

Davy Mitchell, Sergey Akopkokhyants
Ivo Balbaert

Dart: Scalable Application Development

Master the art of designing web clients and servers with Google's bold, productive language—Dart



Packt

Dart: Scalable Application Development

**Master the art of designing web client and server
with Google's bold productive language – Dart**

A course in three modules

Packt

BIRMINGHAM - MUMBAI

Dart: Scalable Application Development

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this course by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Published on: November 2016

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78728-802-7

www.packtpub.com

Credits

Authors

Davy Mitchell
Sergey Akopkokhyants
Ivo Balbaert

Content Development Editor

Parshva Sheth

Graphics

Jason Monterio

Reviewers

Aristides Villarreal Bravo
Claudio d'Angelis
Joris Hermans
Jana Moudrá
Yan Cui
Predrag Končar
Martin Sikora

Production Coordinator

Shraddha Falebhai

Preface

Designed to create next generation apps, Google's Dart offers a much more robust framework and also supersedes JavaScript in several aspects. Familiar yet innovative, compact yet scalable, it blows away the accumulated JavaScript legacy limitations. Dart was designed for great tool-ability and developer productivity, allowing you to create better application faster than before. Google chose it for their billion dollar advertising business and you have its power for your projects too.

What this learning path covers

Module 1, Dart By Example,

This book will introduce you the Dart language starting from its conception to its current form, and where it headed is through engaging substantial practical projects. You will be taken through building typical applications and exploring the exciting new technologies of HTML5.

With example code projects such as a live data monitoring and viewing system, a blogging system, a slides presentation application, and more, then this book will walk you through step by step through building data-driven web applications with ease and speed.

Module 2, Mastering Dart,

Starting with a discussion about the basic features of Dart, we will dive into the more complicated concepts such as generics, annotation with reflection, errors and exceptions, which will help us improve our code. Moving on, you will learn how and when to create objects and also advanced techniques that will help you execute asynchronous code. You will also learn about the collection framework and how to communicate with the different programs written in JavaScript using Dart.

This module will show you how to add internalization support to your web applications and how i18n and i10n access can be embedded into your code to design applications that can be localized easily. You will be shown how to organize client-to-server communication and how different HTML5 features can be used in Dart. Finally, this book will show you how you can store data locally, break the storage limit, and prevent security issues in your web application.

Module 3, Dart Cookbook,

This module is a pragmatic guide that will increase your expertise in writing all kinds of applications, including web apps, scripts, and server-side apps. It provides rich insights on how to extend your Dart programming skills.

What you need for this learning path

A modern computer running Windows, Linux, or Mac OS will be sufficient to run the tools and programs in this book.

The details of the following and alternatives to WebStorm are discussed in Module 1, Chapter 1,

Starting the Text Editor:

- Dart SDK and Dartium (<http://www.dartlang.org>)
- WebStorm (<https://www.jetbrains.com/webstorm/>)
- Java (<https://java.com/en/download>)
- wrk – a HTTP benchmarking tool (<https://github.com/wg/wrk>)

For the database project, we are using PostgreSQL and pgAdmin:

- PostgreSQL (<http://www.postgresql.org/>)
- pgAdmin (<http://www.pgadmin.org/>)

To work with this Module 3 code, you need the Dart SDK and Dart Editor, which you can download from www.dartlang.org. Simply unzip the downloaded file and you are good to go. Because Dart Editor is based on Eclipse, you also need a Java Runtime (<http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html>). Choose the appropriate version for your system (32-bit or 64-bit); after the download, double-click on the .exe file to install it.

Who this learning path is for

If you are familiar with web development and are looking to learn, or even just evaluate, Dart as a multipurpose language, this learning path is for you. No familiarity with the Dart language is assumed. For beginners, it will serve as a guide to rapidly accelerate from a novice level to the master level; for intermediate to advanced developers, it will quickly fill in the gaps on Dart and can explore a range of application types and powerful packages that are demonstrated in a practical manner.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the course's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this course from.

7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged in to your Packt account.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Dart-Scalable-Application-Development>. We also have other code bundles from our rich catalog of books, videos, and courses available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our courses – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1: Dart By Example

Chapter 1: Starting the Text Editor

Defining Dart	3
History of Web scripting	4
The origins of Dart	5
Downloading the tools	6
Building your first application	10
Debugging a Dart application	20
Summary	23

Chapter 2: Advancing the Editor

The next steps for the text editor	25
Building the dialog package	27
The command-line app for source code statistics	40
Building web interfaces with Dart	47
Compiling to JavaScript	47
Summary	49

Chapter 3: Slideshow Presentations

Building a presentation application	51
Accessing private fields	56
Mixin' it up	56
Changing the colors	66
Adding a date	67
Timing the presentation	68
An overview of slides	70
Handout notes	71
Summary	73

Table of Contents —————

Chapter 4: Language, Motion, and Sound 75

Going fullscreen	75
Adding metadata	79
Exploring the <code>intl</code> package	81
Working with dates	87
Animating slides	89
Playing sound in the browser	90
Summary	93

Chapter 5: A Blog Server 95

The Hello World server example	95
A blog server	97
Introducing Dart's server frameworks	106
Deployment	108
Load testing	112
Summary	113

Chapter 6: Blog Server Advanced 115

Logging	115
A blog editor	117
Caching	123
Watching the filesystem	124
XML feed generation	125
The JSON feed generation	128
Static generation	131
Introducing the <code>await</code> and <code>async</code> keywords	131
Load testing revisited	134
Summary	135

Chapter 7: Live Data Collection 137

Kicking off the earthquake monitoring system	137
Introducing the data source	138
Exploring the GeoJSON format	138
Logging	140
Saving to the database	142
Introducing the pgAdmin GUI	144
Observing the Dart VM internals	153
Unit testing	154
Summary	158

Chapter 8: Live Data and a Web Service 159

Freeing the data	159
Creating the web service	164

Table of Contents

Recapping the system so far	170
Consuming application	171
Summary	178
Chapter 6: A Real-Time Visualization	179
Iteration overview	179
Application overview	180
Notifying the user of an update	191
Plotting the user's location	192
Sorting the feature list	194
Documenting Dart code with dartdoc	196
Summary	199
Chapter 7: Reports and an API	201
Recapping the earthquake system	201
Advancing the REST API	203
Passing parameters to the API	204
Posting on the API	205
Summary	218

Module 2: Mastering Dart

Chapter 1: Beyond Dart's Basics	221
Modularity and a namespace	221
Functions and closures in different scopes	227
Classes and mixins	231
Methods and operators	240
Summary	243
Chapter 2: Advanced Techniques and Reflection	245
Generics	245
Errors versus exceptions	253
Annotations	257
Reflection	263
Summary	268
Chapter 3: Object Creation	269
Creating an object	269
Summary	290
Chapter 4: Asynchronous Programming	291
Call-stack architectures versus event-driven architectures	291
Future	295

Table of Contents

Zones	299
Isolates	310
Summary	314

Chapter 5: The Stream Framework **315**

Why you should use streams	315
Single-subscription streams versus broadcast streams	317
An overview of the stream framework API	318
Summary	336

Chapter 6: The Collection Framework **337**

A Dart collection framework	337
Ordering of elements	338
Collections and generics	340
The collection class hierarchy	340
The Iterable interface	341
The Iterable interface	348
BidirectionalIterator	349
The collection classes	351
Unmodifiable collections	365
Choosing the right collection	367
Summary	368

Chapter 7: Dart and JavaScript Interoperation **369**

Interoperation at a glance	369
The dart:js library	370
Type conversion	374
JsObject and instantiation	379
JsFunction and the this keyword	380
Dart with jQuery	382
Summary	393

Chapter 8: Internalization and Localization **395**

The key principles	395
The Intl library	397
Internationalizing your web application	405
Extracting messages	409
Using Google Translator Toolkit	410
Using translated messages	412
Summary	414

Chapter 9: Client-to-server Communication **415**

Communication at a glance	415
Hypertext Transfer Protocol	417

Table of Contents

AJAX polling request	427
AJAX long polling request	430
WebSocket	439
Summary	444
Chapter 10: Advanced Storage	445
Cookies	445
Web Storage	453
Web SQL	456
IndexedDB	460
Summary	464
Chapter 11: Supporting Other HTML5 Features	465
The notification APIs	465
The native drag-and-drop APIs	471
The geolocation APIs	478
Canvas	486
Summary	502
Chapter 12: Security Aspects	503
Web security	503
Securing a server	508
Securing a client	509
Security best practices	526
Summary	527

Module 3: Modular Programming with PHP 7

Chapter 1: Working with Dart Tools	531
Introduction	531
Configuring the Dart environment	532
Setting up the checked and production modes	533
Rapid Dart Editor troubleshooting	536
Hosting your own private pub mirror	538
Using Sublime Text 2 as an IDE	539
Compiling your app to JavaScript	541
Debugging your app in JavaScript for Chrome	543
Using the command-line tools	545
Solving problems when pub get fails	547
Shrinking the size of your app	548
Making a system call	549
Using snapshotting	550

Table of Contents

Getting information from the operating system	551
<hr/>	
Chapter 2: Structuring, Testing, and Deploying an Application	555
Introduction	556
Exiting from an app	556
Parsing command-line arguments	557
Structuring an application	559
Using a library from within your app	562
Microtesting your code with assert	564
Unit testing a Polymer web app	565
Adding logging to your app	568
Documenting your app	571
Profiling and benchmarking your app	573
Publishing and deploying your app	575
Using different settings in the checked and production modes	576
<hr/>	
Chapter 2: Working with Data Types	579
Introduction	580
Concatenating strings	580
Using regular expressions	581
Strings and Unicode	583
Using complex numbers	584
Creating an enum	588
Flattening a list	591
Generating a random number within a range	592
Getting a random element from a list	593
Working with dates and times	594
Improving performance in numerical computations	597
Using SIMD for enhanced performance	600
<hr/>	
Chapter 3: Object Orientation	605
Introduction	605
Testing and converting types	606
Comparing two objects	608
Using a factory constructor	610
Building a singleton	614
Using reflection	615
Using mixins	619
Using annotations	621
Using the call method	623
Using noSuchMethod	624
Making toJSON and fromJSON methods in your class	627
Creating common classes for client and server apps	630

Chapter 4: Handling Web Applications 633

Introduction	634
Responsive design	634
Sanitizing HTML	635
Using a browser's local storage	637
Using application cache to work offline	640
Preventing an onSubmit event from reloading the page	643
Dynamically inserting rows in an HTML table	644
Using CORS headers	648
Using keyboard events	649
Enabling drag-and-drop	651
Enabling touch events	656
Creating a Chrome app	659
Structuring a game project	664
Using WebGL in your app	666
Authorizing OAuth2 to Google services	670
Talking with JavaScript	674
Using JavaScript libraries	678

Chapter 5: Working with Files and Streams 681

Introduction	681
Reading and processing a file line by line	682
Writing to a file	686
Searching in a file	687
Concatenating files	690
Downloading a file	693
Working with blobs	696
Transforming streams	698

Chapter 6: Working with Web Servers 703

Introduction	703
Creating a web server	704
Posting JSON-formatted data	707
Receiving data on the web server	710
Serving files with http_server	715
Using sockets	717
Using WebSockets	720
Using secure sockets and servers	726
Using a JSON web service	729

Table of Contents —————

Chapter 7: Working with Futures, Tasks, and Isolates 733

Introduction	733
Writing a game loop	734
Error handling with Futures	736
Scheduling tasks using Futures	741
Running a recurring function	743
Using isolates in the Dart VM	746
Using isolates in web apps	752
Using multiple cores with isolates	755
Using the Worker Task framework	757

Chapter 8: Working with Databases 761

Introduction	761
Storing data locally with IndexedDB	762
Using Lawndart to write offline web apps	766
Storing data in MySQL	769
Storing data in PostgreSQL	774
Storing data in Oracle	779
Storing data in MongoDB	782
Storing data in RethinkDB	786

Chapter 9: Polymer Dart Recipes 791

Introduction	792
Data binding with polymer.dart	792
Binding and repeating over a list	798
Binding to a map	800
Using custom attributes and template conditionals	802
Binding to an input text field or a text area	805
Binding to a checkbox	806
Binding to radio buttons	808
Binding to a selected field	810
Event handling	812
Polymer elements with JavaScript interop	814
Extending DOM elements	816
Working with custom elements	818
Automatic node finding	821
Internationalizing a Polymer app	823

Chapter 10: Working with Angular Dart 827

- Introduction 827**
- Setting up an Angular app 828**
- Using a controller 830**
- Using a component 834**
- Using formatters as filters 838**
- Creating a view 841**
- Using a service 843**
- Deploying your app 845**

Bibliography 849

Module 1

Dart By Example

Design and develop modern web applications with Google's bold and productive language through engaging example projects

1

Starting the Text Editor

The rung of a ladder was never meant to rest upon, but only to hold a man's foot long enough to enable him to put the other somewhat higher.

– Thomas Huxley

Defining Dart

Dart is a language and platform for modern web applications that can run both in the web browser and on the server. The Dart language, tools, and API allow innovative, productive, enlightened, and talented developers (that's you!) to write scalable web applications that make the best of the modern Web.

With Dart, you can take a leap forward in web development and tooling. It is a clean, modern, yet familiar language syntax that runs on a platform created by the world's leading virtual machine experts.

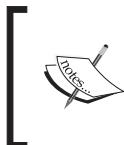
Dart is run as an open source project with a defined process for enhancement proposals. It has flexible libraries with common documentation packaging, unit testing, and dependency resolution. The language became an ECMA standard in July of 2014, and does not require any plugin for end users to run as it is compiled to JavaScript.

If that is not enough, you can bring your existing HTML, CSS, JavaScript skills, and even code along for the ride—Dart plays nicely with others!

History of Web scripting

The high sophistication of current web pages with animations, dynamic content, fades, 3D effects, responsive designs, and clever navigation make it easy to forget that the early web was mostly textual pages, dumb forms, and images that often took a while to load. Then, along came JavaScript, in the form of a script interpreter built into the browser, providing form data validation, news ticker moving displays, animation, and games. For small projects, it succeeded in spicing up static websites without requiring server-side CGI scripts.

