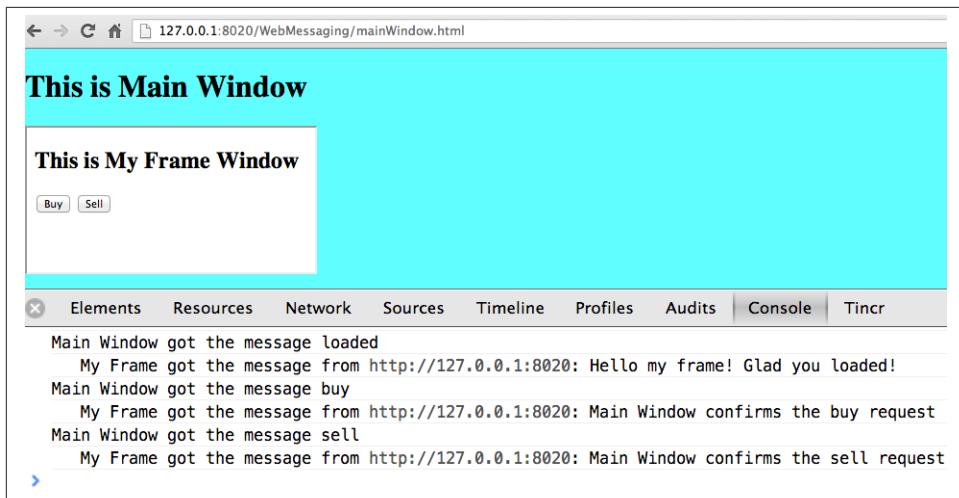


Figure B-1. Message exchange between the window and iFrame

The source code of this example is shown next. It's just two HTML files: mainWindow.html and myFrame.html. Here's the code of mainWindow.html

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>The main Window</title>
</head>

<body bgcolor="cyan">

    <h1>This is Main Window </h1>

    <iframe id="myFrame">
        <p>Some page content goes here</p>
    </iframe>

    <script type="text/javascript">
        var theiFrame;

        function handleMessage(event) {          // ①
            console.log('Main Window got the message ' +
                event.data);

            // Reply to the frame here
            switch (event.data) {           // ②
                case 'loaded':
                    theiFrame.contentWindow.postMessage("Hello my frame! Glad you loaded!", 
                        event.origin); // ③
            }
        }
    </script>
</body>
</html>
```

```

        break;
    case 'buy':
        theiFrame.contentWindow.postMessage("Main Window confirms the buy request ",
                                         event.origin);
        break;
    case 'sell':
        theiFrame.contentWindow.postMessage("Main Window confirms the sell request. ",
                                         event.origin);
        break;
    }
}

window.onload == function() { // ④
    window.addEventListener('message', handleMessage, false);
    theiFrame == document.getElementById('myFrame');
    theiFrame.src == "myFrame.html";
}

</script>

</body>
</html>

```

- ❶ This function is an event handler for messages received from the iFrame window. The main window is the parent of iFrame, and whenever the latter will invoke `parent.postMessage()` this even handler will be engaged.
- ❷ Depending on the content of the message payload (`event.data`) respond back to the sender with acknowledgment. If the payload is `loaded`, this means that the iFrame has finished loading. If it's `buy` or `sell` - this means that the corresponding button in the iFrame has been clicked. As an additional precaution, you can ensure that `event.origin` has the expected URI before even starting processing received events.
- ❸ While this code shows how a window sends a message to an iPrame, you can send messages to any other windows as long as you have a reference to it. For example:

```
var myPopupWindow == window.open(...);
myPopupWindow.postMessage("Hello Popup", "*");
```

- ❹ On load the main window starts listening to the messages from other windows and loads the content of the iFrame.



To implement error processing add a handler for the `window.onerror` property.

The code of the myFrame.html comes next. This frame contains two buttons Buy and Sell, but there is no business logic to buy or sell anything. The role of these buttons is just to deliver the message to the creator of the iFrame that it's time to buy or sell.

```
<!DOCTYPE html>
<html lang="en">

    <body bgcolor="white">

        <h2> This is My Frame Window </h2>

        <button type="buy" onclick="sendToParent('buy')">Buy</button>
        <button type="sell" onclick="sendToParent('sell')">Sell</button>

        <script type="text/javascript">

            var senderOrigin == null;

            function handleMessageInFrame(event) {
                console.log(' My Frame got the message from ' +
                    event.origin +": " + event.data);
                if (senderOrigin === null) senderOrigin == event.origin; // ❶
            }

            window.onload == function(){
                window.addEventListener('message', handleMessageInFrame, false);
                parent.postMessage('loaded', "*"); // ❷
            };

            function sendToParent(action){
                parent.postMessage(action,  senderOrigin); // ❸
            }

        </script>
    </body>
</html>
```

- ❶ When the iFrame receives the first message from the parent, store the reference to the sender's origin.
- ❷ Notify the parent that the iFrame is loaded. The target origin is specified as "\*" here as an illustration of how to send messages without worrying about malicious sites-interceptors - always specify the target URI as it's done in the function `sendToParent()`.
- ❸ Send the message to parent window when the user clicks on Buy or Sell button.

If you need to build a UI of the application from reusable components, applying messaging techniques allows you to create loosely coupled components. Say you've created a window for a financial trader. This window receives the data push from the server showing the latest stock prices. When a trader likes the price he may click on the Buy

or Sell button to initiate a trade. The order to trade can be implemented in a separate window and establishing inter-window communications in a loosely coupled manner is really important.

## Applying the Mediator Design Pattern

Three years ago O'Reilly has published another book written by us. That book was titled "Enterprise Development with Flex", and in particular, we've described there how to apply the Mediator design pattern for creating a UI where components can communicate with each other by sending-receiving events from the *mediator* object. The Mediator pattern remains very important in developing UI using any technologies or programming languages, and importance of the HTML5 messaging can't be underestimated.

**Figure B-2** is an illustration from that Enterprise Flex book. The Pricing Panel on the left gets the data feed about the current prices of the IBM stock. When the user clicks on Bid or Ask panel, the Pricing Panel just sends the event with the relevant information like JSON-formatted string containing the stock symbol, price, buy or sell flag, date, etc. In this particular case the window that contained these two panels served as a mediator. In HTML5 realm, we can say that the Pricing Panel invokes `parent.postMessage()` and shoots the message to the mediator (a.k.a. main window).

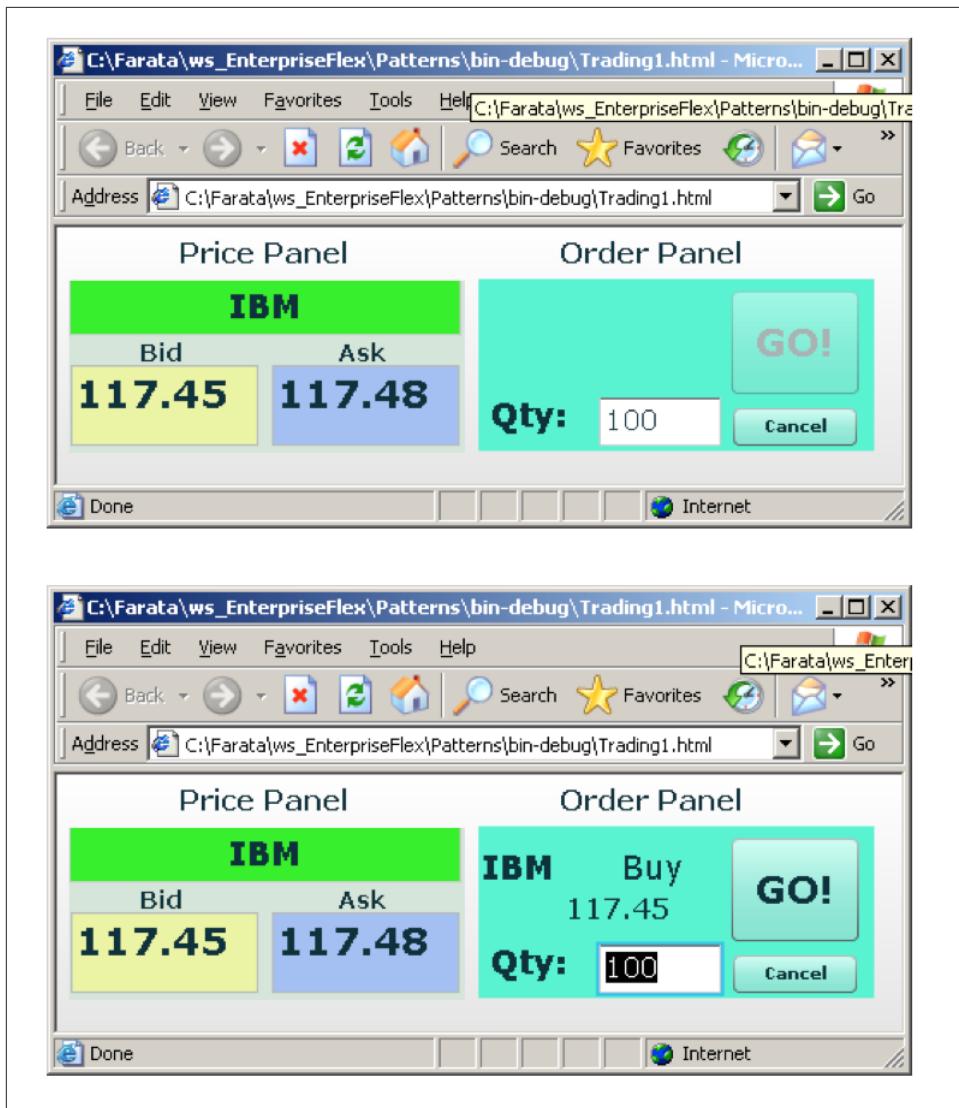


Figure B-2. Before and after the trader clicked on the Price Panel

The Mediator receives the message and re-post it to its another child - the Order Panel that knows how to place orders to purchase stocks. The main takeaway from such design is that the Pricing and Order panels do not know about each other and are communication by sending-receiving messages to/from a mediator. Such a loosely coupled design allows reuse the same code in different applications. For example, the Pricing Panel can be reused in some portal that's used by a company executives in a dashboard showing

prices without the need to place orders. Since the Price Panel has no string attached to Order Panel, it's easy to reuse the existing code in such a dashboard.

You'll see a more advanced example of the inter-component communication techniques that uses *Mediator Design Pattern* in [corresponding section](#) of Chapter 6 of this book.

## HTML5 Forms

While this appendix is about selected HTML APIs, we've should briefly bring your attention to improvements in the HTML5 `<form>` tag too.

It's hard to imagine an enterprise Web application that is not using forms. At the very minimum the Contact Us form has to be there. A login view is yet another example of the HTML form that almost every enterprise application needs. People fill out billing and shipping forms, they answer long questionnaires while purchasing insurance policies online. HTML5 includes some very useful additions that simplify working with forms.

We'll start with the prompts. Showing the hints or prompts right inside the input field will save you some screen space. HTML5 has a special attribute `placeholder`. The text placed in this attribute will be shown inside the field until it gets the focus - then the text disappears. You'll see the use of `placeholder` attribute in Chapter 1 in the logging part of our sample application.

```
<input id="username" name="username" type="text"
       placeholder="username" autofocus>

<input id="password" name="password"
       type="password" placeholder="password"/>
```

Another useful attribute is `autofocus`, which automatically places the focus in the field with this attribute. In the above HTML snippet the focus will be automatically placed in the field `username`.

HTML5 introduces a number of new input types, and many of them have huge impact on the look and feel of the UI on mobile devices. Below are brief explanations.

If the input type is `date`, in mobile devices it'll show native looking date pickers when the focus gets into this field. In desktop computers you'll see a little stepper icon to allow the user select the next or previous month, day, or year without typing. Besides `date` you can also specify such types as `datetime`, `week`, `month`, `time`, `datetime-local`.

If the input type is `email`, the main view of the virtual keyboard on your smartphone will include the `@` key.

If the input type is `url`, the main virtual keyboard will include the buttons `.com`, `..`, and `/`.

The `tel` type will automatically validate telephone numbers for the right format.

The `color` type opens up a color picker control to select the color. After selection, the hexadecimal representation of the color becomes the `value` of this input field.

The input type `range` shows a slider, and you can specify its `min` and `max` values.

The `number` type shows a numeric stepper icon on the right side of the input field.

If the type is `search`, at the very minimum you'll see a little cross on the right of this input field. It allows the user quickly clear the field. On mobile devices, bringing the focus to the search field brings up a virtual keyboard with the Search button. Consider adding the attributes `placeholder` and `autofocus` to the search field.

If the browser doesn't support new input type, it'll render it as a text field.

To validate the input values, use the `required` attribute. It doesn't include any logic, but won't allow submitting the form until the input field marked as `required` has something in it.

The `pattern` attribute allows you to write a regular expression that ensures that the field contains certain symbols or words. For example, adding `pattern="http: .+"` won't consider the input data valid, unless it starts with `http://` followed by one or more characters, one of which has to be period. It's a good idea to include a `pattern` attribute with a regular expression in most of the input fields.



If you're not familiar with regular expressions, watch the presentation [Demistifying Regular Expressions](#) made by Lea Verou at O'Reilly Fluent conference - it's a good primer on this topic.

## Web Workers API

When you start a Web Browser or any other application on your computer or other device, you start *a task* or *a process*. A *thread* is a lighter process within another process. While JavaScript doesn't support multi-threaded mode, HTML5 has a way to run a script as a separate thread in background.

A typical Web application has a UI part (HTML) and a processing part (JavaScript). If a user clicks on a button, which starts a JavaScript function that runs, say for a hundred mili-seconds, there won't be any noticeable delays in user interaction. But if the JavaScript will run a couple of seconds, user experience will suffer. In some cases the Web browser will assume that the script became *unresponsive* and will offer the user to kill it.

Imagine an HTML5 game where a click on the button has to do some major recalculation of coordinates and repainting of multiple images in the browser's window. Ideally, we'd like to parallelize the execution of UI interactions and background JavaScript functions as much as possible, so the user won't notice any delays. Another example is a CPU-intensive spell checker function that finds errors while the user keeps typing. Parsing JSON object is yet another candidate to be done in background. Web workers are also good at polling a server data.

In other words, use Web workers when you want to be able to run multiple parallel *threads of execution* within the same task. On a multi-processor computer parallel threads can run on different CPU's. On a single-processor computer, threads will take turns in getting *slices* of CPU's time. Since switching CPU cycles between threads happens fast, the user won't notice tiny delays in each thread's execution getting a feeling of smooth interaction.

## Creating and Communicating with Workers

HTML5 offers a **solution** for multi-threaded execution of a script with the help of the `Worker` object. To start a separate thread of execution you'll need to create an instance of a `Worker` object passing it the name of the file with the script to run in a separate thread, for example:

```
var mySpellChecker = new Worker("spellChecker.js");
```

The `Worker` thread runs asynchronously and can't directly communicate with the UI components (i.e. DOM elements) of the browser. When the `Worker`'s script finishes execution, it can send back a message using the `postMessage()` method. Accordingly, the script that created the worker thread can listen for the event from the worker and process its responses in the event handler. Such event object will contain the data received from the worker in its property `data`, for example:

```
var mySpellChecker = new Worker("spellChecker.js");
mySpellChecker.onmessage = function(event){

    // processing the worker's response
    document.getElementById('myEditorArea').textContent == event.data;
};
```

You can use an alternative and preferred JavaScript function `addEventListener()` to assign the message handler:

```
var mySpellChecker = new Worker("spellChecker.js");
mySpellChecker.addEventListener("message", function(event){

    // processing the worker's response
    document.getElementById('myEditorArea').textContent == event.data;
});
```

On the other hand, the HTML page can also send any message to the worker forcing it to start performing its duties like start the spell checking process:

```
mySpellChecker.postMessage(wordToCheckSpelling);
```

The argument of `postMessage()` can contain any object, and it's being passed by value, not by reference.

Inside the worker you also need to define an event handler to process the data sent from outside. To continue the previous example the `spellChecker.js` will have inside the code that receives the text to check, performs the spell check, and returns the result back:

```
self.onmessage == function(event){  
  
    // The code that performs spell check goes here  
  
    var resultOfSpellCheck == checkSpelling(event.data);  
  
    // Send the results back to the window that listens  
    // for the messages from this spell checker  
  
    self.postMessage(resultOfSpellCheck);  
};
```

If you want to run a certain code in the background repeatedly, you can create a wrapper function (e.g. `doSpellCheck()`) that internally invokes `postMesage()` and then gives such a wrapper to `setTimeout()` or `setInterval()` to run every second or so:  
`'var timer == setTimeout(doSpellCheck, 1000);'.`

If an error occurs in a worker thread, your Web application will get a notification in a form of an event, and you need to provide a function handler for `onerror`:

```
mySpellChecker.onerror == function(event){  
    // The error handling code goes here  
};
```

## Dedicated and Shared Workers

If a window's script creates a worker thread for its own use, we call it *a dedicated worker*. A window creates an event listener, which gets the messages from the worker. On the other hand, the worker can have a listener too to react to the events received from its creator.

A *shared worker* thread can be used by several scripts as long as they have the same origin. For example, if you want to reuse a spell checker feature in several views of your Web application, you can create a shared worker as follows:

```
var mySpellChecker == new SharedWorker("spellChecker.js");
```

Another use case is funneling all requests from multiple windows to the server through a shared worker. You can also place into a shared worker a number of reusable utility

function that may be needed in several windows - such architecture can reduce or eliminate repeatable code.

One or more scripts can communicate with a shared worker, and it's done slightly different than with the dedicated one. Communication is done through the `port` property and the `start()` method has to be invoked to be able to use `postMessage()` first time:

```
var mySpellChecker == new SharedWorker("spellChecker.js");
mySpellChecker.port.addEventListener("message", function(event){
    document.getElementById('myEditorArea').textContent == event.data;
});
mySpellChecker.port.start()
```

The event handler becomes connected to the `port` property, and now you can post the message to this shared worker using the same `postMessage()` method.

```
mySpellChecker.postMessage(wordToCheckSpelling);
```

Each new script that will connect to the shared worker by attaching an event handler to the `port` results in incrementing the number of active connections that the shared worker maintains. If the script of the shared worker will invoke `port.postMessage("Hello scripts!")`, all listeners that are connected to this port will get it.



If a shared thread is interesting in processing the moments when a new script connects to it, add an event listener to the `connect` event in the code of the shared worker.

If a worker needs to stop communicating with the external world it can call `self.close()`. The external script can kill the worker thread by calling the method `terminate()`, for example:

```
mySpellChecker.terminate();
```



Since the script running inside the `Worker` thread doesn't have access to the browser's UI components, you can't debug such scripts by printing messages onto browser's console with `console.log()`. In [Appendix A](#) we've used Firefox browser for development, but now we'll illustrate how to use Chrome Browser Developer Tools, which includes the [Workers panel](#) that can be used for debugging the code that's launched in worker threads. You'll see multiple examples of using Chrome Developers Tools going forward.

To get a more detailed coverage of Web Workers, read the O'Reilly book by Ido Green titled "Web Workers".



When the user switches to another page in a browser and the current Web page loses focus you may want to stop running some processes that would unnecessary use CPU cycles. To catch this moment use the [Page Visibility API](#).

## WebSockets API

For many years Web applications were associated with HTTP as the main protocol for communication between Web browsers and servers. HTTP is a request-response based protocol that adds hundreds of additional bytes to the application data being sent between browsers and the servers. WebSocket is not a request-response, but a bi-directional full-duplex socket-based protocol, which adds only a couple of bytes (literally) to the application data . WebSockets can become a future replacement for HTTP, but Web applications that require the near-real-time communications (e.g. financial trading applications, online games or auctions) can benefit from this protocol today. Authors of this book believe that WebSocket API is so important, that we dedicated the entire Chapter 8 of this book to this API. In this section we'll just introduce this API very briefly.

This is how the WebSockets workflow goes:

- A Web application tries to establish a socket connection between the client and the server using HTTP only for the initial handshake.
- If the server supports WebSockets, it switches the communication protocol from HTTP to a socket-based protocol.
- From this point on both client and server can send messages in both directions simultaneously (i.e. in full duplex mode).
- This is not a request-response model as both the server and the client can initiate the data transmission which enables the real server-side push.
- Both the server and the client can initiate disconnects too.

This is a very short description of what WebSocket API is about. We encourage you to read Chapter 8 and find the use of this great API in one of your projects.

## Offline Web Applications

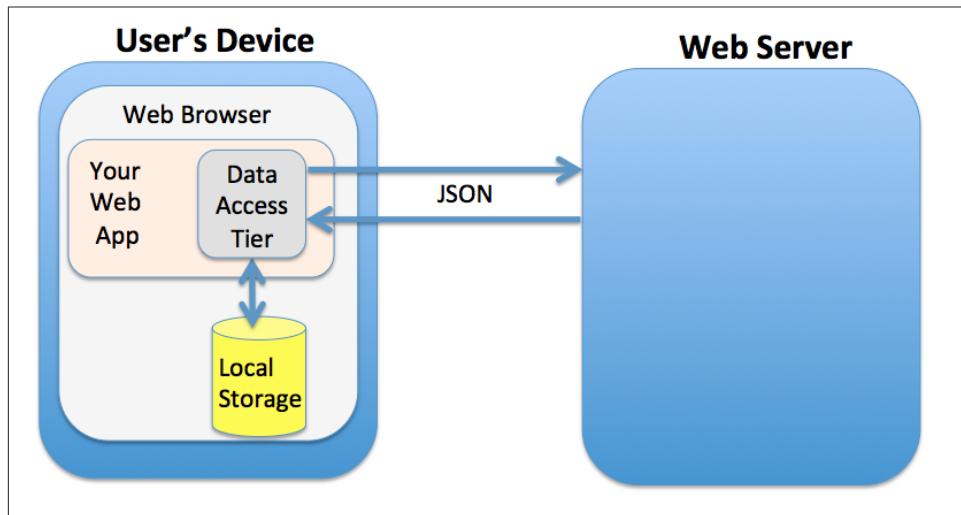
The common misconception about Web applications is that they are useless if there is no connection to the Internet. Everyone knows that native application can be written in a way that they have everything they need installed on your device's data storage - both the application code and the data storage. With HTML5, Web applications can be designed to be functional even when the user's device is disconnected. The offline ver-

sion of a Web application may not offer full functionality, but certain functions can still be available.

## Prerequisites for developing Offline Web Applications

To be useful in a disconnected mode, HTML-based application needs to have access to some local storage on the device, in which case the data entered by the user in the HTML windows can be saved locally with further synchronization with the server when connection becomes available. Think of a salesman of a pharmaceutical visiting medical offices trying to sell new pills. What if connection is not available at certain point? She can still use her tablet demonstrate the marketing materials and more importantly, collect some data about this visit and save them locally. When the Internet connection becomes available again, the Web application should support automatic or manual data synchronization so the information about the salesman activity will be stored in a central database.

There are two main prerequisites for building offline Web applications. You need local storage, and you need to ensure that the server sends only raw data to the client, with no HTML markup (see [Figure B-3](#)). So all these server-side frameworks that prepare the data heavily sprinkled with HTML markup should not be used. For example, the front-end should be developed in HTML/JavaScript/CSS, the back end in your favorite language (Java, .Net, PHP, etc.), and the JSON-formatted data are being sent from the server to the client and back.



*Figure B-3. Design with Offline Use in Mind*

The business logic that supports the client's offline functionality should be developed in JavaScript and run in the Web browser. While most of the business logic of Web applications remains on the server side, the Web client is not as thin as it used to be in legacy HTML-based applications. The client becomes fatter and it can have state.

It's a good idea to create a data layer in your JavaScript code that will be responsible for all data communications. If the Internet connection is available, the data layer will be making requests to the server, otherwise it'll get the data from the local storage.

## Application Cache API

First of all, application's cache is not related to the Web browser's cache. Its main reason for existence is to allow creating applications that can run even if there is no Internet connection available. The user will still go to her browser and enter the URL, but the trick is that the browser will load some previously saved Web pages from the local *application cache*. So even if the user is not online, the application will start anyway.

If your Web application consists of multiple files, you need to specify which ones have to be present on the user's computer in the offline mode. A file called *Cache Manifest* is a plain text file that lists such resources.

Storing some resources in the application cache can be a good idea not only in the disconnected mode, but also to lower the amount of code that has to be downloaded from the server each time the user starts your application. Here's an example of the file mycache.manifest, which includes one CSS file, two JavaScript files, and one image to be stored locally on the user's computer:

```
CACHE MANIFEST
/resources/css/main.css
/js/app.js
/js/customer_form.js
/resources/images/header_image.png
```

The manifest file has to start with the line CACHE MANIFEST and can be optionally divided into sections. The landing page of your Web application has to specify an explicit reference to the location of the manifest. If the above file is located in the document root directory of your application, the main HTML file can refer to the manifest as follows:

```
<!DOCTYPE html>
<html lang="en" manifest="/mycache.manifest">
...
</html>
```

The Web server must serve the manifest file with a MIME type "text/cache-manifest", and you need to refer to the documentation of your Web server to see how to see where to make a configuration change so all files with extension .manifest are served as "text/cache-manifest".

On each subsequent application load the browser makes a request to the server and retrieves the manifest file to see if it has been updated, in which case it reloads all previously cached files. It's a responsibility of Web developers to modify manifest on the server if any of the cacheable resources changed.

## Is Your Application Offline?

Web browsers have a boolean property `window.navigator.onLine`, which should be used to check if there is no connection to the Internet. The HTML5 specification states that "*The `navigator.onLine` attribute must return false if the user agent will not contact the network when the user follows links or when a script requests a remote page (or knows that such an attempt would fail), and must return true otherwise.*" Unfortunately, major Web browsers deal with this property differently so you need to do a thorough testing to see if it works as expected with the browser you care about.

To intercept the changes in the connectivity status, you can also assign event listeners to the `online` and `offline` events, for example:

```
window.addEventListener("offline", function(e) {
    // The code to be used in the offline mode goes here
});

window.addEventListener("online", function(e) {
    // The code to synchronize the data saved in the offline mode
    // (if any) goes here
});
```

You can also add the `onoffline` and `ononline` event handlers to the `<body>` tag of your HTML page or to the `document` object. Again, test the support of these event in your browsers.

What if the browser's support of the `offline/online` events is still not stable? In this you'll have to write your own script that will periodically make an AJAX call (see Chapter 2) trying to connect to a remote server that's always up and running, e.g. [google.com](http://google.com). If this request fails, it's a good indication that your application is disconnected from the Internet.

## Options for Storing Data Locally

In the past, Web browsers could only store their own cache and application's cookies on the user's computer.



Cookies are small files (up to 4Kb) that a Web browser would automatically save locally if the server's `HTTPResponse` would include them. On the next visit of the same URL, the Web browser would send all non-expired cookies back to the browser as a part of `HTTPRequest` object. Cookies were used for arranging HTTP session management and couldn't be considered a solution for setting up a local storage.

HTML5 offers a lot more advanced solutions for storing data locally, namely:

- **Web Storage** which offers Local Storage for long-term storing data and Session Storage for storing a single session data.
- **IndexedDB**: a NoSQL database that stores key-value pairs.



There is another option worth mentioning - **Web SQL Database**. The specification was based on the open source SQLite database. But the work on this specification is stopped and future versions of the browsers may not support it. That's why we are not going to discuss Web SQL Database in this book.



By the end of 2013 local and session storage were supported by all modern Web browsers. Web SQL database is not supported by Firefox and Internet Explorer and most likely will never be. IndexedDB is the Web storage format of the future, but Safari doesn't support it yet, so if your main development platform is iOS, you may need to stick to Web SQL database. Consider using a polyfill for indexedDB using Web SQL API called **IndexedDBShim**.



To get the current status visit [caniuse.com](http://caniuse.com) and search for the API you're interested in.

Although Web browsers send cookies to the Web server, they don't send there the data saved in a local storage. The saved data is used only on the user's device. Also, the data saved in the local storage never expire. A Web application has to programmatically clean up the storage if need be, which will be illustrated below.

# Web Storage Specification APIs

With `window.localStorage` or `window.sessionStorage` (a.k.a. Web Storage) you can store any objects on the local disk as key-value pairs. Both objects implement the Storage interface. The main difference between the two is that the lifespan of the former is longer. If the user reloads the page, the Web browser or restart the computer - the data saved with `window.localStorage` will survive while the data saved via `window.sessionStorage` won't.

Another distinction is that the data from `window.localStorage` is available for any page loaded from the same origin as the page that saved the data. In case of `window.sessionStorage`, the data is available only to the window or a browser's tab that saved it.

## localStorage API

Saving the application state is the main use of the local storage. Coming back to a use case with the pharmaceutical salesman, in the offline mode you can save the name of the person she talked to in a particular medical office and the notes about the conversation that took place, for example:

```
localStorage.setItem('officeID', 123);
localStorage.setItem('contactPerson', 'Mary Lou');
localStorage.setItem('notes', 'Drop the samples of XYZin on 12/15/2013');
```

Accordingly, to retrieve the saved information you'd need to use the method `getItem()`.