Developers enjoyed the near instant edit and refresh cycle—changing a line of code and hitting *F5* (**refresh**) in the browser to see the result. JavaScript did not stay in the browser and was soon found on the server side of web applications. It also became a general purpose script for use outside the browser.



Fun fact: **JavaScript** was written in just ten days by Brendan Eich for the Netscape browser and was originally called **LiveScript**. Dart has been renamed too—originally, it was called Dash.



Considering the timescale it was written under, JavaScript is a great technical achievement, but in 20 years it has not advanced very much, while web applications have rapidly progressed. Web applications can contain thousands of lines of JavaScript code. Outside of very simple pages, plain JavaScript is not enough anymore, as evidenced by the number of tools and libraries that have sprung up to assist development.

Many of these solutions are created to fix problems with JavaScript, ranging from syntax and features to design and productivity. The language simply was not designed for the type of web application that the modern web requires.

Recent advances in JavaScript engines have produced great leaps forward in performance. The V8 engine that powers the Chrome browser and `Node.js` has shown great improvement in making new kinds of applications viable. However, the returns from JavaScript virtual machine optimizations have been diminishing over time.

The origins of Dart

Google has a lot of experience with both large web applications and writing web browsers. They clearly have a strong self-interest in a better web platform (so people search more) and an improved developer productivity (to stay ahead of the competition). It is mentioned in Google presentations that a single code change in their Gmail web application takes around 20 minutes to rebuild the site for the developer to test it out.

This harks back to software build times decades ago. The project to fix this problem was started, and Google wanted to share and work with the development community as an open source project.



In 2011, at the GOTO conference, the Dart language and virtual machine was unveiled to the world. Dart is designed to be a "batteries included" project—a complete stack for writing, compiling, testing, documenting, and deploying web applications.

Developed by the Chrome team, the project was founded by Lars Bak (the developer of the Java HotSpot VM and JavaScript V8 Engine) and Kasper Lund (a V8 developer). The aim was both to improve the open web platform by opening up new avenues for high performance client web applications and to improve developer productivity.

The upstart language was designed to have a familiar 'curly brackets' syntax similar to Java, C-sharp, and JavaScript, run on both the client and the server, and to support the full range of modern web browsers by being able to compile to regular JavaScript. New language features were only added to Dart if they could be compiled satisfactorily to JavaScript.

Dart is often referred to as **DartLang** to avoid confusion with other "darts." Keep this in mind when searching the Web for better results. The Dart language, like JavaScript, is not only meant for the web browser; it is also available for server applications and command-line applications. Future targets are mobile applications on iOS and Android.

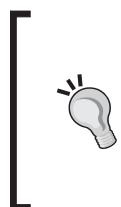
That is the history, the challenge, and the reaction of the biggest Internet company in the world. So, what is Dart all about, then? The remainder of this chapter will compare and contrast Dart and JavaScript and take you into building your first Dart application so that you can see for yourself.

The rest of this book will take you on a tour of Dart through a set of interesting projects, exploring all of Dart's habitats. We will be building useful applications straight away and using increasingly powerful features.

Downloading the tools

Let's get started by installing a complete Dart development environment on your computer. The home of Dart on the Internet is <https://www.dartlang.org>, which contains the software, a wealth of documentation, and links to the online Dart community.

I would strongly recommend signing up for the e-mail lists on this site to keep up with Dart, and also the Dartisans Google+ community (<http://g.co/dartisans>). The number of daily messages can be overwhelming at first, but there are some great discussions and information sharing. It is also a great opportunity to interact with Dart's creators.

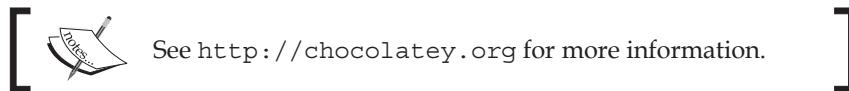


It is recommended that you use the Stable channel. This version updates roughly every few months via a built-in update tool. There is also a **Dev (development)** channel that updates once or twice a week with the very latest code from the Dart developers. This is great for seeing new features early, but it is not as polished or reliable as the Stable releases.

The downloads for all supported platforms (Linux, Mac, and Windows) are available at <https://www.dartlang.org/tools/download.html>. The same location provides information regarding building from source. Ensure that you download both the SDK and Dartium (a specialist browser).

It is recommended that Mac users use the Homebrew system to manage the installation and updates of Dart. Linux users can use a .deb if that format is supported on their distribution, with ZIP archives being the alternative. Windows versions are also supplied as ZIP archives.

Windows users can download, install, and update the SDK and Dartium using the Chocolatey installation system that is built on nuget and Powershell.



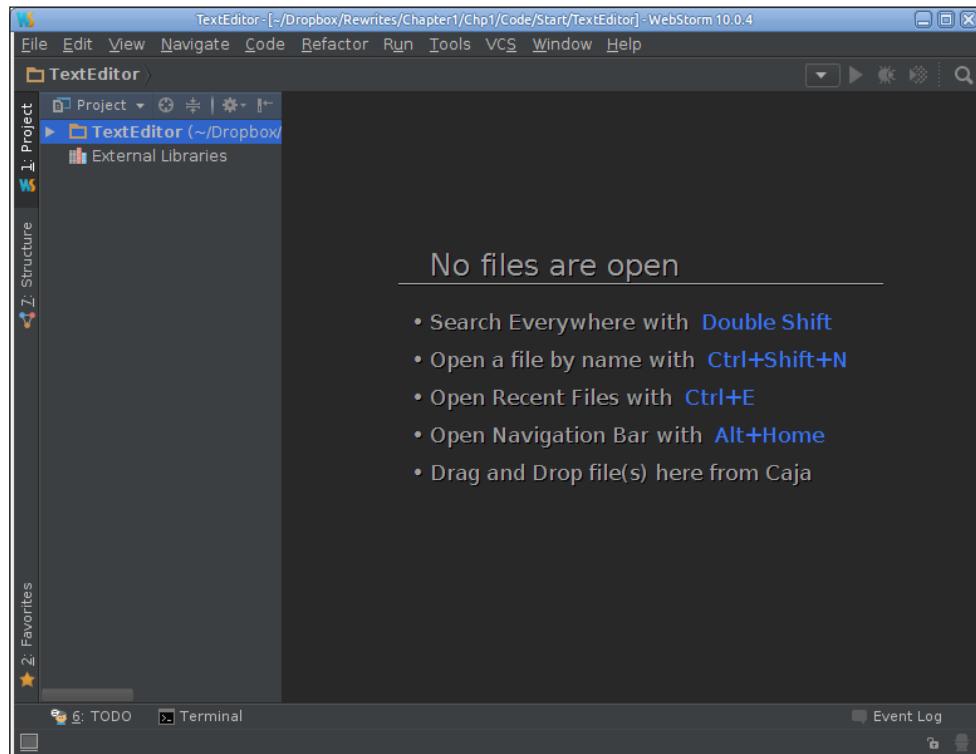
The Dart packages are called `dart-sdk` and `dartium`.

Introducing the WebStorm IDE

WebStorm is the premiere development environment for Dart and was created by JetBrains, which also publishes IntelliJ IDEA for Java and ReSharper for .Net.

WebStorm is a commercial package available at <https://www.jetbrains.com/>. There are free licenses available for many and deep discounts for individuals. It supports a range of languages and development platforms and Dart support is provided via a plug-in. The plugin comes pre-installed with the IDE.

Download the installation file and follow the installation steps on screen:



WebStorm has a number of powerful features to help write our web applications:

- Auto-completion of code: properties and methods are listed as you type
- A static analyzer that scans as you type for possible problems
- Refactoring tools to help rename entities and extract methods
- Quick fixes to correct common errors easily
- Navigation via a code tree and finding usages of a function
- A code reformatter to keep the source code in a tidy format
- A powerful debugger

Alternative development environments

If you wish to use another IDE, there are plugins for IntelliJ IDEA and Eclipse. If you prefer a simple text editor, there are plugins for Sublime, Emacs, and Vim. These are all listed on the DartLang download page.

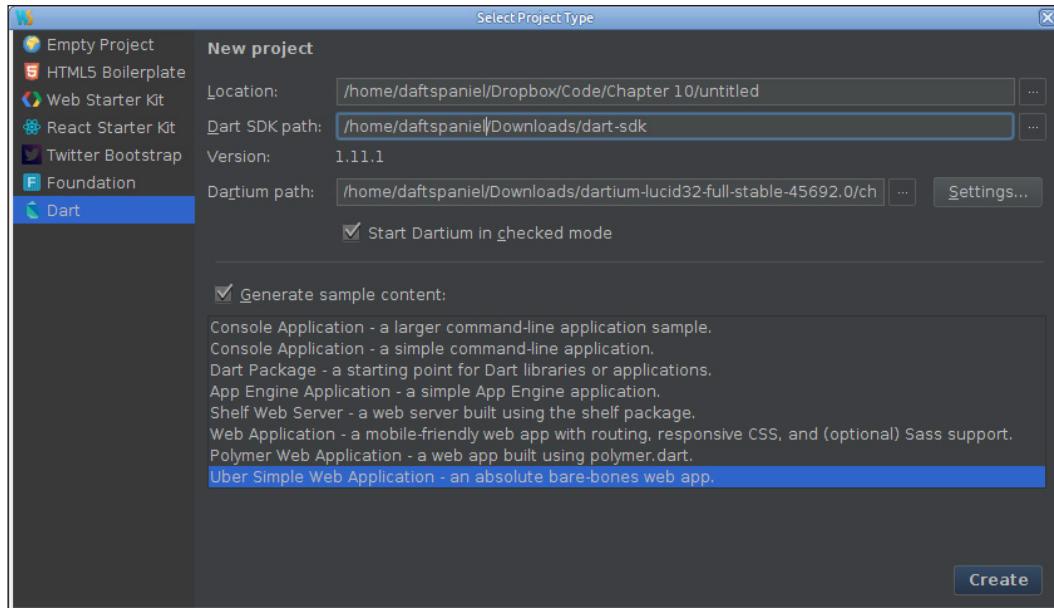
If you are an avid Visual Studio user, there is a community project underway to bring Dart to that environment. DartVS can be downloaded from the Visual Studio Gallery. It directly uses the DartAnalyzer from the SDK to report errors, warnings, and hints inside the IDE. Support for intellisense and other features is underway, too.

The Atom Editor (<https://atom.io/>) is an alternative for those who prefer a lightweight editor to a fully fledged development environment. The plugin, named dartlang (<https://atom.io/packages/dartlang>) provides code-completion, syntax highlighting, and access to some pub commands.

Help starting a project

To help create your new project, Dart has a tool called Stagehand that creates the scaffolding from a high quality project template of your choice. It takes care of common tasks such as folder structure, common libraries, and style sheets. It even has its own website (<http://stagehand.pub/>).

In WebStorm, creating a new project presents you with a range of powerful options, allowing you to set the locations of the Dart SDK and choose a project type for sample content. The project type gives a range of skeleton applications and is powered by the Stagehand application that is part of the SDK:



Stagehand can also be run from the command line, with the first step being to install it as a command:

```
pub global activate stagehand
```

This step installs the `stagehand` command from the `stagehand` package:

```
mkdir aproject
cd aproject
stagehand web-full
```

This will create a folder of sample content based on the `web-full` template.



For full details of the current templates, see

<https://pub.dartlang.org/packages/stagehand>.

Elsewhere in the SDK

The download also includes a set of command-line tools, which can be found under the `dart-sdk/bin` path.

Tool	Description
<code>dart</code>	The standalone Dart Virtual Machine itself.
<code>dart2js</code>	The tool that converts Dart into JavaScript so that it can run on any modern browser.
<code>Dartanalyzer</code>	Analyzes the Dart source code for errors and suggestions. This is also used by development environments to provide feedback to users.
<code>Dartdocgen</code>	The documentation generator that works directly on Dart code.
<code>Dartfmt</code>	The intelligent code formatter that makes your code tidy and consistent.
<code>pub</code>	The package management and application deployment tool.



You may wish to place these tools on the system PATH so that you can call them from anywhere on the command line. This is optional as WebStorm provides an easy interface to these tools; however, many developers find it essential to a good workflow.

For more detailed information on the installation and configuration of `pub`, see

<https://www.dartlang.org/tools/pub/installing.html>

Also in this location is a `snapshots` folder. Most of the tools are written in Dart itself; when they are started up and initialized, a binary image of the initialized state can be stored and used for future executions of the program.

The snapshots are generated using the Dart executable and are mostly intended for frequently used command-line applications in deployment.

Building your first application

Starting simple is a good idea, but we also want to start with something useful that can be expanded as we explore Dart.

In day-to-day software development, the text editor is where gigantic, world-changing software often starts. We are going to build a multi-purpose web-based text editor to use as a scratchpad for snippets of code, a to-do list, or stripping formatting from copied text – immensely handy. For added realism, there will be an imaginary customer along the way asking for new features.

In the next chapter, we will build on this foundation for some flashier features and graphical displays.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Exploring the Web project structure

Go to the WebStorm welcome page and click on **Open**. Select the folder where you extracted the sample code and choose the `Chapter One` folder, then the sub-folder named `Start`.

A good first step when opening any Dart project is to ensure all dependencies are present and up to date. To achieve this, select `pubspec.yaml` and right-click to show the context menu. Click on the `Pub: Get Dependencies` item. The dependencies will then be resolved and fetched if required.

File / Folder	Description
<code>packages</code>	A link to the packages (or libraries) that are being used by this project. In this case, you will see the 'browser' package.
<code>pubspec.lock</code>	Controls the specific version of the packages being used.
<code>pubspec.yaml</code>	Meta information on our project (name, description, and so on) and a list of packages being used.
<code>web/styles/main.css</code>	The standard cascading style sheet file.
<code>web/index.html</code>	The entry point for our web application.
<code>main.dart</code>	Last but not least, a Dart file!

Unwrapping packages

From the preceding list, there is clearly a lot of structure and meta files for packages in even a simple Dart project. The philosophy of the language is that libraries are small, focused, and likely to be from different sources. They are also numerous and update on different schedules, so version controls are important—dependencies are hungry and must be fed.

You want your team members to be able to build and use your new Uber package without dependency horrors! Likewise, you want the package you download off the Internet to "just work" and reduce the effort required to write your application.