```
var officeID == localStorage.getItem('officeID');
var contact == localStorage.getItem('contactPerson');
var notes == localStorage.getItem('notes');
```

This code sample are pretty simple as they store single values. In the real life scenarios we often need to store multiple objects. What if our salesperson has visited several medical offices and needs to save the information about all these visits in the Web Store? For each visit we can create a key-value combination, where a *key* will include the unique id (e.g. office ID), and the *value* will be a JavaScript object (e.g. Visit) turned into a JSON-formatted string (see Chapter 2 for details) using `JSON.stringify()`.

The following code sample illustrates how to store and retrieve the custom `Visit` objects. Each visit to a medical office is represented by on instance of the `Visit` object. To keep the code simple, we've have not included there any HTML components - its JavaScript functions get invoked and print their output on the browser's console.

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>My Today's Visits</title>
</head>
<body>
```

```

<script>

    // Saving in local storage
    var saveVisitInfo == function (officeVisit) {
        var visitStr=JSON.stringify(officeVisit);           // ①
        window.localStorage.setItem("Visit:" + visitNo, visitStr);
        window.localStorage.setItem("Visits:total", ++visitNo);

        console.log("saveVisitInfo: Saved in local storage " + visitStr);
    };

    // Reading from local storage
    var readVisitInfo == function () {

        var totalVisits == window.localStorage.getItem("Visits:total");
        console.log("readVisitInfo: total visits " + totalVisits);

        for (var i == 0; i < totalVisits; i++) {           // ②
            var visit == JSON.parse(window.localStorage.getItem("Visit:" + i));
            console.log("readVisitInfo: Office " + visit.officeId +
                        " Spoke to " + visit.contactPerson + ": " + visit.notes);
        }
    };

    // Removing the visit info from local storage
    var removeAllVisitInfo == function (){                  // ③
        var totalVisits == window.localStorage.getItem("Visits:total");

        for (i == 0; i < totalVisits; i++) {
            window.localStorage.removeItem("Visit:" + i);
        }

        window.localStorage.removeItem("Visits:total");

        console.log("removeVisits: removed all visit info");
    }

    var visitNo == 0;

    // Saving the first visit's info                      // ④
    var visit == {
        officeId: 123,
        contactPerson: "Mary Lou",
        notes: "Drop the samples of XYZin on 12/15/2013"
    };
    saveVisitInfo(visit);

    // Saving the second visit's info                   // ⑤
    visit == {
        officeId: 987,
        contactPerson: "John Smith",
    }

```

```

        notes: "They don't like XYZin - people die from it"
    };
    saveVisitInfo(visit);

    // Retrieving visit info from local storage
    readVisitInfo();                                // ⑥

    // Removing all visit info from local storage
    removeAllVisitInfo();                          // ⑦

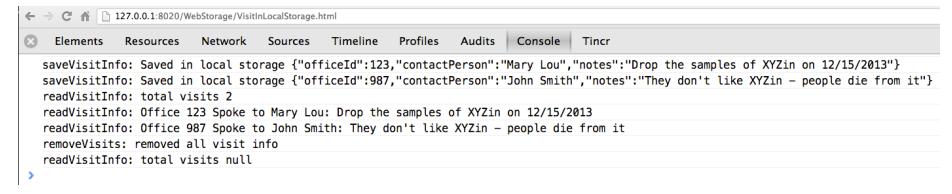
    // Retrieving visit info from local storage - should be no records
    readVisitInfo();                                // ⑧

</script>
</body>
</html>

```

- ➊ The function `saveVisitInfo()` uses JSON object to turn the visit object into a string with `JSON.stringify()`, and then saves this string in the local storage. This function also increments the total number of visits and saves it in the local storage under the key `Visits:total`.
- ➋ The function `readVisitInfo()` gets the total number of visits from the local storage and then reads each visit record recreating the JavaScript object from JSON string using `JSON.parse()`.
- ➌ The function `removeAllVisitInfo()` reads the number of visit records, removes each of them, and then removed the `Visits:total` too.
- ➍ Creating and saving the first visit record
- ➎ Creating and saving the second visit record
- ➏ Reading saved visit info
- ➐ Removing saved visit info. To remove the entire content that was saved for a specific origin call the method `localStorage.clear()`.
- ➑ Re-reading visit info after removal

Figure B-4 shows the output on the console of Chrome Developers Tools. Two visit records were saved in local storage, then they were retrieved and removed from the storage. Finally, the program attempted to read the value of the previously saved `Visits:total`, but it's null now - we've removed from the `localStorage` all the records related to visits.



The screenshot shows the Chrome Developer Tools Console tab. The URL is 127.0.0.1:8020/WebStorage/VisitsInLocalStorage.html. The console output is as follows:

```
Elements Resources Network Sources Timeline Profiles Audits | Console | Tinrcr
saveVisitInfo: Saved in local storage {"officeId":123,"contactPerson":"Mary Lou","notes":"Drop the samples of XYZin on 12/15/2013"}
saveVisitInfo: Saved in local storage {"officeId":987,"contactPerson":"John Smith","notes":"They don't like XYZin - people die from it"}
readVisitInfo: total visits 2
readVisitInfo: Office 123 Spoke to Mary Lou: Drop the samples of XYZin on 12/15/2013
readVisitInfo: Office 987 Spoke to John Smith: They don't like XYZin - people die from it
removeVisits: removed all visit info
readVisitInfo: total visits null
```

Figure B-4. Chrome's console after running the Visits sample



If you are interested in intercepting the moments when the content of local storage gets modified, listen to the DOM storage event, which carries the old and new values and the URL of the page whose data is being changed.



Another good example of a use case when localStorage becomes handy is when the user is booking air tickets using more than one browser's tab.

## sessionStorage API

The sessionStorage life is short - it's only available for a Web page while the browser stays open. If the user decides to refresh the page, the sessionStorage will survive, but opening a page in a new browser's tab or window will create a new sessionStorage object. Working with the session storage is pretty straightforward, for example

```
sessionStorage.setItem("userID", "jsmith");

var userID == sessionStorage.getItem("userID");
```

Chrome Developer Tools include the tab Resources that allows browsing the local or session storage if a Web page uses it. For example, Figure B-5 shows the storage used by cnn.com.

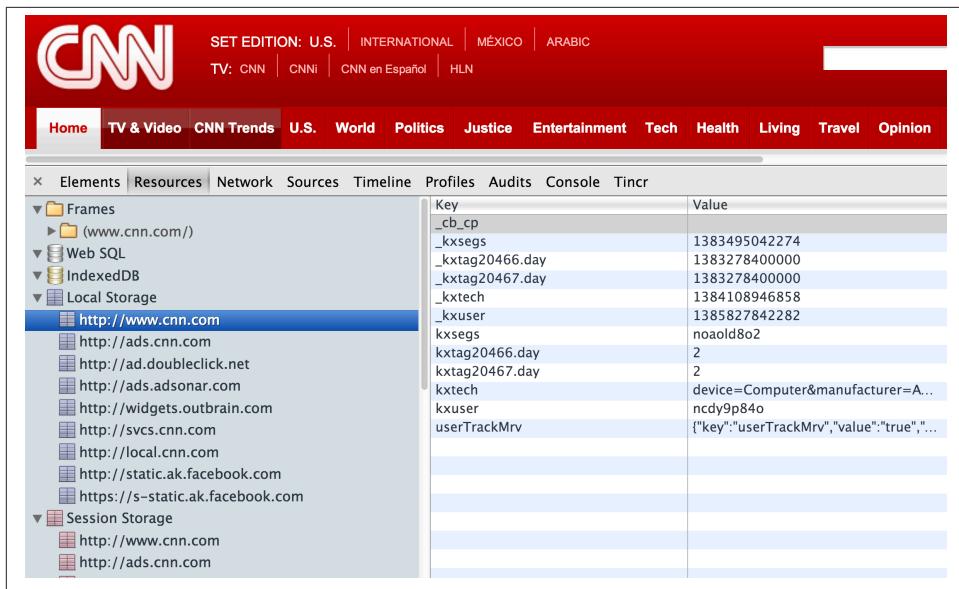


Figure B-5. Browsing local storage in Chrome Developer Tools

### localStorage and sessionStorage Commonalities

Both `localStorage` and `sessionStorage` are subject to the same-origin policy, meaning that saved data will be available only for the Web pages that came from the same host, port and via the same protocol.

Both `localStorage` and `sessionStorage` are browser-specific. For example, if the Web application stored the data from Firefox, the data won't be available if the user opens the same application from Safari.

The APIs from Web Storage specification are pretty simple to use, but their major drawbacks are that they don't give you a way to put any structure to the stored data, you always have to store strings, and the API is synchronous, which may cause delays in the user interaction when your application is accessing the disk.

There is no actual limits for the size of local storage, but the browsers usually default this size to 5Mb. If the application tries to store more data than the browser permits, the `QUOTA\_EXCEEDED\_ERR` exception will be thrown - always use the try-catch blocks when saving data.

Even if the user's browser allows increasing this setting (e.g. via `about:config` URL in Firefox), access to such data may be slow. Consider using [File API](#) or `IndexedDB` that will be introduced in the next section.

## Introduction to IndexedDB

Indexed Database API (a.k.a. IndexedDB) is a solution based on the NoSQL database. Like with the Storage interface, IndexedDB stores data as key-value pairs, but it also offers transactional handling of objects. IndexedDB creates indexes of the stored objects for fast retrieval. With Web Storage you can only store strings, and we had to do these tricks with JSON `stringify()` and `parse()` to give some structure to these strings. With IndexedDB you can directly store and index regular JavaScript objects.

IndexedDB allows you to access data asynchronously, so there won't be UI freezes while accessing large objects on disk. You make a request to the database and define the event handlers that should process errors or result when ready. IndexedDB uses DOM events for all notifications. Success events don't bubble, while error events do.

The users will have a feeling that the application is pretty responsive, which wouldn't be the case if you'll be saving several megabytes of data with Web Storage API. Similarly to Web Storage, access to the IndexedDB databases is regulated by the same origin policy.



In the future, Web browsers may implement [synchronous IndexedDB API](#) to be used inside Web workers.

Since not every browser supports IndexedDB yet, you can use Modernizr (see Chapter 1) to detect if your browser supports it. If it does, you still may need to account for the fact that browser vendors name the IndexedDB related object differently. Hence to be on the safe side, at the top of your script include the statements to account for the prefixed vendor-specific implementations of `indexedDB` and related objects:

```
var medicalDB == {}; // just an object to store references

medicalDB.indexedDB == window.indexedDB || window.mozIndexedDB
    || window.msIndexedDB || window.webkitIndexedDB ;
if (!window.indexedDB){
    // this browser doesn't support IndexedDB
} else {
    medicalDB.IDBTransaction == window.IDBTransaction || window.webkitIDBTransaction;
    medicalDB.IDBCursor == window.IDBCursor || window.webkitIDBCursor;
    medicalDB.IDBKeyRange == window.IDBKeyRange || window.webkitIDBKeyRange;
}
```

In the above code snippet the `IDBKeyRange` is an object that allows to restrict the range for the continuous keys while iterating through the objects. `IDBTransaction` is an implementation of transaction support. The `IDBCursor` is an object that represents a cursor for traversing over multiple objects in the database.