To look after all these packages, Dart has a tool called pub. This is a Swiss Army knife tool that helps create, manage, and deploy your programs and packages. The main home for Dart packages is the website <https://pub.dartlang.org>, where numerous open source libraries are published.



If you are curious to know the origin of the name, "pub" is British slang for a bar where people drink and play a game of darts.



A look at Pubspec

The `pubspec.yaml` has a curious file extension—**YAML** stands for **Yet Another Markup Language**:

```
name: 'TextEditor'  
version: 0.0.1  
description: A web based text editor.  
environment:  
  sdk: '>=1.0.0 <2.0.0'  
dependencies:  
  browser: any
```

From this, it is clear to see `pubspec.yaml` is Dart's equivalent of a project or solution file. This holds the typical dependency details and meta-information. The `Pubspec` file is fully detailed at <https://www.dartlang.org/tools/pub/pubspec.html>. Although it is an extensive and powerful file, the main interaction you will probably have with it is adding new dependencies.

Putting Dart into the web page

The file `index.html` is a regular web page. The rest of the file consists of a few simple `<div>` elements and, as you might expect, a `<textarea>` tag:

```
<!DOCTYPE html>  
<html>  
<head>  
  <title>TextEditor</title>  
  <link rel="stylesheet" href="styles/main.css">  
</head>
```

```

<body>
<div id="output">

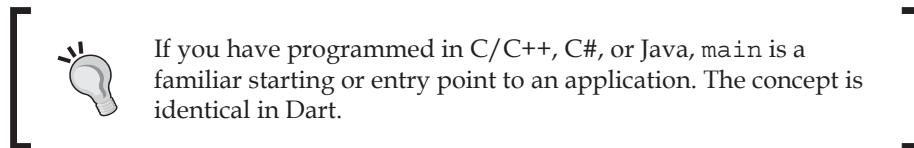
    <div id="toolbar">
        TextOnTheWeb
    </div>

    <textarea id="editor" cols="80" autofocus>
    </textarea>

</div>
<script type="application/dart" src="main.dart"></script>
<script data-pub-inline src="packages/browser/dart.js"></script>
</body>
</html>

```

The foot of this page is where the initial Dart file is specified. The `main.dart` file is loaded and the entry point, the first function to be run, is the `main()` function.



Let's work through the `main.dart` file section one at a time and discover some Dart features along the way.

Importing packages

Dart features a keyword, `import`, to use other packages and classes in our current code context. In this case, both are prefixed with `dart:` to signify they are from the SDK:

```

import 'dart:html';
import 'dart:convert' show JSON;

```

The keyword `show` modifies the second import to only make the `JSON` property sections of the `dart:convert` package. This limits the classes in the current namespace and therefore helps avoid name clashes and developer confusion.

Variable declarations

The `TextArea` is central to our editor and will be referenced multiple times; therefore, we will use a specific type for it rather than `var`. In contrast, JavaScript declares variables with the all-encompassing `var` keyword.

```
TextAreaElement theEditor;
```

This declares a new variable called `theEditor` and declares it as the type `TextAreaElement`. The `dart:HTML` package contains classes to cover the DOM and the `TextArea` input element.

Dart has an optional type system, so it would be possible to declare `var theEditor` and the program would still run successfully. By being specific with the type, we gain clarity in the source code and we provide more detailed information to developer tools, such as code editors and documentation generators:

```
void main() {  
    theEditor = querySelector("#editor");  
    theEditor  
        ..onKeyUp.listen(handleKeyPress)  
        ..text = loadDocument();  
}
```

In the `main` function, we use the `querySelector` method from `dart:HTML` to connect to the `TextArea` control (with the `ID` attribute set to `editor`) on the web page. Once we have a reference to the control, we want to a) connect an event handler so that we know when the user has typed something, and b) load any existing text and display it.

Because we are using the same object, we can use the cascade operator (`..`), which helps us write very readable and flowing code. The preceding can be written as:

```
theEditor.onKeyUp.listen(handleKeyPress);  
theEditor.text = loadDocument();
```

The more properties we set, the more we have `theEditor` cluttering up the code. With the cascade operator, we can replace the object name with `(..)`, and call the method/set a property as before. One important point with the cascade operator is that only the final use has a semi-colon at the end of the line.

Writing the event handler

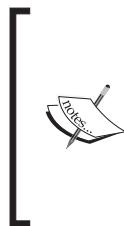
The editor's `TextArea` `KeyUp` event has been connected to this short handler function:

```
void handleKeyPress(KeyboardEvent event) {
    saveDocument();
}
```

In contrast to JavaScript, there is no `function` keyword and we have `void` before the function. The `void` keyword just means that this function does not return anything when it is finished; it can be any class or type. As Dart is optionally typed, we could omit `void` altogether.

Just a minute! Where are we going to be storing this text? After all, we have only written a single HTML page and do not have a database connection or web service to take care of the persistence. Fortunately, HTML5 has a simple key/value based built-in storage feature called Local Storage that is well-supported by modern web browsers. This operates in a similar fashion to a dictionary (sometimes called a map) data structure.

In the next two functions, we will look at loading and saving from `localStorage`.



The HTML5 feature `window.localStorage` persistently stores data with the client's browser. Unlike data stored in cookies, it does not expire based on a date. The data is not lost when the browser window is closed or the computer is switched off.

The amount of storage can vary per browser and according to user settings, but the typical default value is 5 MB. This is plenty for our text editor and many other types of applications.

Loading the saved text

The `loadDocument` function lets the world know it will be returning a `String` object:

```
String loadDocument() {
    String readings = "";
    String jsonString = window.localStorage["MyTextEditor"];
    if (jsonString != null && jsonString.length > 0)
        readings = JSON.decode(jsonString);
    return readings;
}
```

We will store the text under the key `MyTextEditor`. The first time the user loads up this page, there will be nothing in the local storage for our web page, so we will check if it is empty or null before putting the value into the `readings` string variable.

 **JSON (JavaScript Object Notation)** is an ECMA standard data format that is based on a subset of JavaScript and is an alternative to XML.

For example:

```
{  
  "Name": "Davy Mitchell",  
  "LuckyNumber": 123,  
  "CarModel": null,  
  "Language": "English"  
}
```

Saving the text

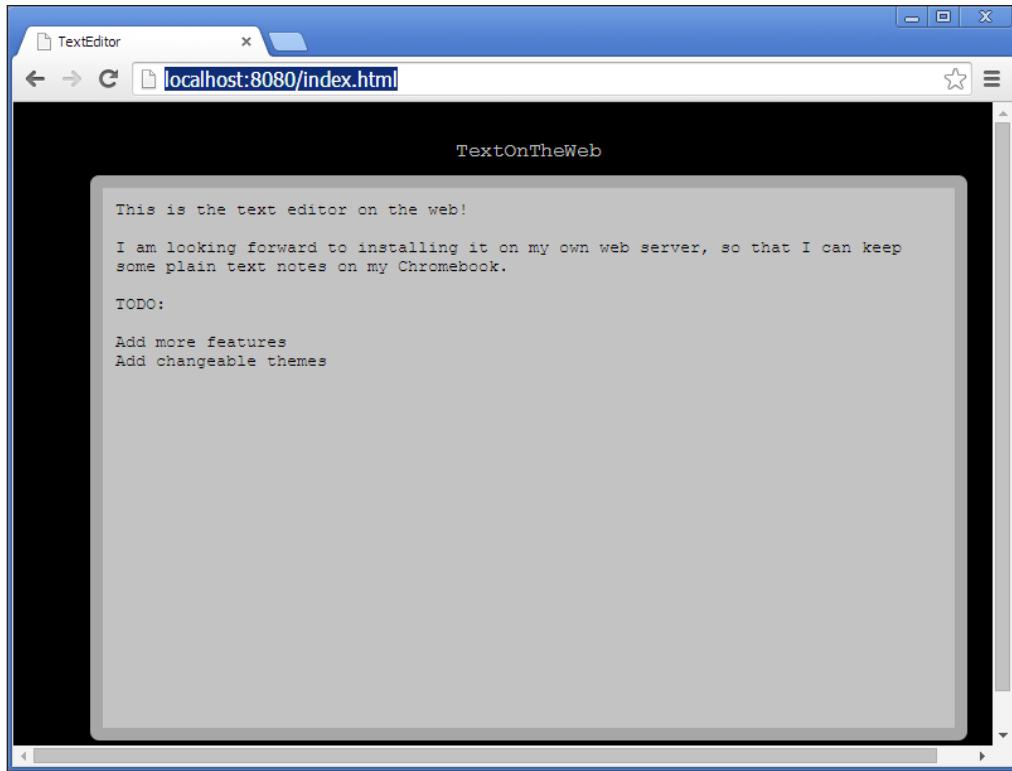
The format we are using to save the text is JSON – JavaScript Object Notation – which is a common standard in web pages. The package `dart:convert` gives us both the encode and decode functions:

```
void saveDocument() {  
  window.localStorage["MyTextEditor"] = JSON.encode(theEditor.value);  
}
```

Finally, the saved document uses the local storage to keep our text safe. This time, we are encoding the data into the JSON format.

Running in the browser

We are now finally ready to actually run the text editor in the target environment. How is a browser going to execute a `.dart` file? Internet Explorer, Chrome, and Firefox don't speak Dart.



Dartium is a special build of the Chromium browser (the open source project behind Google's Chrome browser) that contains the Dart virtual machine. Though not recommended for day-to-day browsing, it is a full powerful modern browser to use while developing Dart applications. As shown in the screenshot, it looks just like (a very plain) Google Chrome.

If you are running on Ubuntu and receive a start up error mentioning libudev.so.0, run the following command in a terminal to resolve it:

```
sudo ln -s /lib/x86_64-linux-gnu/libudev.so.1 /lib/
x86_64-linux-gnu/libudev.so.0
```

For more information, see <https://github.com/dart-lang/sdk/issues/12325>

In WebStorm, right-click on the index.html file to bring up the context menu and select the run index.html. This will start a local HTTP server for your application.

Once the application is running, type some text into the editor, and then close the browser window. Then, run the program again and you should see your text reappear once the page has loaded.

Editing and reloading

A new requirement has arrived from our customer. The title of the application, "TextOnTheWeb", is not to their liking, so we will have to change it.

By this point, Dart should be looking a lot more familiar to you if you have written JavaScript. The workflow is similar, too. Let's make a change to the executing code so that we can see the edit and reload cycle in effect.

This will be achieved in the main function so, while keeping Dartium open, open the `main.dart` file again in WebStorm, and locate the `main` function.

Add the following code snippet to start of the function:

```
DivElement apptitle = querySelector("#toolbar");
apptitle.text = "TextEditor";
```

Once you have made the change, save the file and switch back to Dartium. Refresh the page and the new version of the main function will run and the new title will be displayed.

Extending the interface

Our customer has asked for a button to clear the editing area. This is a reasonable request and should not take too long to code with the productivity of Dart.

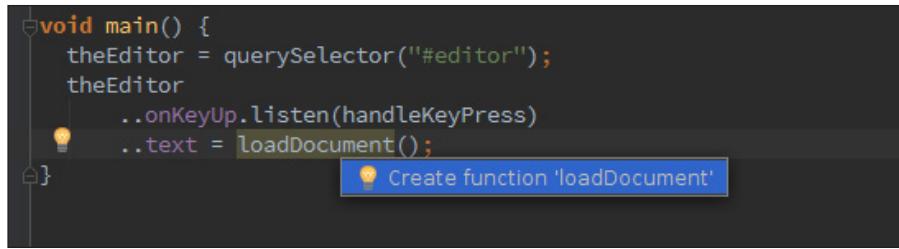
Firstly, we will open the `index.html` and add an input button to the page underneath the `TextArea` control:

```
<input type="button" id="btnClearText" value="Clear" />
```

Open `main.dart` and in the `main` function, connect up an event handler:

```
ButtonInputElement btnClear = querySelector("#btnClearText");
btnClear.onClick.listen(clearEditor);
```

As with JavaScript, it is very easy to pass functions around as parameters. Move the cursor over the text `clearEditor` and a lightbulb will appear. This offers the keyboard shortcut of *Alt* and the *Enter/Return* key, or you can click on the lightbulb icon.



The implementation of the function is straightforward. Notice that we receive a properly typed `MouseEvent` object as part of the event:

```

void clearEditor(MouseEvent event) {
    theEditor.text = "";
    saveDocument();
}

```

Refresh the page and click the button to clear the existing text. One happy customer and we hardly broke a sweat!

The WebStorm Dart plugin has a number of quick fixes to aid your productivity and help avoid coding errors in the first place. They also encourage good design and keeping to Dart conventions.

Using the CSS editor

Keep the text editor running for the moment and switch back to WebStorm. Go to the **Project** tab and in the `TextEditor` folder, open the `web/styles/main.css` file.

This is going to be your editor, too, so it is important that it is your favorite color—perhaps you would like a thinner border or are not too fond of my chosen gray palette.

Make some changes to the color classes and save the file.

The WebStorm CSS editor is powerful and shows previews of colors. If you switch back to Dartium and click reload, the text editor should now reflect your chromatic preferences.

Debugging a Dart application

A good debugger is a key part of a productive development system. Let's step through our code so that we can look closely at what is going on.

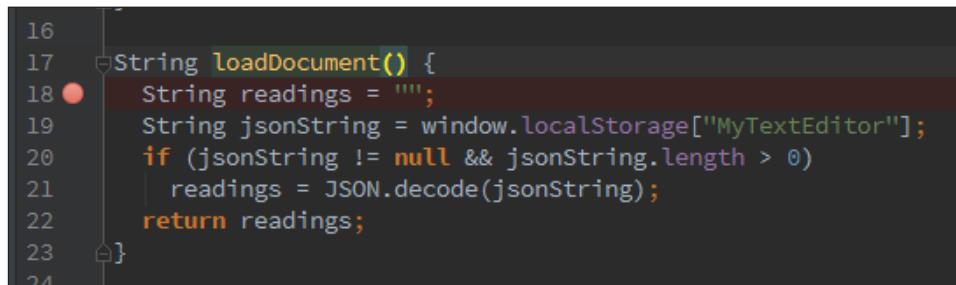
To debug an application running in Dartium from WebStorm, we need to install a small browser extension to act as a bridge between the two applications.



Follow the guide from JetBrains at the following web page:
<https://www.jetbrains.com/webstorm/help/using-jetbrains-chrome-extension.html>

The main section to be concerned with is the "Installing the JetBrains Chrome extension" section. This only has to be done once.

1. Click on the first line of the `loadDocument` function.
2. Next, open the **Run** menu and choose **Toggle Line Breakpoint**. A red circle will appear to the right of the line:



```
16
17  String loadDocument() {
18  ●   String readings = '';
19  String jsonString = window.localStorage['MyTextEditor'];
20  if (jsonString != null && jsonString.length > 0)
21      readings = JSON.decode(jsonString);
22  return readings;
23  }
24
```

3. Select `index.html` in the **Project** tab, then open the **Run** menu and choose the `Debug index.html` menu.
4. Once Dartium opens, the `loadDocument` function should run and the breakpoint should hit. The **Debugger** tab will appear in WebStorm.
5. The **Debugger** tab shows the call stack and values of current variables. Under the **Run** menu and on the toolbar, the usual debug operations of **Step Into**, **Step Over**, and **return** are available.
6. Before we take any steps, hover the pointer over the return statement line. A tool-tip will appear to show that the string variable `readings` is currently set to `null`.

7. Choose **Step Over** and the execution will move onto the next line.
8. Move the pointer over the return statement again, and the readings variable is shown to be an empty string object.
9. **Step Over** until the end of the function, and the return variable will be set to the text retrieved from local storage.
10. To get the application running again, use the **Resume Program** menu option from the **Run** menu, or to stop it from running, select the **Terminate** menu option.

Working in harmony with JavaScript

The clear button on the editor is a bit dangerous as it may wipe out some vital notes. We will provide the user with a simple *Are you sure?* prompt so that they have a chance to back out of the operation.

You are probably thinking that we could use the Dart equivalent of `window.confirm` to carry it out. We certainly could, but to demonstrate the ability to call JavaScript, we will use the non-Dart version to display a prompt to the user.

Open `main.dart` and add the following import to the top of the file:

```
import 'dart:js';
```

In the **Dart Analysis** tab directly below the Dart code editor window, you will see a warning that we have an unused import. This can be a useful tip once a project has grown and code has been moved around into separate packages and classes. Import lists can soon acquire clutter.

To call the JavaScript confirm dialog, we use the context object from `dart:js` in the button click event handler. The context object is a reference to JavaScript's global object:

```
void clearEditor(MouseEvent event) {
    var result = context.callMethod('confirm',
        ['Are you sure you want to clear the text?']);
    if (result == true) {
        theEditor.text = "";
        saveDocument();
    }
}
```

The `callMethod` method can be used to call any JavaScript function available in the scope of the page—not just built in objects. In this case, the first parameter is the name of the function we wish to call and the second parameter is the list of arguments for that method.

Commenting in the code

Our text editor foundation is looking complete at this point, but there is one important element that is missing from the `main.dart` file—code comments.

Dart uses the following familiar commenting syntax:

Comment Syntax	Description
<code>// Your comment here.</code>	A single line comment
<code>/** * Your comment here. */</code>	A multiple line comment
<code>/// Your comment here.</code>	A doc comment (the preferred method because it is more compact)

In mentioning an identifier, place square brackets around the name. For example:

```
/// Returns double of [a] input.  
int doubleANumber(int a){  
    //Assumes parameter valid.  
    return a * 2;  
}
```

Take a short time to comment each function with the above style. Use a sentence format and punctuation.



For more information on comment style and other coding conventions, see the guidelines for Doc comments:

<https://www.dartlang.org/articles/doc-comment-guidelines/>, and the Dart coding style guide: <https://www.dartlang.org/articles/style-guide/>

Summary

This chapter was focused on giving you the background story of Dart and getting to work with the SDK to produce a useful application.

We have discovered how the JavaScript language and its development limits, leading to the creation of the Dart open source project (centered around the <https://www.dartlang.org> website), which is being developed as an ECMA standard and can be used to write a range of application types from client to server and command line.

We have seen that Dart has a familiar syntax and powerful package management tool called pub. WebStorm can be used to create, launch, and debug different types of applications, and other IDEs and text editors have Dart language support, too.

We have worked through setting up a Dart development environment and wrote our first application using HTML5 features. We saw how to navigate the structure of a client-side web project and carry out debugging and development.

I am certain that the simple text editor that we have created is firing off ideas in your mind of what to do next! In the next chapter, we will continue to look at client-side Dart and add more features to the text editor, including some that will help us write more, and better, Dart code.

2

Advancing the Editor

We made the buttons on the screen look so good you'll want to lick them.

– Steve Jobs

The technical possibilities of ever shinier web designs and more smoothly animated layouts that modern web browsers are capable of makes the challenge of balancing features and attractive presentation even harder for web application developers.

Functional and non-functional requirements, such as interface design and performance, are often conflicting priorities. There is only a finite amount of time for any project. A good strategy (sometimes!) is to focus on those that deliver both, and there are many new possibilities within the expanding HTML standard.

Once the requirements are all settled and the application is constructed, deploying the applications to the production environment smoothly is key. Testing on our target systems, such as a range of web browsers, is vital, and that is where we are heading with this chapter.

The next steps for the text editor

We have had some feedback on the text editor so far. Users like it and want more buttons, and the top priority is a word-count feature that displays a dialog with the current count. Customers have indicated that they would like a powerful toolbar above the editor pane for a range of features. They like the pop-up dialog style, but say that the current one does not fit into the look of the application.

Starting point

Open the code sample for *Chapter 2, Advancing the Editor* `texteditor` to see a slightly modified version of the project from the first chapter. It is not ready to run out of the box as we will add in a package to deal with the dialog creation.

The text editor functionality has been moved into a separate file—our `main.dart` was getting rather cluttered. My personal preference is to keep it as minimal as possible. The `main.dart` now imports a file, `editor.dart`, and the `main` function is used to connect the interface to the `Editor` object, which now contains our functions as methods of the `TextEditor` class.

Dart is an object-orientated language like C# and Java. In object-orientated languages, objects can be defined as class declarations that typically group together variables and functions to model an aspect of a system. For more information on object-orientated programming, see https://en.wikipedia.org/wiki/Object-oriented_programming.

For example, web browsers have the `document` and `window` classes that can be accessed from JavaScript. These objects encapsulate aspects of the web page structure and web browser window in functions and properties.

Even if you have not written classes with JavaScript, you will likely have called functions and accessed properties on the built-in browser page objects such as the `document.getElementById()` method and the `document.body` property.

Dart classes

We have already encountered classes for HTML elements and event handlers. Dart has an advanced single-inheritance class system. Classes are declared with the `class` keyword. The `editor.dart` file has a class for the `TextEditor` (an abbreviated version is shown below):

```
class TextEditor {  
    final TextAreaElement theEditor;  
    ...  
    TextEditor(this.theEditor)  
    ...  
}
```

The constructor is declared as the same name of the class, and we have a single parameter. Dart constructors have a convenient feature for directly setting member variables from the parameter declaration. This saves having a separate parameter that is only ever used to assign to the member variable. Without using this shorthand feature, the code would be as follows:

```
class TextEditor {  
    final TextAreaElement theEditor;  
    ...  
    TextEditor(TextAreaElement theEditor) {  
        this.theEditor = theEditor;  
    }  
    ...  
}
```

The field `theEditor` is declared as `final`; this means it can only be assigned a value once in the lifetime of the application.

Structuring the project

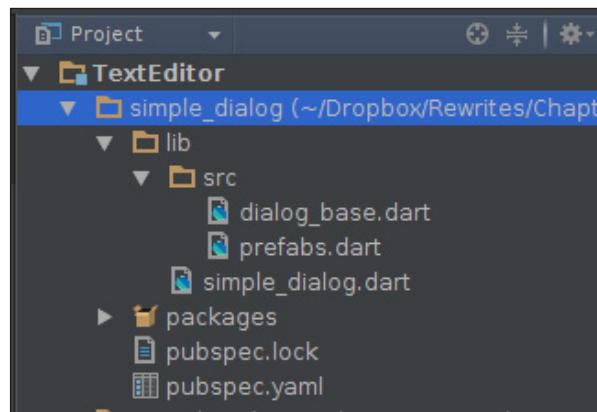
Our new requirements are for two pop-up dialogs, and it sounds like we may need more. We need to build an HTML interface library to cover these needs. It is general-purpose, so can be used in many types of application. The rest of the application can go in the `TextEditor` project.

Building the dialog package

The `pub` packaging system is a great resource for sharing finished packages online. If we want to keep a package local and private, we can create it locally and use it in much the same manner.

The package project structure

The package project is structured in a similar manner to a web application project, with a `pubspec.yaml` file being the main project file. Instead of a `web` folder, there is a `lib` folder containing the `simple_dialog.dart` file that declares our library and defines what is exposed outside the project:



This file contains any classes, functions, and references to other files—in this case, the two files in the `src` folder. The first line in `simple_dialog.dart` with the keyword `library` states the name and declares that this is a library. Note that it is not surrounded by any quotation marks:

```
library simple_dialog;
import 'dart:html';
part 'src/dialog_base.dart';
part 'src/prefabs.dart';
```

The next section contains the imports required for all the files that are being exposed by this package declaration in the part declarations. Open either of the files in the `src` folder and you will see `part of simple_dialog` declared on the first line.

Adding a local package reference

To reference the library, go to the `TextEditor` project's `pubspec.yaml` and open it in the editor. Click on the **source** tab to see the plain `yaml` text file. There is currently no graphical interface for the local reference, so it has to be added by hand:

```
dependencies:
  browser: any
  intl: any
  simple_dialog:
    path: ..\simple_dialog
```

The path is relative to the root project directory, and, as this is a local filesystem reference, the path separator must match the operating system, so we use `\` (backslash) for Windows and `/` (forward slash) for Linux and Mac.

The package is then referenced in `editor.dart` with a package prefix:

```
import 'dart:convert';
import 'dart:html';
import 'package:simple_dialog/simple_dialog.dart';
```

It operates in exactly the same way as a remotely installed package, with the added advantage that it automatically refreshes if changes are made.

To avoid any problems with relative paths (which make many people's heads hurt!), if possible, keep projects in the same directory so that the relative path is always the same. Alternatively, full paths can be used, although this may introduce problems when the code is on different computers.

To verify that the path is right, look in the Project tab, expand the Packages folder, and the package should be listed with its source.

Path dependencies are useful for local development, but will not work when sharing projects with the wider world.

Understanding the package scope

Dart operates under a library scoping system. All items in a package are available throughout – there are no private or protected assets at this level. However, items inside the library can be kept private to external code.

To achieve the equivalent of a private field, a field of a class can be marked private if it has a preceding `_` (underscore) character. Any project referencing the library can only use the assets that the package has exposed.

For example, the `TextEditor` project can use the `Dialog` class that is public by default and access the `content` field, but not the `_visible` field.

Defining the base dialog box

The dialog core is defined in the `Dialog` class in the `dialog_base.dart` file. This will contain all the foundational functionality, such as the dialog frame and the **OK** and **CANCEL** buttons. More specialized dialog boxes will inherit (`extend`) this class and modify it for their purposes.

The package will contain common dialogs for the following:

- A simple alert dialog box
- An application about dialog box
- Confirmation dialog from the user

The alert dialog box

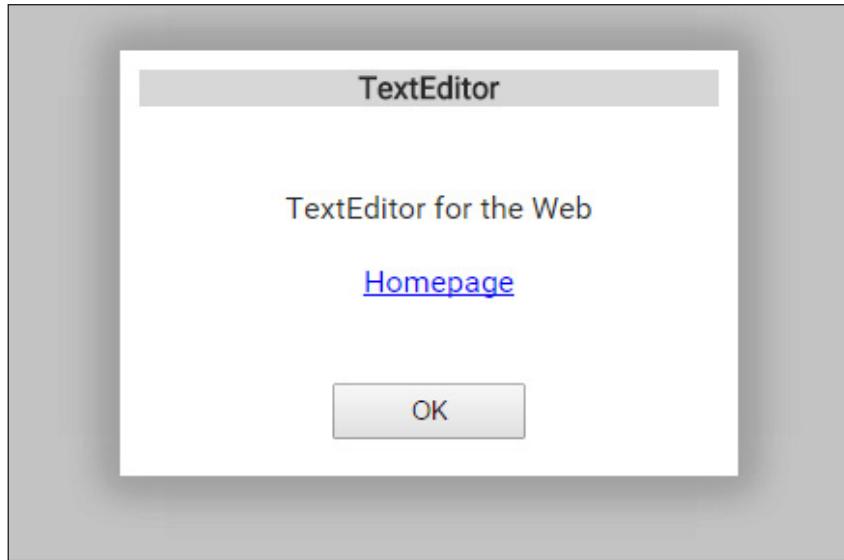
This is the simplest of all the dialogs. It will show a line of text and allow the user to press the **OK** button so that the dialog box is dismissed:

```
void alert(String dlgTitle, String prompt, int width, int height) {  
  Dialog dlg = new Dialog(dlgTitle, prompt, width, height, true,  
  false);  
  dlg.show();  
}
```

The dialog box does not return any value, and to make this as easy as possible for the user, it is a single function call. This will be used to display the result of the word-count feature.