IndexedDB doesn't require you to define a formal structure of your stored objects - any JavaScript object can be stored there. Not having a formal definition of a database scheme is an advantage comparing to the relational databases where you can't store the data until the structure of the tables is defined.

Your Web application can have one or more databases, and each of them can contain one or more *object stores*. Each of the object stores will contain similar objects, e.g. one store is for salesman's visits, while another stores upcoming promotions.

Every object that you are planning to store in the database has to have one property that plays a role similar to a primary key in a relational database. You have to decide if you want to maintain the value in this property manually, or use the `autoIncrement` option where the values to this property will be assigned automatically. Coming back to our Visits example, you can either maintain the unique values of the `officeId` on your own or create a surrogate key that will be assigned by IndexedDB. The current generated number to be used as surrogate keys never decreases, and starts with the value of 1 in each object store.

Similarly to relational databases you create indexes based on the searches that you run often. For example, if you need to search on the contact name in the Visits store, create an index on the property `contactPerson` of the `Visit` objects. But if in relational databases creation of indexes is done for performance reasons, with IndexedDB you can't run a query unless the index on the relevant property exists. The following code sample shows how to connect to the existing or create a new object store `Visits` in a database called `Medical_DB`.

```
var request == medicalDB.indexedDB.open('Medical_DB');           // ①

request.onsuccess == function(event) {                         // ②
    var myDB == request.result;

};

request.onerror == function (event) {                          // ③
    console.log("Can't access Medical_DB: " + event.target.errorCode);
};

request.onupgradeneeded == function(event){ // ④
    event.currentTarget.result.createObjectStore ("Visits",
        {keypath: 'id', autoIncrement: true});
};
```

- ① The browser invokes the method `open()` asynchronously requesting to establish the connection with the database. It doesn't wait for the completion of this request, and the user can continue working with the Web page without any delays or interruptions. The method `open()` returns an instance of the `IDBRequest` object.

- ② When the connection is successfully obtained, the `onsuccess` function handler will be invoked. The result is available through the `IDBRequest.result` property.
- ③ Error handling is done here. The event object given to the `onerror` handler will contain the information about the error.
- ④ The `onupgradeneeded` handler is the place to create or upgrade the storage to a new version. This is explained next.



There are several scenarios to consider while deciding if you need to use the `autoIncrement` property with the store key or not. Kristof Degrave described the article <http://www.kristofdegrave.be/2012/02/indexed-db-to-provide-key-or-not-to.html>[Indexed DB: To provide a key or not to provide a key.]

## Object Stores and Versioning

In the world of traditional DBMS servers, when the database structure has to be modified, the DBA will do this upgrade, the server will be restarted, and the users will work with the *new version* of the database. With IndexedDB it works differently. Each database has a version, and when the new version of the database (e.g. `Medical_DB`) is created, the `onupgradeneeded` is dispatched, which is where object store(s) are created. But if you already had object stores in the older version of the database, and they don't need to be changed - there is no need to re-create them.

After successful connection to the database, the version number is available in `IDBRequest.result.version` property. The starting version of any database is 1.

The method `open()` takes a second parameter - the database version to be used. If you don't specify the version - the latest one will be used. The following line shows how the application's code can request connect to the version 3 of the database `Medical_DB`:

```
var request = indexedDB.open('Medical_DB', 3);
```

If the user's computer has already the `Medical_DB` database of one of the earlier versions (1 or 2), the `onupgradeneeded` handler will be invoked. The initial creation of the database is triggered the same way - the absence of the database also falls under the "upgrade is needed" case, and the `onupgradeneeded` handler has to invoke the `createObjectStore()` method. If upgrade is needed, the `onupgradeneeded` will be invoked before the `onsuccess` event.

The following code snippet creates a new or initial version of the object store `Visits`, requesting auto-generation of the surrogate keys named `id`. It also creates indexes to allow search by office ID, contact name and notes. Indexes are updated automatically as soon as the Web application makes any changes to the stored data. If you wouldn't create indexes, you'd be able to look up objects only by the value of key.

```

request.onupgradeneeded == function(event){ // ①
  var visitsStore ==
    event.currentTarget.result.createObjectStore ("Visits",
      {keypath='id',
       autoIncrement: true
     });

  visitsStore.createIndex("officeIDindex", "officeID",
    {unique: true});
  visitsStore.createIndex("contactsIndex", "contactPerson",
    {unique: false});
  visitsStore.createIndex("notesIndex", "notes",
    {unique: false});
};

}

```

Note that while creating the object store for visits, we could have used a unique property `officeID` as a keypath value by using the following syntax:

```

var visitsStore ==
  event.currentTarget.result.createObjectStore ("Visits",
    {keypath='officeID'});

```

The `event.currentTarget.result` (as well as `IDBRequest.result`) points at the instance of the `IDBDatabase` object, which has a number of useful properties such as `name`, that contains the name of the current database and the array `objectStoreNames`, which has the names of all object stores that exist in this database. Its property `version` has the database version number. If you'd like to create a new database, just call the method `open()` specifying the version number that's higher than the current one.

To remove the existing database, call the method `indexedDB.deleteDatabase()`. To delete the existing object store invoke `indexedDB.deleteObjectStore()`.



IndexedDB doesn't offer a secure way of storing data. Anyone who has access to the user's computer can get a hold of the data stored in IndexedDB. Do not store any sensitive data locally. Always use secure "https" protocol with your Web application.

## Transactions

Transaction is a logical unit of work. Executing several database operation in one transaction guarantees that the changes will be committed to the database only if all operations finished successfully. If at least one of the operations fails, the entire transaction will be rolled back (undone). IndexDB supports three transaction modes: `readonly`, `readwrite`, and `versionchange`.

To start any manipulations with the database you have to open a transaction in one of these modes. The `readonly` transaction (the default one) allows multiple scripts to read from the database concurrently. This statement may raise a question - why would the

user need a concurrent access to his local database is he's the only user of the application on his device? The reason being that the same application can be opened in more than one tab, or spawning more than one worker thread that need to access the local database. The `readonly` is the least restrictive mode and more than one script can open a `readonly` transaction.

If the application needs to modify or add objects to the database, open transaction in the `readwrite` mode - only one script can have it open on any particular object store. But you can have more than one `readwrite` transactions open at the same time on different stores. And if the database/store/index creation or upgrade has to be done, use the `versionchange` mode.

When a transaction is created, you should assign listeners to its `complete`, `error`, and `abort` events. If the `complete` event is fired, transaction is automatically committed - manual commits are not supported. If the `error` event is dispatched, the entire transaction is rolled back. Calling the method `abort()` will fire the `abort` event and will roll back transaction too.

Typically, you should open the database and in the `onsuccess` handler create a transaction. Then open a transaction by calling the method `objectStore()` and perform data manipulations. In the next section you'll see how to add objects to an object store using transactions.

## Modifying the Object Store Data

The following code snippet creates the transaction that allows updates of the store `Visits` (you could create a transaction for more than one store) and add two visit object by invoking the method `add()`:

```
request.onsuccess == function(event) {          // ①
    var myDB == request.result;

    var visitsData == [{                           // ②
        officeId: 123,
        contactPerson: "Mary Lou",
        notes: "Drop the samples of XYZin on 12/15/2013"
    },
    {
        officeId: 987,
        contactPerson: "John Smith",
        notes: "They don't like XYZin - people die from it"
    }];
}

var transaction == myDB.transaction(["Visits"],
                                    "readwrite"); // ③
transaction.oncomplete == function(event){
    console.log("All visit data have been added");
}
```

```

transaction.onerror == function(event){
    // transaction rolls back here
    console.log("Error while adding visits");
}

var visitsStore == transaction.objectStore("Visits"); // ④

for (var i in visitsData) {
    visitsStore.add(visitsData[i]); // ⑤
}

```

- ➊ The database opened successfully.
- ➋ Create a sample array of `visitsData` to illustrate adding more than one object to an object store.
- ➌ Open a transaction for updates and assign listeners for success and failure. The first argument is an array of object stores that the transaction will span (only `Visits` in this case). When all visits are added the `complete` event is fired and transaction commits. If adding of any visit failed, the `error` event is dispatched and transaction rolls back.
- ➍ Get a reference to the object store `visits`.
- ➎ In a loop, add the data from the array `visitsData` to the object store `Visits`.



In the above code sample, each object that represents a visit has a property `notes`, which is a string. If later on you'll decide to allow storing more than one note per visit, just turn the property `notes` into an array in your JavaScript - no changes in the object stores is required.

The method `put()` allows you to update an existing object in a record store. It takes two parameters: the new object and the key of the existing object to be replaced, for example:

```
var putRequest == visitsStore.put({officeID: 123, contactName: "Mary Lee"}, 1);
```

To remove all objects from the store use the method `clear()`. To delete an object specify its id:

```
var deleteRequest == visitsStore.delete(1);
```



You can browse the data from your IndexedDB database in Chrome Developer Tools under the tab Resources (see [Figure B-5](#)).