The About dialog box

The standard `About` box for an application contains a small amount of information and a link to a website. This dialog is visually identical to the `Alert` box apart from the addition of the hyperlink, and as this involves changing the dialog interface elements, this will be implemented in a new class called `AboutDialog`:



This is created from the `showAbout` method of the `TextEditor` object:

```
void showAbout(MouseEvent event) {  
    AboutDialog textEditorAbout = new AboutDialog(AppTitle,  
        "TextEditor for the Web", "http://www.packtpub.com", "Homepage",  
        300, 200);  
    textEditorAbout.show();  
}
```

The `AboutDialog` box inherits from the `Dialog` class and calls the constructor of the `Dialog` class via `super`, which calls the constructor to form the foundation of the object:

```
    AboutDialog(String titleText, String bodyText, this.linkLabel, this.  
        linkText, int width, int height)  
    : super(titleText, bodyText, width, height, true, false) {
```

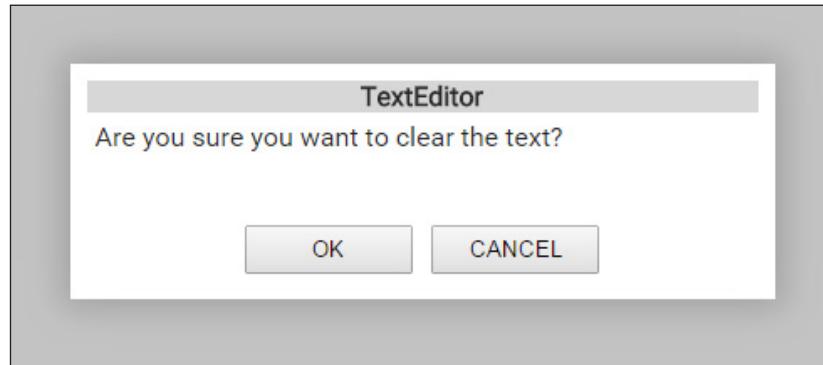
The elements for the box are created and added to the nodes member of the content DivElement:

```
content..nodes.insert(0, new BRElement())
..nodes.insert(0, new BRElement())
..append(new BRElement())
..append(new BRElement())
..append(link)
..style.textAlign = "center";
```

The cascade operator keeps the code compact and focused on setting up the different components of the dialog.

Using the confirmation dialog box

This dialog presents OK and CANCEL options, and calls a specified function if the former option is clicked. This is used in the `TextEditor` class `clearEditor` method:



This is implemented by using the base `Dialog` class directly:

```
void confirm(String dlgTitle, String prompt, int w, int h, Function
action) {
    Dialog dlg = new Dialog(dlgTitle, prompt, w, h);
    dlg.show(action);
}
```

The application can configure and display this dialog with a single line:

```
confirm(AppTitle, "Are you sure you want to clear the text?",  
 400, 120,  
  performClear);
```

A function is passed to the `show` method that is performed if the user presses the **OK** button.

Counting words using a list

Word count is an important feature for students, technical writers, and competition entrants. Our customers see this as a required feature, too.

In the event handler, we want to calculate the word count and throw the dialog on screen for the user to see. This is a simple, short-running task, so the implementation can exist directly in the event handler. As the output is simple, the alert dialog can be used.

To determine the word count, Dart's `List` data structure and `String` classes can be used. The first step is to remove any punctuation characters that may affect our count. This is declared in `custom_dialogs.dart` as an unmodifiable `const` string as it can be used for other word-based features:

```
const String punctuation = ",.-!\"";
```

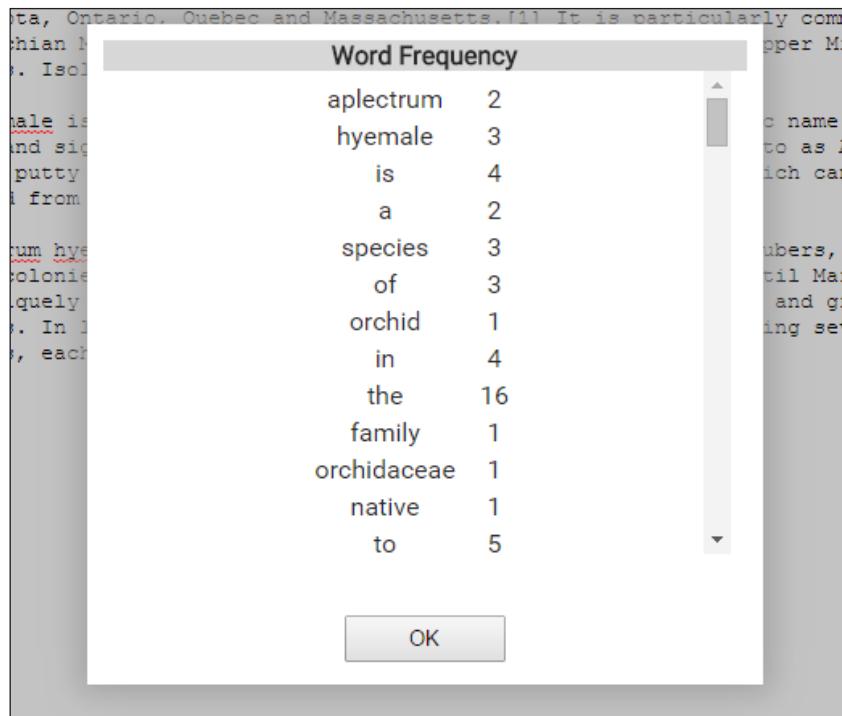
This is used in the `showWordCount` method of the `TextEditor` class.

Once the punctuation is removed, the string can be split into a `List`. Dart supports generics, so the types of general data structure can be optionally specified, in this case with `<String>`. The list of words is then filtered to clean out any non-word entries remaining.

Once we have obtained the list of words, the final word count can be obtained by using the `length` property of the list.

The Word Frequency feature

Our customers would like a feature to help with writing quality. Overuse of particular words can make a text repetitive and hard to read. To determine the frequency of word usage, Dart's Map (sometimes called an associative array or dictionary), data structure class, and String features can be used:



Depending on the number of different words used, the content areas may be allowed to scroll, as the dialog cannot grow to the size of any data.

Generics are used again to specify the types for the key and value; in this case, `String` for the key and `int` for the value:

```
Map<String, int> wordFreqCounts = {};
String out = "";

for (var character in punctuation.split(''))
  text = text.replaceAll(character, " ");
text = text.toLowerCase();
```

```
List<String> words = text.split(" ") ;  
words  
..removeWhere((word) => word == " ")  
..removeWhere((word) => word.length == 0)  
.forEach((word) {  
    if (wordFreqCounts.containsKey(word))  
        wordFreqCounts[word] = wordFreqCounts[word] + 1;  
    else wordFreqCounts[word] = 1;  
});  
  
wordFreqCounts  
.forEach((k, v) => out += ("<tr><td>$k</td><td>$v</td></tr>"));
```

The processing of the text is similar to the word count, where a list of individual words is built up. Once the list is filtered, `forEach` is used to iterate through the list and count the occurrences of each word. The `forEach` method is used again to build the output HTML, with the map's key and value parameters being passed to the function.

Understanding the typing of Dart code

The types used so far for the `Map` and `List` help the design-time tools, such as the Dart Analyzer and WebStorm, which allows us to write code by providing the code completion and validating assignments.

When run in Dartium with the checked mode enabled (the default setting), types are checked at runtime. In the final environment, most likely a web browser such as Firefox or Internet Explorer, the typing information is not used and does not affect the execution speed of the application.

Dart coding recommendations advise using types on all public-facing code. For example, a class in a package can use private `var` variables, but methods require and return typed data. This aids developers calling the code in understanding what is required, and helps create better documentation.

The file download feature

It can be very useful to have an actual file, so we will add a feature to let the user download and save the contents of the editor to their local device. This is possible with the custom data attributes feature in HTML5, which supports text as one of the formats:

```
void downloadFileToClient(String filename, String text) {  
    AnchorElement tempLink = document.createElement('a');  
    tempLink  
        ..attributes['href'] =  
            'data:text/plain;charset=utf-8,' + Uri.encodeComponent(text)  
        ..attributes['download'] = filename  
        ..click();  
}
```

The process is straightforward: an anchor element is created and its target is set as the encoded text data. The click event is then fired and the download proceeds as if it were being downloaded from a web server.

The clock feature

Everyone has deadlines, so keeping an eye on the clock is important. The `dart:async` package can help with this task by using a `Timer` class to trigger and update. This can be carried out in one line in `main.dart` file of the `TextEditor` project:

```
new Timer.periodic(newTimer.periodic(new Duration(seconds: 1),  
    (timer) => querySelector("#clock").text =  
        (new DateFormat('HH:mm'))  
            .format(new DateTime.now())  
    );
```

The `Timer` constructor being called here (`Timer.periodic`) is a specially named constructor; in this case, the `periodic` version is being used. There are numerous variations of `Timer` objects. Named constructors allow the classes to have instances easily created for a particular purpose as they do not require additional initialization configuration before they are ready to use.

A periodic `Timer` will fire an event every duration period; in this case, one second. The remainder of the line declares the function that is called. Dart has a shorthand of `=>` for functions that are a single line (which are named arrow functions), thus avoiding the use of curly brackets. The `querySelector` function is used to get a reference to the element on the web page with an ID of `clock` where the time will be displayed.

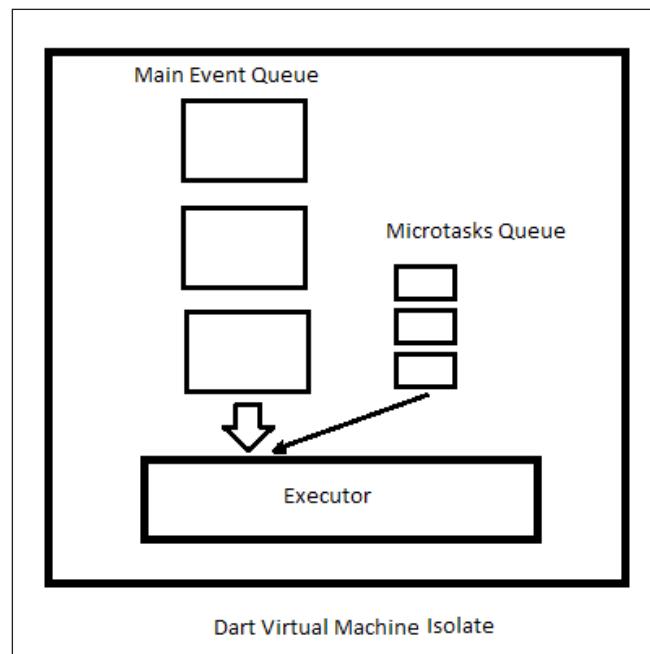
The current time is obtained and formatted into hours and minutes for display. As the clock will run for the lifetime of the application and does not require any interaction, no variable is required to store a reference to the object.

That was a busy line of code, and very powerful, but we are not finished examining it yet! There has been no extra thread or process created. This is not a user-initiated event. We will take a look at the what and when of the Dart task queue.

Executing Dart code

The internal architecture of the Dart VM has a single-threaded execution model that is event-driven. In Dart terminology, this is called an isolate. The execution of a Dart program places each event (for example, a function call) into one of two queues. The VM then takes the next event and executes it.

The first queue is the main event queue, and the second is called the microtask queue. This second queue is a higher priority, but is reserved, as the name implies, for short, small tasks, such as creating a new event or setting a value:



This execution detail is hidden from the Dart developer for the most part. However, it is important to understand that execution of Dart code is highly asynchronous. The aim of this is to allow applications to continue to be responsive, while still performing longer tasks such as calculations or accessing remote resources.

The advantage of such architecture is the complete removal of the need for any locking or synchronization during program execution. Other virtual machines have to do this, and this greatly affects complexity and performance. For example, Python's **GIL (Global Interpreter Lock)** has been debated for years, with many alternatives suggested and experimentally implemented.

Multi-processing the VM

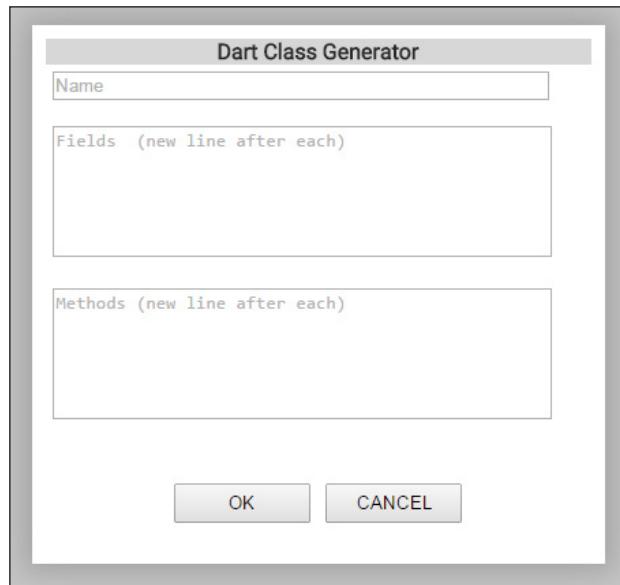
The design of the Dart VM allows multiple isolates to run simultaneously, which can be running across different threads, CPU cores, and, in the web browser, web workers. They do not share any resources such as memory, and their sole form of communication is through a messaging system.

The class designer

The customers want a feature that will appeal to programmers. They would like a way for the user to quickly sketch out ideas for a class, including details such as the class name, fields, and methods.

Building a more complicated dialog

To handle this feature, a custom dialog will be required – `GenClassDialog` in `custom_dialogs.dart`. This will inherit (`extend`) the `Dialog` class, and handle the **OK** button click event handler itself. Unlike the other dialogs so far in the application, this dialog must return a piece of data back to the text editor:



The input controls are one `TextInputElement` and two `TextAreaElements`. The `placeholder` property, available in modern web browsers, will be used to guide the user on what to enter, and, as a bonus, they are convenient space savers, as we do need to add extra elements to label the inputs:

```
String className = name.value;
String fieldsSrc = "";
String methodsSrc = "";

List<String> classFields = fields.value.split("\n");
List<String> classMethods = methods.value.split("\n");

classFields.forEach((field) => fieldsSrc += "    var $field;\n");
classMethods.forEach((method) => methodsSrc += "    $method(){};\n");
```

The text is retrieved from the controls and split by line into lists. The method `forEach` is then used to traverse the list and build the source code.

Constructing the class

The template of a simple class in the `GenClassDialog` `makeClass` method has placeholders for the generated source code of the fields and methods. This class code template is declared as a multi-line string using three double-quote characters. Dart supports string interpolation, which helps when putting values of variables into strings:

```
result = """ // The $className Class.
class $className {

    $fieldsSrc

    $className() {}

    $methodsSrc
}

""";
```

Variables can be embedded in strings prefixed by a `$` sign. These are then replaced with the actual value of the variable when the string is used.

It is also possible to insert more complicated code, such as an `objects` property, "The word is `${text.length}` characters long". Note the use of curly brackets around the expression being evaluated.

Understanding the flow of events

The dialog for creating the class source code is created and handled by the `TextEditor` class located in the `editor.dart` source file.