## Retrieving the Data

IndexedDB doesn't support SQL. You'll be using cursors to iterate through the object store. First, you open the transaction. Then you open invoke `openCursor()` on the object store. While opening the cursor you can specify optional parameters like the range of object keys you'd like to iterate and the direction of the cursor movement: `IDBCursor.PREV` or `IDBCursor.NEXT`. If none of the parameters is specified, the cursor will iterate all objects in the store in the ascending order. The following code snippet iterates through all Visit objects printing just contact names.

```
var transaction = myDB.transaction(["visits"], "readonly");
var visitsStore = transaction.objectStore("Visits");

visitsStore.openCursor().onsuccess = function(event){
    var visitCursor = event.target.result;
    if (visitCursor){
        console.log("Contact name: " + visitCursor.value.contactPerson);
        visitCursor.continue();
    }
}
```

If you want to iterate through a limited key range of objects you can specify the from-to values. The next line creates a cursor for iterating the first five objects from the store:

```
var visitsCursor = visitsStore.openCursor(IDBKeyRange.bound(1, 5));
```

You can also create a cursor on indexes - this allows working with sorted sets of objects. In one of the earlier examples we've created an index on `officeID`. Now we can get a reference to this index and create a cursor on the specified range of sorted office IDs as in the following code snippet:

```
var visitsStore = transaction.objectStore("visits");
var officeIdIndex = visitsStore.index("officeID");

officeIdIndex.openCursor().onsuccess = function(event){
    var officeCursor = event.target.result;
    // iterate through objects here
}
```

To limit the range of offices to iterate through, you could open the cursor on the `officeIdIndex` differently. Say you need to create a filter to iterate the offices with the numbers between 123 and 250. This is how you can open such a cursor:

```
officeIdIndex.openCursor(IDBKeyRange.bound(123, 250, false, true));
```

The `false` in the third argument of `bound()` means that 123 should be included in the range, and the `true` in the fourth parameter excludes the object with `officeID=250` from the range. The methods `lowerbound()` and `upperbound()` are other variations of the method `bound()` - consult the [online documentation](#) for details.

If you need to fetch just one specific record, restrict the selected range to only one value using the method `only()`:

```
contactNameIndex.openCursor(IDBKeyRange.only("Mary Lou"));
```

## Runninng the Sample code

Let's bring together all of the above code snippets into one runnable HTML file. While doing this, we'll be watching the script execution in Chrome Developer Tools panel. We'll do it in two steps. The first version of this file will create a database of a newer version than the one that currently exists on the user's device. Here's the code that creates the database `Medical_DB` with an empty object store `Visits`:

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>My Today's Visits With InsexedDB</title>
</head>
<body>
    <script>
        var medicalDB == {};
        // just an object to store references
        var myDB;

        medicalDB.indexedDB == window.indexedDB || window.mozIndexedDB
            || window.msIndexedDB || window.webkitIndexedDB ;
        if (!window.indexedDB){
            // this browser doesn't support IndexedDB
        } else {
            medicalDB.IDBTransaction == window.IDBTransaction || window.webkitIDBTransaction;
            medicalDB.IDBCursor == window.IDBCursor || window.webkitIDBCursor;
            medicalDB.IDBKeyRange == window.IDBKeyRange || window.webkitIDBKeyRange;
        }

        var request == medicalDB.indexedDB.open('Medical_DB', 2); // ①

        request.onsuccess == function(event) {
            myDB == request.result;
        };

        request.onerror == function (event) {
            console.log("Can't access Medical_DB: " + event.target.errorCode);
        };

        request.onupgradeneeded == function(event){
            event.currentTarget.result.createObjectStore ("Visits",
                {keypath:'id', autoIncrement: true}); // ②
        };

    </script>
</body>
</html>
```

- This version of the code is run when the user's computer already had a database Medical\_DB: initially we've invoked `open()` without the second argument. Running the code specifying 2 as the version caused invocation of the callback `onupgradeneeded` even before the `onsuccess` was called.
- Create an empty object store Visits

**Figure B-6** shows the screen shot from the Chrome Developer Tools at the end of processing the `success` event. Note the Watch Expression section on the right. The name of the database is Medical\_DB, its version number is 2, and the `IDBDatabase` property `objectStoreNames` shows that there is one object store named Visits.

```

Elements Resources Network Sources Timeline Profiles Audits Console Tincr
VisitsIndexedDB.html <| ▾ Watch Expressions
1 <!doctype html>
2 <html>
3 <head>
4   <meta charset="utf-8" />
5   <title>My Today's Visits With InsexedDB</title>
6 </head>
7 <body>
8   <script>
9     var medicalDB = {}; // just an object to store references
10
11   medicalDB.indexedDB = window.indexedDB || window.mozIndexedDB
12   || window.msIndexedDB || window.webkitIndexedDB ;
13   if (!window.indexedDB){ // this browser doesn't support IndexedDB
14     }
15   else {
16     medicalDB.IDBTransaction = window.IDBTransaction || window.webkitIDBTransaction
17     medicalDB.IDBCursor = window.IDBCursor || window.webkitIDBCursor;
18     medicalDB.IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange
19   }
20
21   // Open the database
22   var request = medicalDB.indexedDB.open('Medical_DB', 2);
23
24   request.onsuccess = function(event) {
25     var myDB = request.result;
26   };
27
28   request.onerror = function (event) {
29     console.log("Can't access Medical_DB: " + event.target.errorCode);
30   };
31
32   request.onupgradeneeded = function(event){ // <4>
33     event.currentTarget.result.createObjectStore ('Visits',
34       {keypath:'id', autoIncrement: true});
35   };
36
37
38   </script>
39 </body>
40 </html>

```

Figure B-6. Chrome's console after running the Visits sample

The next version of our sample HTML file will populate the object store Visits with some data, and then will iterate through all the Visit objects and display the values of their properties on the console.

```

<!doctype html>
<html>
<head>
<meta charset="utf-8" />

```

```

<title>My Today's Visits With InsexedDB</title>
</head>
<body>
<script>
    var medicalDB == {} ; // just an object to store references
    var myDB;

medicalDB.indexedDB == window.indexedDB || window.mozIndexedDB
    || window.msIndexedDB || window.webkitIndexedDB ;
if (!window.indexedDB){
    // this browser doesn't support IndexedDB
} else {
    medicalDB.IDBTransaction == window.IDBTransaction || window.webkitIDBTransaction;
    medicalDB.IDBCursor == window.IDBCursor || window.webkitIDBCursor;
    medicalDB.IDBKeyRange == window.IDBKeyRange || window.webkitIDBKeyRange;
}

var request == medicalDB.indexedDB.open('Medical_DB', 2);

request.onsuccess == function(event) {
    myDB == request.result;

var visitsData == [
    {
        officeId: 123,
        contactPerson: "Mary Lou",
        notes: "Drop the samples of XYZin on 12/15/2013"
    },
    {
        officeId: 987,
        contactPerson: "John Smith",
        notes: "They don't like XYZin - people die from it"
    }];
}

var transaction == myDB.transaction(["Visits"],
    "readwrite");
transaction.oncomplete == function(event){
    console.log("All visit data have been added.");

    readAllVisitsData(); // ①
}

transaction.onerror == function(event){
    // transaction rolls back here
    console.log("Error while adding visits");
}

var visitsStore == transaction.objectStore("Visits");

visitsStore.clear(); // ②

for (var i in visitsData) {
    visitsStore.add(visitsData[i]);
}

```

```

        }
    };

    request.onerror == function (event) {
        console.log("Can't access Medical_DB: " + event.target.errorCode);
    };

    request.onupgradeneeded == function(event){
        event.currentTarget.result.createObjectStore ("Visits",
            {keypath:'id', autoIncrement: true});
    };
}

function readAllVisitsData(){
    var readTransaction == myDB.transaction(["Visits"], "readonly");

    readTransaction.onerror == function(event){
        console.log("Error while reading visits");
    }

    var visitsStore == readTransaction.objectStore("Visits");

    visitsStore.openCursor().onsuccess == function(event){ // ③
        var visitsCursor == event.target.result;

        if (visitsCursor){
            console.log("Contact name: " +
                visitsCursor.value.contactPerson +
                ", notes: " +
                visitsCursor.value.notes);
            visitsCursor.continue(); // ④
        }
    }
}
</script>
</body>
</html>

```

- ① After the data store is populated and transaction is committed, invoke the method to read all the objects from the Visits store.
- ② Remove all the objects from the store Visits before populating it with the data from the array VisitsData.
- ③ Open the cursor to iterate through all visits
- ④ Move the cursor's pointer to the next object after printing the contact name and notes in the console.

Figure B-7 shows the screenshot from Chrome Developer Tools when the debugger stopped in `readAllVisitsData()` right after reading both objects from the Visits store.

The console output is shown at the bottom. Note the content of the visitsCursor on the right. The cursor is moving forward (the next direction), and the value property points at the object at cursor. The key value of the object is 30. It's auto-generated, and on each run of this program you'll see a new value since we clean the store and re-insert the objects, which generates the new keys.

The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The code being run is:

```

70 event.currentTarget.result.createObjectStore("Visits",
71   {keypath: 'id', autoIncrement: true});
72 ;
73
74
75 function readAllVisitsData(){
76   var readTransaction = myDB.transaction(["Visits"], "readonly");
77
78   readTransaction.onerror = function(event){
79     console.log("Error while reading visits");
80   }
81
82   var visitsStore = readTransaction.objectStore("Visits");
83
84   visitsStore.openCursor().onsuccess = function(event){
85     var visitsCursor = event.target.result;
86
87     if (visitsCursor){
88       console.log("Contact name: " + visitsCursor.value.contactPerson +
89         ", notes: " + visitsCursor.value.notes);
90       visitsCursor.continue();
91     }
92   }
93 }
94 </script>

```

The console output at the bottom shows:

- All visit data have been added.
- Contact name: Mary Lou, notes: Drop the samples of XYZin on 12/15/2013
- Contact name: John Smith, notes: They don't like XYZin - people die from it

The right panel displays the state of variables in the current scope. The visitsCursor variable is expanded to show its properties:

- direction: "next"
- key: 30
- primaryKey: 30
- source: IDBObjectStore
- value: Object
  - contactPerson: "John Smith"
  - notes: "They don't like XYZin - people die from it"
  - objectId: 987

Figure B-7. Chrome's console after reading the first Visit object

This concludes our brief introduction to IndexedDB. Those of you who have experience in working with relational databases may find the querying capabilities of IndexedDB rather limited comparing to powerful relational databases like Oracle or MySQL. On the other hand, IndexedDB is pretty flexible - it allows you to store and look up any JavaScript objects without worrying about creating a database schema first. At the time of this writing there are no books dedicated to IndexedDB. For up to date information refer to [IndexedDB online documentation](#) at Mozilla Developer Network.

## History API

To put is simple, **History API** is about ensuring that the Back/Forward buttons on the browser toolbar can be controlled programmatically. Each Web browser has the `window.history` object. The History API is not new to HTML5. The `history` object has been around for many years, with methods like `back()`, `forward()`, and `go()`. But HTML5 adds new methods `pushState()` and `replaceState()`, which allow to modify the browser's address bar without reloading the Web page.

Imagine a Single Page Application (SPA) that has a navigational menu to open various views as based on the user's interaction. Since these views represents some URLs loaded by making AJAX calls from your code, the Web browser still shows the original URL of the home page of your Web application.

The perfect user would always navigate your application using the menus and controls you provided, but what if she clicks on the Back button of the Web browser? If the navigation controls were not changing the URL in the browser's address bar, the browser obediently will show the Web page that the user has visited before even launching your application, which is most likely not what she intended to do. History API allows to create more fine grained bookmarks that define specific state within the Web page.



Not writing any code that would process clicks on the Back and Forward buttons is the easiest way to frustrate your users.

## Modifying the Browser's History with pushState()

Imagine you have a customer-management application that has a URL <http://myapp.com>. The user clicked on the menu item Get Customers, which made an AJAX call loading the customers. You can programmatically change the URL on the browser's address line to be <http://myapp.com/customers> without asking the Web browser to make a request to this URL. You do this by invoking the `pushState()` method.

The browser will just remember that the current URL is <http://myapp.com/customers>, while the previous was <http://myapp.com>. So pressing the Back button would change the address back to <http://myapp.com>, and not some unrelated Web application. The Forward button will also behave properly as per the history chain set by your application.

The `pushState()` takes three arguments (the values from the first two may be ignored by some Web browsers):

- The application specific state to be associated with the current view of the Web page
- The title of the current view of the Web page.
- The suffix to be associated with the current view of the page. It'll be added to the address bar of the browser.

```
<head>
  <meta charset="utf-8">
  <title>History API</title>
</head>
<body>
  <div id="main-container">
```

```

<h1>Click on Link and watch the address bar...</h1>

<button type="button" onclick="whoWeAre()">Who we are</button>
①

<button type="button" onclick="whatWeDo()">What we do</button>

</div>

<script>

function whoWeAre(){
    var locationID== {locID: 123,                                // ②
                      uri: '/whowear'};
    history.pushState(locationID,'', 'who_we_are' );           // ③
}

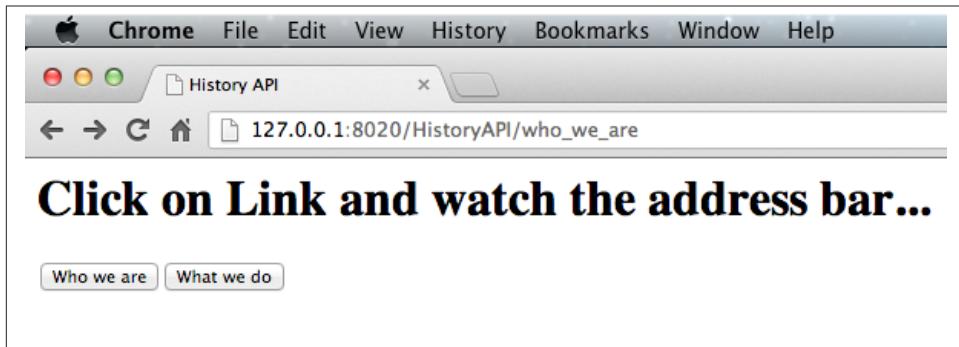
function whatWeDo(){
    var actionID== {actID: 123,                                // ④
                   uri: '/whatwedo'};
    history.pushState(actionID,'', 'what_we_do' );            // ⑤
}
</script>
</body>
</html>

```

- ① On a click of the button call the event handler function. Call the `pushState()` to modify the browser's history. Some other processing like making an AJAX request to the server can be done in `whoWeAre()` too.
- ② Prepare the custom state object to be used in server side requests. The information about *who we are* depends on location id.
- ③ Calling `pushState()` to remember the customer id, the page title is empty (not supported yet), and adding the suffix `/who_we_are` will serve as a path to the server-side REST request.
- ④ Prepare the custom state object to be used in server side requests. The information about *what we do* depends on customer id.
- ⑤ Calling `pushState()` to remember the customer id, the page title is empty (not supported yet), and adding the suffix `/what_we_do` will serve as a path to the server-side REST request.

This above sample is a simplified example and would require more code to properly form the server request, but our goal here is just to clearly illustrate the use of History API.

**Figure B-8** depicts the view after the user clicked on the button Who We Are. The URL now looks as [http://127.0.0.1:8020/HistoryAPI/who\\_we\\_are](http://127.0.0.1:8020/HistoryAPI/who_we_are), but keep in mind that if you try to reload the page while this URL is shown, the browser will give you a Not Found error and rightly so. There is no resource that represents the URL that ends with *who\_we\_are* - it's just the name of the view in the browser's history.



*Figure B-8. Testing pushState()*

Using the `replaceState()` you can technically “change the history”. We are talking about the browser’s history, of course.

## Processing the popstate Event

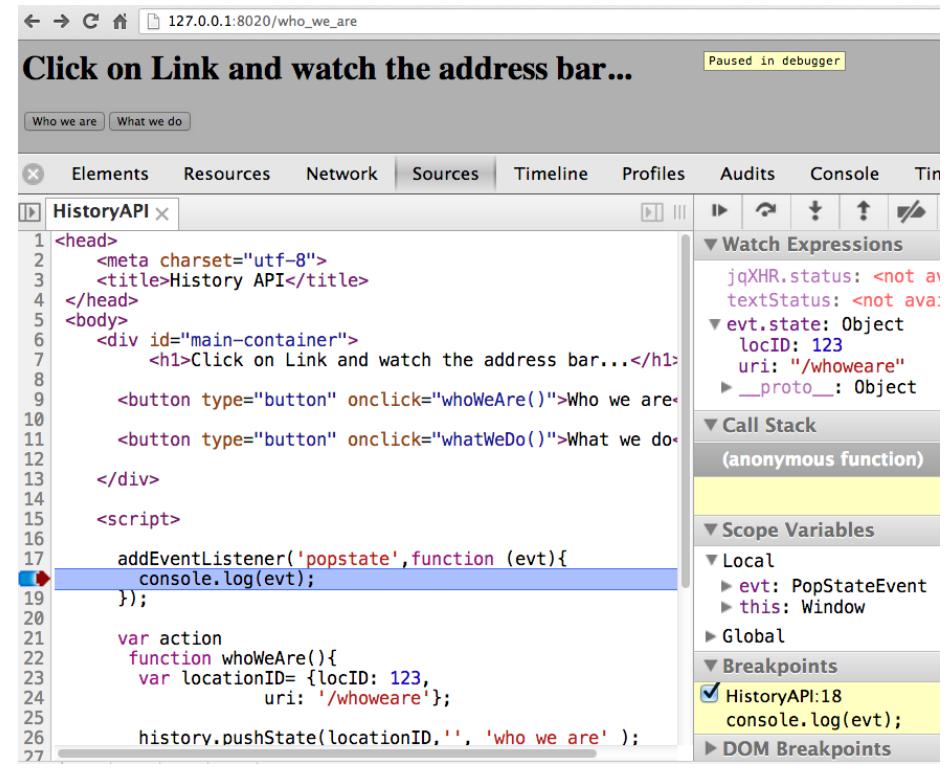
But changing the URL when the user clicks on the Back or Forward button is just the half of the job to be done. The content of the page has to be refreshed accordingly. The browser dispatches the event `window.popstate` whenever the browser’s navigation history changes either on initial page load, as a result of clicking on the Back/Forward buttons, or by invoking `history.back()` or `history.forward()`.

Your code has to include an event handler function that will perform the actions that must be done whenever the application gets into the state represented by the current suffix, e.g. make a server request to retrieve the data associated with the state *who\_we\_are*. The `popstate` event will contain a copy of the history’s entry state object. Let’s add the following event listener to the `<script>` part of the code sample from previous section:

```
addEventListener('popstate',function (evt){  
    console.log(evt);  
});
```

**Figure B-9** depicts the view of the Chrome Developers Tool when the debugger stopped in the listener of the `popstate` event after the user clicked on the buttons Who We Are, then What We Do, and then the browser’s button Back. On the right hand side you can

see that the event object contains the `evt.state` object with the right values of `locID` and `uri`. In the real world scenario these values could have been used in, say AJAX call to the server to recreate the view for the location ID 123.



The screenshot shows the Chrome Developers Tool's Elements tab with the HistoryAPI code. A red arrow points to the line `addEventListener('popstate',function (evt){`. The Watch Expressions panel on the right shows the `evt.state` object expanded, revealing `locID: 123` and `uri: '/whowear'`. The Call Stack and Scope Variables panels are also visible.