The sequence of events to display the dialog display is as follows:

1. The `showClassGen` event handler constructs the class and puts it on screen, and the function completes, leaving the dialog visible to the user.
2. The user fills in the details and hits the **OK** button. This event triggers the `OK pressed` method on the `ClassGenDialog`.
3. The text is extracted from the fields and the source code in the `ClassGenDialog makeClass` method. Then, the `resultHandler` is called.
4. The `resultHandler` is the `createClassCode` method on the editor that sets the `TextAreaElement` as the constructed source code.

Launching the application

To launch the application and try the text editor application for yourself, select `index.html` in the **Project** tab, right-click to bring up the context menu, and click on **Run 'index.html'**. This Run command is also available in the **Run** menu on the main WebStorm menu bar.

The command-line app for source code statistics

Our customers like the software development angle of the text editor and has asked for some source code statistics such as **SLOC (source lines of code)** to be made accessible. This feature may be of use outside the web page, such as in a continuous integration environment, so we will construct our first command-line application.

We will create the project in WebStorm from scratch, but a completed version is also supplied in the sample code in a folder called `dart_stats`.

The command-line project structure

Open WebStorm, click on **Create New Project**, and select **Console application** from the **Project Templates** list. Choose a location for your project and call it `dart_stats`.

The structure of the application has a `bin` folder with `main.dart`, which is the entry point for the application. This is where the `main` function is located. The project template contains several other items. For this tool, the focus will be on the `main` function.



Dart is cross-platform, and the majority of filesystems used by the OS platforms are case-sensitive. It is a highly-recommended convention that all filenames are lowercase.



Add the source code file `sourcescan.dart` to the project. This contains the code for scanning the text and tallying up line counts for classes, code, comments, imports, and whitespace.

Processing the source code

The `SourceCodeScanner` class contains five integer fields to store the counts and a single method to perform the analysis:

```
void scan: (List<String> lines) {
    totalLines = lines.length;
    lines.forEach((line) {
        line = line.trim();
        if (line.startsWith("class")) classes++;
        else if (line.startsWith("import")) imports++;
        else if (line.startsWith("//")) comments++;
        else if (line.length == 0) whitespace++;
    });
}
```

The list of lines is iterated over and the `trim` method is used to remove extra whitespace from the lines to aid in matching the keywords that trigger and increment the count. The `startsWith` method makes sure that the keyword is not appearing mid-line in another context.

File handling with the `dart:io` package

The application will be passed a single command-line argument, which is the full file path to a Dart source code file. The command-line arguments can be read from the parameter passed to the main function in `main.dart`.

One aspect that we will not have to deal with in the browser is loading the source code from a text file. A package that is not available to Dart in the web browser, due to security, is the `io` package, which contains direct file handling functionality:

```
import 'sourcescan.dart';
import 'dart:io';

main(List<String> arguments) {
  print(arguments[0]);
  SourceCodeScanner codeScan = new SourceCodeScanner();
  File myFile = new File(arguments[0]);
  myFile.readAsLines().then((List<String> Lines) {
    codeScan.scan(Lines);
    print("${codeScan.totalLines}");
    print("${codeScan.classes}");
    print("${codeScan.comments}");
    print("${codeScan.imports}");
    print("${codeScan.whitespace}");
  });
}
```

Once the text file is in a string, it can be processed in the same manner as in the web text editor.

The program can be run from the command-line, as shown here on Linux:

```
$ dart bin/main.dart lib/dartstats.dart
lib/dartstats.dart
9
0
3
0
2
```

The program can examine its own source code!



For more advanced handling of command-line arguments, see the very powerful `args` package on pub at <https://pub.dartlang.org/packages/args>.

`readAsLines` is an asynchronous function that immediately returns a `Future` object. A function is passed into the future object's `then` method. The timing of the execution of this function is unknown to the developer. The flow of the program continues straight away. When the file read operation is run and has completed, the function passed to the `then` method is executed when the Dart VM schedules it.

To see this in action, move the `print` statements outside of the `then` function, as shown, and run the program again:

```
...
myFile.readAsLines().then((List<String> Lines) {
    codeScan.scan(Lines);
});
print("${codeScan.totalLines}");
print("${codeScan.classes}");
print("${codeScan.comments}");
print("${codeScan.imports}");
print("${codeScan.whitespace}");
...

```

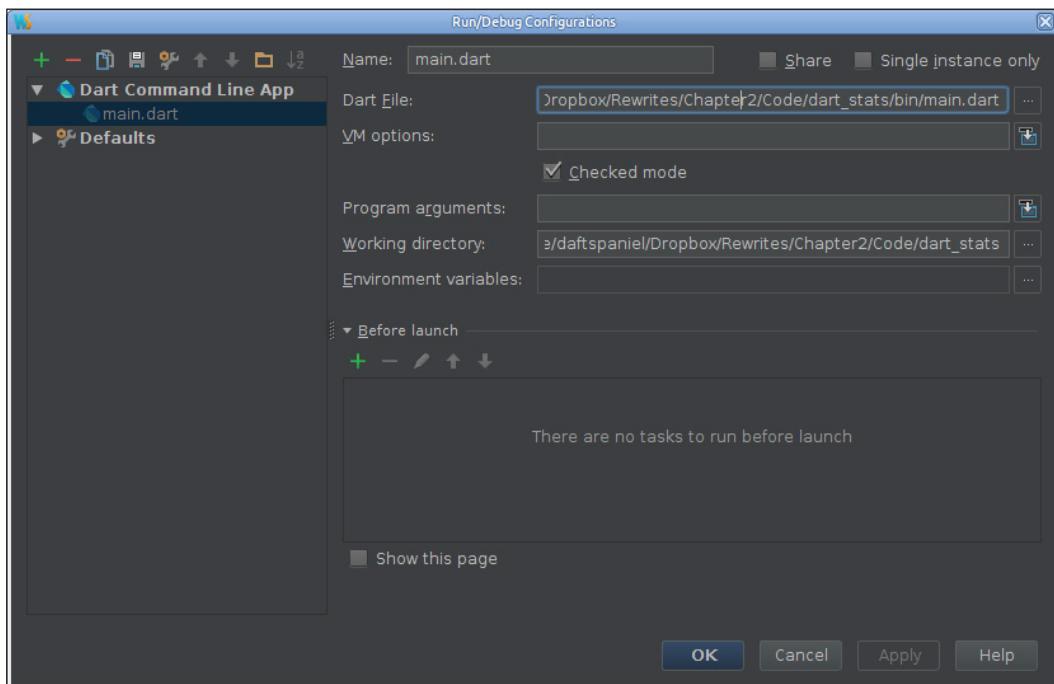
The probable output is all zeros as the code lines have not been scanned yet; this is because the `print` functions are being called first.

It is also possible that the program will run just as before. The key point is that the overall program execution keeps moving forward even when the code is asked to perform a potentially long-running task, allowing other tasks to continue and keeping the application responsive.

Debugging the command-line program

If you launch the program as it is in WebStorm by using `Run`, then it will open up the debugger with a `RangeError` exception. This is because the program is assuming that a command-line argument has been passed and the program has been run without any input.

In WebStorm, launches of programs into debugging can be configured, including a feature to set command-line arguments. In this case, a full file path to a Dart source file should be put in the **Program arguments** field. Note that, by default, no command-line arguments are passed. It is also possible to set multiple launches with different sets of command-line options:



To configure launches, select the **Edit Configurations** option from the **Run** menu. Select `main.dart` from the list and enter the full file path of any Dart source code file, such as one for the sample code for this book or from the Dart SDK.

Once the program is run, the results are printed to standard output using the `print` function.

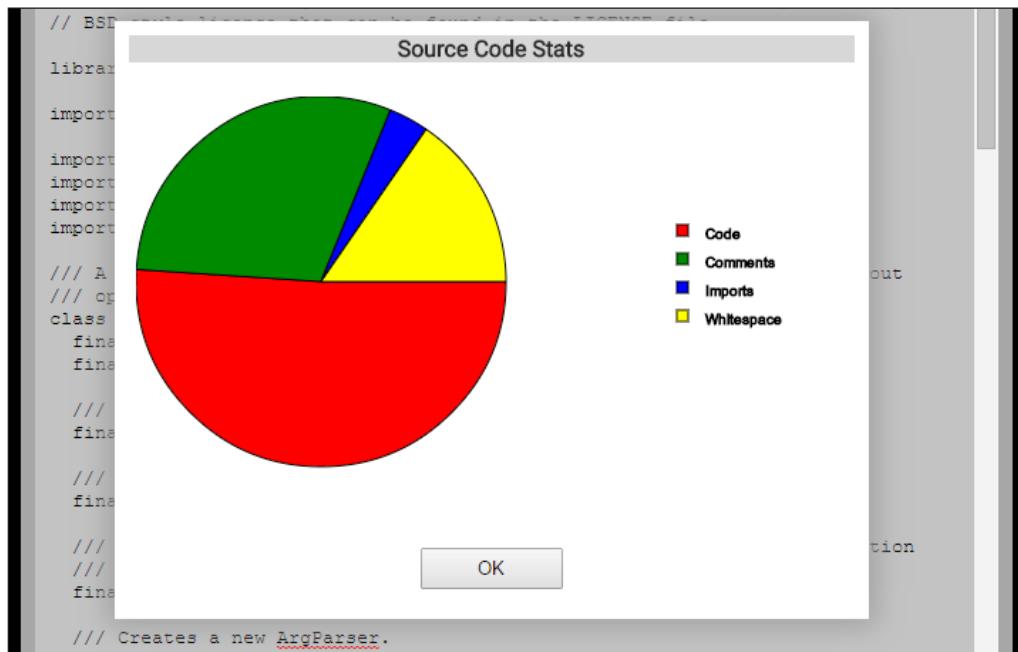
Integrating the statistics

The class `SourceStats` can be added to the text editor project by adding the `sourcescan.dart` file to the `bin` folder and referencing it as a straightforward import. The scan is to be performed on the contents of the editor in the `scanCode` method of the `CodeStatDialog`, and the data is to be extracted for drawing on the pie chart.

HTML5 and the canvas

Given that the `Canvas` HTML element has been around for approximately 10 years since Apple introduced it into their version of Webkit, it seems odd that it is still considered a new feature! Widespread support has been seen in the major and minor browsers for several years now.

The `canvas` element provides a high-performance 2D raster-based graphics bitmap that can be scripted dynamically. In contrast to `SVG`, there are no scene graphs or selectable objects, just a graphic image. It provides an easy-to-use and powerful way to display dynamic images on the client side for applications such as graphs, animations, and games:



Drawing the pie chart

The four pieces of data collected are to be represented as segments in a pie chart. In the `CodeStatsDialog` constructor, the canvas HTML element is selected in the same manner as any other page element. A **2D** context is requested, which returns an object we can call methods on to draw on the Canvas:

```
CanvasElement graphs = new CanvasElement();
graphs.width = 500;
graphs.height = 270;
c2d = graphs.getContext("2d");
contentDiv..append(new BRElement())..append(graphs);
```

The stroke is the pen to be used and the fill is analogous to the paint. Each pie segment is drawn in a black outline (`stroke`) and painted in with the `fill`:

```
for (int i = 0; i < 4; i++) {
    c2d
        ..fillStyle = colors[i]
        ..strokeStyle = "black"
        ..beginPath()
        ..moveTo(radius, radius)
        ..arc(radius, radius, radius, lastpos,
              (lastpos + (PI * 2.0 * (data[i] / totalLines))), false)
        ..lineTo(radius, radius)
        ..fill()
        ..stroke()
        ..closePath();
    lastpos += PI * 2.0 * (data[i] / totalLines);
    print(lastpos);
```

The `arc` method uses radians, so the constant `PI` is used in the calculations:

```
c2d
..beginPath()
..strokeStyle = "black"
..fillStyle = colors[i]
..fillRect(380, 90 + 20 * i, 8, 8)
..strokeRect(380, 90 + 20 * i, 8, 8)
..strokeText(labels[i], 400, 100 + 20 * i)
..stroke()
..closePath();
```

The labels are drawn for each of the four counts by using a square `Rect` to show the color, and the text is drawn alongside it.



The canvas is a versatile feature, but it can be frustrating when nothing appears on it. This often occurs when `beginPath` is used and the path has not been closed with `closePath`.



Building web interfaces with Dart

The `dart:html` package is not the only option for building web interfaces with Dart. The main contenders are Polymer and Angular:

- Polymer (<https://www.polymer-project.org/>) is a JavaScript framework for creating reusable web components (using the Web Component standard) that are encapsulated and inter-operable. They take the form of custom HTML elements. For example, the dialogs in this chapter could be made into web components. To use them in a page, we would use the tags `<DialogConfirm>` and `<DialogWordCount>`, which would avoid a set of nested `div` elements. `Polymer.dart` is the Dart version of the framework and is available from `pub`.
- Angular (<https://angularjs.org/>) is a framework for dynamic data views and is a giant in the JavaScript world. Again, there is a Dart version, called `Angular.dart`. The forthcoming Angular 2 will be written in Typescript and will support both JavaScript and Dart.

Whichever framework or package is chosen, it is likely to use `dart:html` classes at some point, so it is worthwhile to be familiar with them. Not every developer or application needs a framework, so be pragmatic!

Compiling to JavaScript

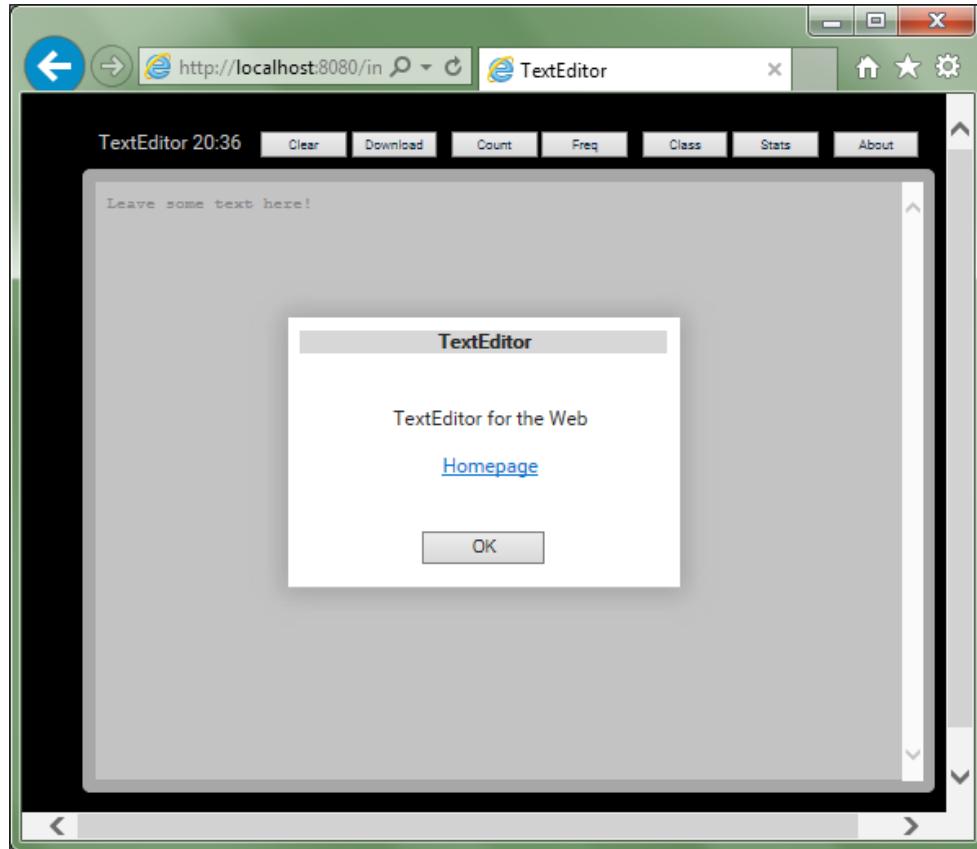
It is great that we have our enhanced text editor application running in Dartium, but that is not what most web users have installed. What if I want to use it in my daily-driver web browser; for example, Firefox or Internet Explorer?

In order to run on the entire modern Web, Dart code can be compiled via a tool called `dart2js` (written in Dart) to high quality and performance JavaScript. This can be run as a `pub` command:

```
pub build --mode=release
```

We first encountered this tool when we toured the SDK's command-line tools in the previous chapter. For easy use during development, it is accessible from WebStorm.

In WebStorm's **Project** tab, right-click on the `pubspec.yaml` file, and select **Pub: Build** The **mode** option is then presented for **Release** or **Debug**—for production, choose **Release**. This creates a folder called `build` with the JavaScript version of the program. This can be served by any web server and used from any modern browser:



It may be surprising, if text was entered in the Dartium version, that the editor is blank when run from a regular browser. One downside of using local storage is that each web browser has its own independent storage and the user will have to re-enter data if they switch browsers, even if they are on the same computer.

Minification of JavaScript output

The output files from the dart2js compiler can get quite large if certain advanced Dart features are at play, such as mirrors, which are used for reflection (sometimes called introspection).

Fortunately, the default setting of the Dart editor is to produce a minified output. There is always the overhead of the Dart runtime in the JavaScript that is disproportionately large for small applications. As Dart has all the source code ahead of runtime and knowledge of the entire application, it can compile all the code (including packages), determine which are not required, and then, within the packages that are used, remove any parts that are not used.

This sophisticated technique is called tree shaking. Any unused code is shaken off and not included in the final application that is delivered to the end user, keeping it quick to load and execute. Compare this with JavaScript, where if a library is referenced, then the entire library is sent to the client for interpretation.

Summary

The text editor application now has many more useful features, and can be used for a variety of purposes. You are probably thinking of more customized little features you could add to it to help with day-to-day computing. Our imaginary customer has left at this point, so you are in charge now! I am not sure that the buttons are ones I would want to lick, but I would definitely click them for useful functionality.

We have developed a reusable library for creating dialogs that fit in with our application. Dart's data structures, generics, and string handling, are covered from a variety of angles.

We have also achieved an effective presentation of the data through visualizations on the HTML5 canvas, used asynchronous features, and looked deep into the internals of the Dart virtual machine. Finally, we took the application out of the controlled development environment and prepared it for use in any browser by using dart2js.

Text is very powerful, but a little static and hard to read from across a large meeting room. The next step in our Dart journey is to look at animation, presentation, and making some noise.

3

Slideshow Presentations

It usually takes me more than three weeks to prepare a good impromptu speech.

- Mark Twain

Presentations make some people shudder with fear, yet they are an undeniably useful tool for information sharing when used properly. The content has to be great, and some visual flourish can make it stand out from the crowd.

Too many slides can make the most receptive audience yawn, so having the presenter focus on the content and automatically take care of the visuals (saving the creator from fiddling with different animations and font sizes) can help improve presentations. Compelling content still requires the human touch.

Building a presentation application

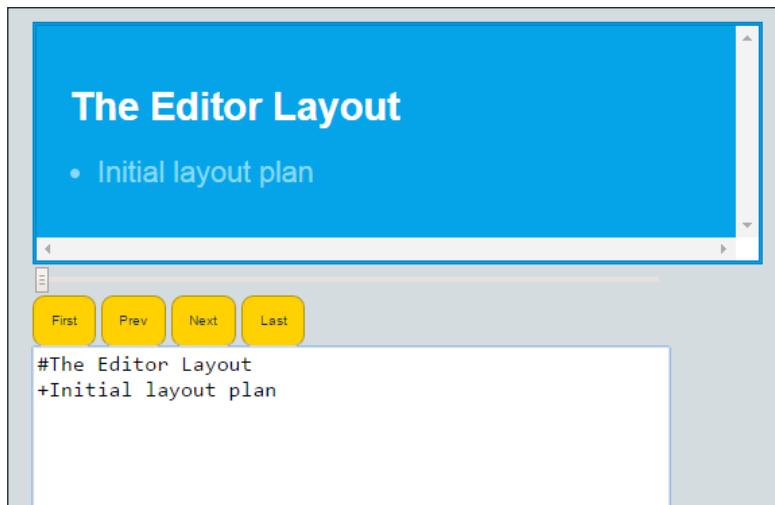
Web browsers are already a type of multimedia presentation application, so it is feasible to write a quality presentation program as we explore more of the Dart language. Hopefully, it will help us pitch another Dart application to our next customer.

Building on our first application, we will use a text-based editor for creating the presentation content. I was very surprised at how much faster a text-based editor is for producing a presentation, and also more enjoyable. I hope that you also experience such a productivity boost!

Laying out the application

The application will have two modes, editing and presentation. In the editing mode, the screen will be split into two panes. The top pane will display the slides and the lower will contain the editor and other interface elements.

This chapter will focus on the core creation side of the presentation, and the following chapter will focus on the presentation mode and advancing the interface. The application will be a single Dart project.



Defining the presentation format

The presentations will be written in a tiny subset of the Markdown format, which is a powerful yet simple to read text file-based format (much easier to read, type, and understand than HTML).

[ In 2004, John Gruber and the late Aaron Swartz created the Markdown language with the goal of enabling people to write using an easy-to-read, easy-to-write plain text format. It is used on major websites such as GitHub.com and StackOverflow.com. Being plain text, Markdown files can be kept and compared in version control. For more details and background on Markdown, see <https://en.wikipedia.org/wiki/Markdown>]

A simple titled slide with bullet points would be defined as:

```
#Dart Language
+Created By Google
+Modern language with a familiar syntax
+Structured Web Applications
+It is Awesomely productive!
```

I am positive you only had to read that once! This will translate into the following HTML:

```
<h1>Dart Language</h1>
<li>Created By Google</li>
<li>Modern language with a familiar syntax</li>
<li>Structured Web Applications</li>
<li>It is Awesomely productive!</li>
```

Markdown is very easy and fast to parse, which probably explains its growing popularity on the Web. It can be transformed into many other formats.

Parsing the presentation

The content of the `TextAreaHtml` element is split into a list of individual lines, and processed in a similar manner to some of the features in the text editor application using `forEach` to iterate over the list. Any lines that are blank once any whitespace has been removed via the `trim` method are ignored:

```
#A New Slide Title
+The first bullet point
+The second bullet point

#The Second Slide Title
+More bullet points
!http://localhost/img/logo.png
#Final Slide
+Any questions?
```

For each line starting with a `#` symbol, a new slide object is created.

For each line starting with a `+` symbol, they are added to this slide's bullet point list.

For each line discovered using a `!` symbol, the slide's image is set (a limit of one per slide).

This continues until the end of the presentation source is reached.

A sample presentation

To get a new user going quickly, there will be an example presentation that can be used as a demonstration and to test the various areas of the application. I chose the last topic that came up around the family dinner table—the coconut as shown in the following code snippet:

```
#Coconut
+Member of Arecaceae family.
+A drupe - not a nut.
+Part of daily diets.
#Tree
+Fibrous root system.
+Mostly surface level.
+A few deep roots for stability.
#Yield
+75 fruits on fertile land
+30 typically
+Fibre has traditional uses
#Finally
!coconut.png
#Any Questions?
```

Presenter project structures

The project is a standard Dart web application with `index.html` as the entry point. The application is kicked off by `main.dart`, which is linked to in `index.html`, and the application functionality is stored in the `lib` folder.

Source File	Description
<code>sampleshows.dart</code>	The text for the slideshow application
<code>lifecycle mixin.dart</code>	The class for the mixin
<code>slideshow.dart</code>	Data structures for storing the presentation
<code>slideshowapp.dart</code>	The application object

Launching the application

The main function has a very short implementation:

```
void main() {
    new SlideShowApp();
}
```

Note that the new class instance does not need to be stored in a variable and that the object does not disappear after that line is executed. As we will see later, the object will attach itself to events and streams, keeping the object alive for the lifetime that the page is loaded.

Building bullet point slides

The presentation is built up using two classes – `slide` and `slideShow`. The `slide` object creates the `DivElement` used to display the content, and the `slideShow` contains a list of `slide` objects.

The `slideShow` object is updated as the text source is updated. It also keeps track of which slide is currently being displayed in the preview pane.



Once the number of Dart files grows in a project, the DartAnalyzer will recommend naming the library. It is good habit to name every `.dart` file in a regular project with its own library name.

The `slideshow.dart` file has the keyword `library` and a name next to it. In Dart, every file is a library, whether it is explicitly declared or not.



If you are looking at Dart code online, you may stumble across projects with imports that look a bit strange, for example:

```
#import ("dart:html");
```

This is the old syntax for Dart's import mechanism. If you see this, it is a sign that other aspects of the code may be out of date, too.

If you are writing an application in a single project, source files can be arranged in a folder structure appropriate for the project, though keeping the relative's paths manageable is advised. Creating too many folders probably means that it is time to create a package!

Accessing private fields

In Dart, as discussed when we covered packages, the privacy is at the library level, but it is still possible to have private fields in a class even though Dart does not have the keywords public, protected, and private. A simple return of a private field's value can be performed with a one-line function:

```
String getFirstName() => _name;
```

To retrieve this value, a function call is required, for example, `Person.getFirstName()`; however, it may be preferable to have a property syntax such as `Person.firstName`. Having private fields and retaining the property syntax in this manner is possible using the `get` and `set` keywords.

Using true getters and setters

The syntax of Dart also supports `get` and `set` via keywords:

```
int get score => score + bonus;  
set score(int increase) => score += increase * level;
```

Using either `get`/`set` or simple fields is down to preference. It is perfectly possible to start with simple fields and scale up to getters and setters if more validation or processing is required.

The advantage of the `get` and `set` keywords in a library is that the intended interface for consumers of the package is very clear. Further, it clarifies which methods may change the state of the object and which merely report current values.

Mixin' it up

In object-oriented languages, it is useful to build on one class to create a more specialized related class. For example, in the text editor, the base dialog class was extended to create an alert and confirm pop ups. What if we want to share some functionality but do not want inheritance occurring between the classes?

Aggregation can solve this problem to some extent:

```
class A {  
    classB usefulObject;  
}
```

The downside is that this requires a longer reference to use:

```
new A().usefulObject.handyMethod();
```

This problem has been solved in Dart (and other languages) by having a `mixin` class do this job, allowing the sharing of functionality without forced inheritance or clunky aggregation.

In Dart, a `mixin` must meet the following requirements:

1. No constructors can be in the class declaration.
2. The base class of the `mixin` must be the `Object`.
3. No calls to a super class are made.

`mixins` are really just classes that are malleable enough to fit into the class hierarchy at any point. A use case for a `mixin` may be serialization fields and methods that could be required on several classes in an application and that are not part of any inheritance chain.

```
abstract class Serialisation {
    void save() {
        //Implementation here.
    }
    void load(String filename) {
        //Implementation here.
    }
}
```

The `with` keyword is used to declare that a class is using a `mixin`:

```
class ImageRecord extends Record with Serialisation
```

If the class does not have an explicit base class, it is required to specify an `Object`:

```
class StorageReports extends Object with Serialisation
```

In Dart, everything is an object, even basic types such as `num` are objects and not primitive types. The classes `int` and `double` are subtypes of `num`. This is important to know as other languages have different behaviors. Let's consider a real example of this:

```
main() {
    int i;
    print("$i");
}
```

In a language such as Java, the expected output would be `0`; however, the output in Dart is `null`. If a value is expected from a variable, it is always good practice to initialize it!



For the classes `Slide` and `SlideShow`, we will use a mixin from the source file `lifecycleMixin.dart` to record a creation and an editing timestamp:

```
abstract class LifecycleTracker {  
  DateTime _created;  
  DateTime _edited;  
  recordCreateTimestamp() => _created = new DateTime.now();  
  updateEditTimestamp() => _edited = new DateTime.now();  
  DateTime get created => _created;  
  DateTime get lastEdited => _edited;  
}
```

To use the mixin, the `recordCreateTimestamp` method can be called from the constructor and the `updateEditTimestamp` from the main edit method. For slides, it makes sense just to record the creation. For the `SlideShow` class, both the creation and update will be tracked.

Defining the core classes

The `SlideShow` class is largely a container object for a list of `Slide` objects and uses the mixin `LifecycleTracker`:

```
class SlideShow extends Object with LifecycleTracker {  
  List<Slide> _slides;  
  List<Slide> get slides => _slides;  
  ...
```

The `Slide` class stores the string for the title and a list of strings for the bullet points. The URL for any image is also stored as a string:

```
class Slide extends Object with LifecycleTracker {  
  String titleText = "";  
  List<String> bulletPoints;  
  String imageUrl = "";  
  ...
```

A simple constructor takes the `titleText` as a parameter and initializes the `bulletPoints` list.