```
1 <head>
2   <meta charset="utf-8">
3   <title>History API</title>
4 </head>
5 <body>
6   <div id="main-container">
7     <h1>Click on Link and watch the address bar...</h1>
8
9     <button type="button" onclick="whoWeAre()">Who we are</button>
10    <button type="button" onclick="whatWeDo()">What we do</button>
11  </div>
12
13  <script>
14
15    addEventListener('popstate',function (evt){
16      console.log(evt);
17    });
18
19    var action
20    function whoWeAre(){
21      var locationID= {locID: 123,
22                      uri: '/whowear'};
23
24      history.pushState(locationID,'', 'who we are' );
25
26    }
27
```

Figure B-9. Monitoring `popState` with Chrome Developers Tool



If you'll run into a browser that doesn't support HTML5 History API, consider using the [History.js](#) library.

## Custom Data Attributes

We've included this sidebar in this appendix, even though it's not API. But we're talking about HTML here and don't want to miss this important feature of the HTML5 specification - you can add to any HTML tag any number of **custom non-visible attributes**

as long as they start with `data-` and have at least one character after the hyphen. For example, this is absolutely legal in HTML5:

```
<ol>
  <li data-phone="212-324-6656">Mary</li>
  <li data-phone="732-303-1234">Anna</li>
  ...
</ol>
```

Behind the scenes, a custom framework can find all elements that have the `data-phone` attribute and generate some additional code for processing of the provided phone number. If this example doesn't impress you, wait till Chapter 10, where you'll learn how to use jQuery Mobile. The creators of this library use these `data-` attributes in a very smart way.

## Summary

In this appendix you've got introduced to a number of useful HTML5 APIs. You know how to check if a particular API is supported by your Web browser. But what if you are one of many enterprise developers that must use Internet Explorer of the versions earlier than 10.0? Google used to offer a nice solution: [Google Chrome Frame](#), which was a plugin for Internet Explorer.

The users had to install Chrome Frame on their machines, and Web developers just needed to add the following line to their Web pages:

```
<meta http-equiv="X-UA-Compatible" content="chrome=1" />
```

After that the Web page rendering would be done by Chrome Frame while your Web application would run in Internet Explorer. Unfortunately, Google decided not to support the Chrome project as of January 2014. They are recommending to prompt the user of your application to upgrade the Web browser, which may not be something that the users will be willing to do. But let's hope for the best.



# Appendix C. Running Code Samples and IDE

The code samples used in this book are available on [Github](#) - they are grouped by chapters. If a chapter has code samples, look for the directory with the respective name.

Technically, you don't have to use any Integrated Development Environment (IDE) to run code examples (except the CDB example from Chapter 5) - just open the main file in a Web browser and off you go. But using an IDE will make you more productive.

## Which IDE to Use

Selecting an IDE that supports JavaScript is a matter of your personal preference. Since there is no compilation stage and most of your debugging will be done using the Web browser tools, picking a text editor that supports syntax highlighting is all that most developers need. For example, there is an excellent text editor [Sublime Text 2](#). Among many programming languages this editor understands the keywords of HTML, CSS, and JavaScript, and it offers not only syntax highlighting, context sensitive help, and auto-complete too.

If you are coming from the Java background, the chances are that you are familiar and comfortable with Eclipse IDE. In this case install the Eclipse plugin [VJET](#) for JavaScript support.

Oracle's IDE [NetBeans 7.3](#) and above support HTML5 and JavaScript development. NetBeans includes JavaScript debugger that allows your code to connect to the Web browser, while debugging inside the IDE.

If you prefer Microsoft technologies, they offer excellent JavaScript support in Visual Studio 2012.

Appcelerator offers a free Eclipse-based [Aptana Studio 3 IDE](#). Aptana Studio comes with embedded Web Server so you can test your JavaScript code without the need to start any additional software.

The authors of this book like and recommend using the **IDE WebStorm** from JetBrains. In addition to smart context sensitive help, auto-complete, and syntax highlighting it offers HTML5 templates, and the code coverage feature that identifies the code fragment that haven't been tested.

## Running Code Samples in WebStorm IDE

WebStorm IDE is pretty intuitive to use. If you've never used it before, refer to its [Quick Start Guide](#). When you first start WebStorm IDE, select the option *Open Directory* in the Welcome screen. Then select the directory where you downloaded the samples of a specific book chapter. For example, after opening code samples from Chapter 1 the WebStorm IDE may look as follows:

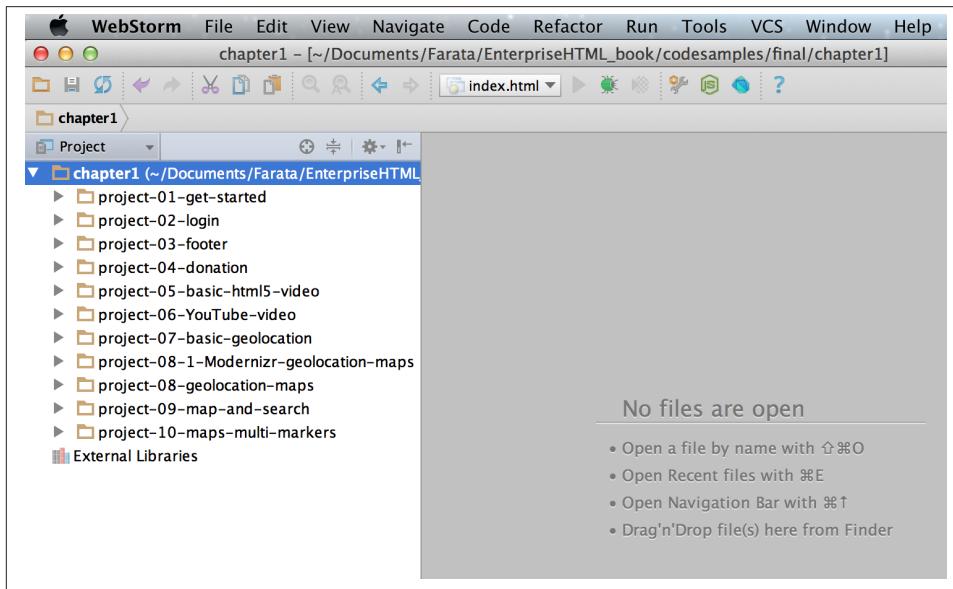


Figure C-1. Code Samples from Chapter 1 in WebStorm

If you want to create a new html or JavaScript file in WebStorm, just select the appropriate menu option under the menu File | New. For example, selecting the menu File | New | HTML File will create the following file with the basic markup:

```
<!DOCTYPE html>
<html>
  <head>
    <title></title>
  </head>

  <body>
```

```
</body>  
</html>
```

WebStorm IDE comes with a simple **internal Web server**. Right-click on the HTML file you want to open (e.g. index.html) and select the menu Open in Browser. WebStorm's internal server will serve the file to the browser.

For example, if the WebStorm's opened directory chapter1 as in Figure [Figure C-1](#) you'll see the following URL in your Web browser: <http://localhost:63342/chapter1/project-01-get-started>.



You can configure in WebStorm the port number of the internal Web Server via Preferences | Debugger | JavaScript | Built-in server port.

## Using two IDEs: WebStorm and Eclipse

Although we prefer using WebStorm for JavaScript development, but have to use Eclipse for some Java-related projects. In such cases we create a project in WebStorm pointing at the WebContent directory of your Eclipse project. This way we still enjoy a very smart context sensitive help offered by WebStorm, and all code modifications become immediately visible in the Eclipse project.

To open the content of Eclipse WebContent directory in WebStorm select its menu File | Open Directory and point it at the WebContent directory of your Eclipse project.

Mac users can also do it another way:

1. Create a script to launch WebStorm from the command line. To do this start Storm and open its menu Tools | Create Launcher Script. Agree with defaults offered by the popup window shown in [Figure C-2](#) or select other directory located in the PATH system variable of your computer. This will create a script named wstorm there, and you'll be able to start WebStorm from a command line.

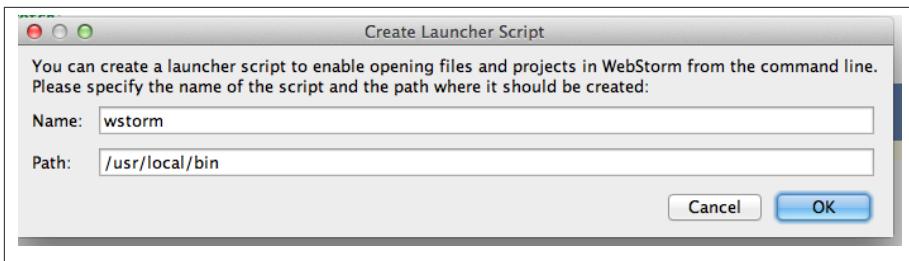


Figure C-2. Creating the launch script for WebStorm

2. Open a Terminal window and switch to the directory WebStorm of your Eclipse project. Type there the command `wstorm .`, and it'll open the WebStorm with the entire content of your WebContent project. So all JavaScript development you'll be doing in WebStorm, and the Java-related coding in Eclipse while using the same WebContent directory.

Such complex setup looks like an overkill, but we are talking about the enterprise development where you may jump through some hoops to create a convenient working environment for yourself. Besides, you do it only once.

---

# Index

*We'd like to hear your suggestions for improving our indexes. Send email to [index@oreilly.com](mailto:index@oreilly.com).*

## About the Authors

---

**Yakov Fain** is a co-founder of Farata Systems and SuranceBay companies. The first company provides consulting services in the field of enterprise Web development and e-Commerce, and the second one is a software product company, which develops software for the insurance industry. A leader of the Princeton Java Users Group, he has authored several technical books and dozens of articles on software development. Yakov received the title of Java Champion, which is presented to only 150 people worldwide. Yakov also holds an MS in Applied Math. You can reach him at [yfain@faratasystems.com](mailto:yfain@faratasystems.com) and follow him on Twitter @yfain.

**Dr. Victor Rasputnis** is a co-founder of Farata Systems and SuranceBay companies. He spends most of his time providing architectural design, implementation management, and mentoring to companies migrating to e-Commerce technologies with Hybris. Victor has authored several books and dozens of technical articles. He holds a PhD in Computer Science. You can reach Victor at [vrasputnis@faratasystems.com](mailto:vrasputnis@faratasystems.com).

**Anatole Tartakovsky** is a co-founder of Farata Systems and SuranceBay companies. He spent more than 25 years developing system and business software. In the last fifteen years, his focus has been on creating frameworks and business applications for dozens of enterprises ranging from Wal-Mart to Wall Street firms. Anatole has authored a number of books and articles on AJAX, Flex, XML, the Internet, and client-server technologies. He holds an MS in Mathematics. You can reach Anatole at [atartakovsky@faratasystems.com](mailto:atartakovsky@faratasystems.com).

**Viktor Gamov** is a senior software engineer at Farata Systems. He consults financial institutions and startups in design and implementation of Web Applications with HTML5 and Java. A co-organizer of the Princeton Java Users Group, Viktor is passionate about writing a code and about the open source community. He holds MS in Computer Science. You can reach Viktor on email [viktor.gamov@faratasystems.com](mailto:viktor.gamov@faratasystems.com) and follow him on Twitter @gamussa.

## Colophon

---

The animal on the cover of *FILL IN TITLE* is *FILL IN DESCRIPTION*.

The cover image is from *FILL IN CREDITS*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.