If you want to focus on just the code when in WebStorm, double-click on the filename title of the tab to expand the source code to the entire window. Double-click again to return to the original layout.

For even more focus on the code, go to the **View** menu and click on **Enter Distraction Free Mode**.

Transforming data into HTML

To add the `slide` object instance into an HTML document, the strings need to be converted into instances of HTML elements to be added to the **DOM (Document Object Model)**. The `getSlideContents()` method constructs and returns the entire slide as a single object:

```
DivElement getSlideContents() {
    DivElement slide = new DivElement();
    DivElement title = new DivElement();
    DivElement bullets = new DivElement();

    title.appendHtml("<h1>$titleText</h1>");
    slide.append(title);

    if (imageUrl.length > 0) {
        slide.appendHtml("<img src=\"$imageUrl\" /><br/>");
    }

    bulletPoints.forEach(bp) {
        if (bp.trim().length > 0) {
            bullets.appendHtml("<li>$bp</li>");
        }
    });
}

slide.append(bullets);

return slide;
}
```

The `Div` elements are constructed as objects (instances of `DivElement`), while the content is added as literal HTML statements. The method `appendHtml` is used for this particular task as it renders HTML tags in the text. The regular method `appendText` puts the entire literal text string (including plain unformatted text of the HTML tags) into the element.

So, what exactly is the difference? The method `appendHtml` evaluates the supplied HTML and adds the resultant object node to the nodes of the parent element, which is rendered in the browser as usual. The method `appendText` is useful, for example, to prevent user-supplied content affecting the format of the page and preventing malicious code being injected into a web page.

Editing the presentation

When the source is updated, the presentation is updated via the `onKeyUp` event. This was used in the text editor project to trigger a save to local storage.

This is carried out in the `build` method of the `SlideShow` class, and follows the pattern we discussed in parsing the presentation:

```
build(String src) {
    updateEditTimestamp();
    _slides = new List<Slide>();
    Slide nextSlide;

    src.split("\n").forEach((String line) {
        if (line.trim().length > 0) {

            // Title - also marks start of the next slide.
            if (line.startsWith("#")) {
                nextSlide = new Slide(line.substring(1));
                _slides.add(nextSlide);
            }
            if (nextSlide != null) {
                if (line.startsWith("+")) {
                    nextSlide.bulletPoints.add(line.substring(1));
                } else if (line.startsWith("!")) {
                    nextSlide.imageUrl = line.substring(1);
                }
            }
        }
    });
}
```

As an alternative to the `startsWith` method, the square bracket [] operator could be used for `line[0]` to retrieve the first character. The `startsWith` method can also take a regular expression or a string to match, as well as a starting index. Refer to the `dart:core` documentation for more information. For the purposes of parsing the presentation, the `startsWith` method is more readable.

Displaying the current slide

The slide is displayed via the `showSlide` method in `slideShowApp.dart`. To preview the current slide, the current index, which is stored in the field `currentSlideIndex`, is used to retrieve the desired slide object and the `Div` rendering method is called:

```
showSlide(int slideNumber) {
    if (currentSlideShow.slides.length == 0) return;

    slideScreen.style.visibility = "hidden";
    slideScreen
    ..nodes.clear()

    ..nodes.add(currentSlideShow.slides[slideNumber].getSlideContents
        ());
    rangeSlidePos.value = slideNumber.toString();
    slideScreen.style.visibility = "visible";
}
```

The `slideScreen` is a `DivElement` that is then updated off screen by setting the `visibility` style property to `hidden`. The existing content of the `DivElement` is emptied out by calling `nodes.clear()` and the slide content is added with `nodes.add`. The range slider position is set, and finally, the `DivElement` is set to `visible` again.

Navigating the presentation

A button set with the familiar first, previous, next, and last slide allows the user to jump around the preview of the presentation. This is carried out by having an index built into the list of slides and stored in the field `slide` in the `SlideShowApp` class.

Handling the button key presses

The navigation buttons require being set up in an identical pattern in the constructor of the `SlideShowApp` object. First, get an object reference using `id`, which is the `id` attribute of the element, and then attach a handler to the `click` event. Rather than repeat this code, a simple function can handle the process:

```
setButton(String id, Function clickHandler) {
    ButtonInputElement btn = querySelector(id);
    btn.onClick.listen(clickHandler);
}
```

Because `Function` is a type in Dart, functions can be passed around easily as a parameter. Let us take a look at the button that takes us to the first slide:

```
setButton("#btnFirst", startSlideShow);

void startSlideShow(MouseEvent event) {
    showFirstSlide();
}

void showFirstSlide() {
    showSlide(0);
}
```

The event handlers do not directly change the slide; these are carried out by other methods that may be triggered by other inputs such as the keyboard.

Using the Function type

The `SlideShowApp` constructor makes use of this feature:

```
Function qs = querySelector;
var controls = qs("#controls");
```

I find the `querySelector` method a little long to type (though it is descriptive of what it does). With `Function` being comprised of types, we can easily create a shorthand version.

The constructor spends much of its time selecting and assigning the HTML elements to member fields of the class. One of the advantages of this approach is that the DOM of the page is queried only once, and the reference is stored and reused. This is good for performance of the application as, once the application is running, querying the DOM may take much longer.

Staying within the bounds

Using the `min` and `max` functions from the `dart:math` package, the index can be kept in the range of the current list:

```
void showLastSlide() {
    currentSlideIndex = max(0, currentSlideShow.slides.length - 1);
    showSlide(currentSlideIndex);
}
void showNextSlide() {
```

```

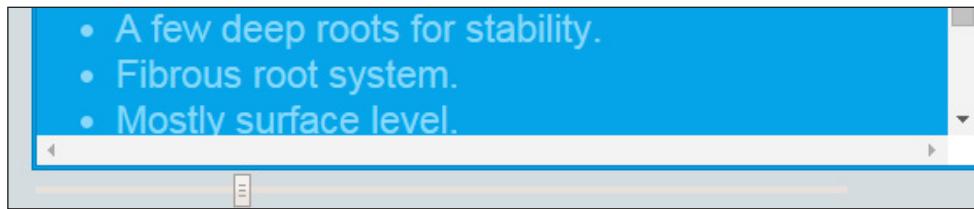
currentSlideIndex =
min(currentSlideShow.slides.length - 1, ++currentSlideIndex);
showSlide(currentSlideIndex);
}

```

These convenient functions can save a great deal of `if` and `else if` comparisons and help make the code a good degree more readable.

Using the slider control

The slider control is another new control in the HTML5 standard. This will allow the user to scroll though the slides in the presentation.



This control is a personal favorite of mine as it is so visual and can be used to give very interactive feedback to the user. It seemed to be a huge omission from the original form controls in the early generation of web browsers. Even with clear, widely accepted features, HTML specifications can take a long time to clear committees and make it into everyday browsers!

```
<input type="range" id="rngSlides" value="0" />
```

The control has an `onChange` event that is given a listener in the `SlideShowApp` constructor:

```
rangeSlidePos.onChange.listen(moveToSlide); rangeSlidePos.onChange
.listen(moveToSlide);
```

The control provides its data via a simple string value, which can be converted to an integer via the `int.parse` method to be used as an index in the presentation's slide list:

```

void moveToSlide(Event event) {
    currentSlideIndex = int.parse(rangeSlidePos.value);
    showSlide(currentSlideIndex);
}

```

The slider control must be kept in synchronization with any other change in its slide display, use of navigation, or change in number of slides. For example, the user may use the slider to reach the general area of the presentation, and then adjust with the Previous and Next buttons:

```
void updateRangeControl() {  
    rangeSlidepos  
    ..min = "0"  
    ..max = (currentSlideShow.slides.length - 1).toString();  
}
```

This method is called when the number of slides is changed, and as with working with most HTML elements, the values to be set need to be converted to strings.

Responding to keyboard events

Using the keyboard, particularly the arrow (cursor) keys, is a natural way to look through the slides in a presentation, even in the preview mode. This is carried out in the `SlideShowApp` constructor.



In Dart web applications, the `dart:html` package allows direct access to the `globalwindow` object from any class or function.

The `Textarea` used to input the presentation source will also respond to the arrow keys, so there will need to be a check to see if it is currently being used. The property `activeElement` on the document will give a reference to the control with focus. This reference can be compared to the `Textarea`, which is stored in the `presEditor` field, so a decision can be made on whether to act on the `keypress` or not.

Key	Event Code	Action
Left arrow	37	Go back a slide
Up arrow	38	Go to first slide
Right arrow	39	Go to next slide
Down arrow	40	Go to last slide

Keyboard events, like other events, can be listened to by using a stream event listener. The listener function is an anonymous function (the definition omits a name) that takes the `KeyboardEvent` as its only parameter:

```
window.onKeyUp.listen((KeyboardEvent e) {
  if (presEditor != document.activeElement) {
    if (e.keyCode == 39)
      showNextSlide();
    else if (e.keyCode == 37)
      showPrevSlide();
    else if (e.keyCode == 38)
      showFirstSlide();
    else if (e.keyCode == 40)
      showLastSlide();
  }
});
```

 It is a reasonable question to ask how to get the keyboard key codes required to write the switching code. One good tool is the W3C's **Key and Character Codes** page at <http://www.w3.org/2002/09/tests/keys.html>. Although this documentation is helpful with this question, it can often be faster to write the handler and print out the event that is passed in.

Showing the key help

Rather than testing the user's memory, there will be a handy reference to the keyboard shortcuts.



This is a simple `Div` element that is shown and then hidden when the key (remember to press *Shift*, too!) is pressed again by toggling the visibility style from visible to hidden.

Listening twice to event streams

The event system in Dart is implemented as a stream. One of the advantages of this is that an event can easily have more than one entity listening to the class.

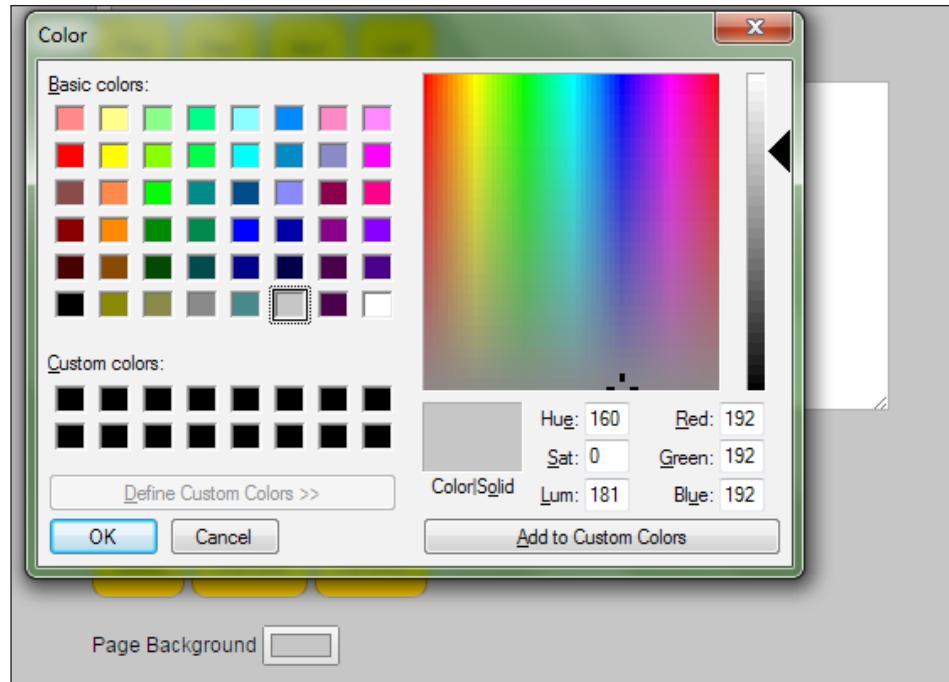
This is useful, for example, in a web application where some keyboard presses are valid in one context but not in another. The `listen` method is an add operation (accumulative) so that the key press for help can be implemented separately. This allows a modular approach, which helps reuse, as the handlers can be specialized and added as required:

```
window.onKeyUp.listen((KeyboardEvent e) {  
  print(e);  
  
  //Check the editor does not have focus.  
  if (presEditor != document.activeElement) {  
    DivElement helpBox = qs("#helpKeyboardShortcuts");  
    if (e.keyCode == 191) {  
      if (helpBox.style.visibility == "visible") {  
        helpBox.style.visibility = "hidden";  
      } else {  
        helpBox.style.visibility = "visible";  
      }  
    }  
  }  
});
```

In a game, for example, a common set of event handling may apply to the title and introduction screen, and the actual in-game screen can contain additional event handling as a superset. This can be implemented by adding and removing handlers to the relevant event stream.

Changing the colors

HTML5 provides browsers with a full-featured color picker (typically, browsers use the native OS's color chooser). This will be used to allow the user to set the background color of the editor application itself:



The color picker is added to the `index.html` page with the following HTML:

```
<input id="pckBackColor" type="color">
```

The implementation is straightforward as the color picker control provides:

```
InputElement cp = qs("#pckBackColor");
cp.onChange.listen(
(e) => document.body.style.backgroundColor = cp.value);
```

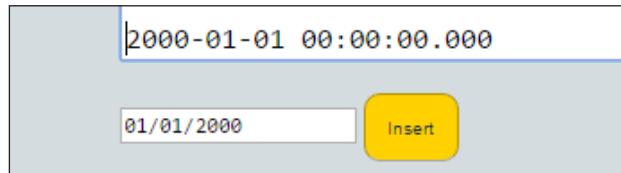
Because the event and property (`onChange` and `value`) are common to the input controls, the basic `InputElement` class can be used.

Adding a date

Most presentations are usually dated, or at least some of the jokes are! We will add a convenient button for the user to add a date to the presentation using the HTML5 input type `date`, which provides a graphical date picker:

```
<input type="date" id="selDate" value="2000-01-01"/>
```

The default value is set in the `index.html` page as follows:



The `valueAsDate` property of the `DateInputElement` class provides the `Date` object, which can be added to the text area:

```
void insertDate(Event event) {  
    DateInputElement datePicker = querySelector("#selDate");  
    if (datePicker.valueAsDate != null) presEditor.value =  
        presEditor.value +  
        datePicker.valueAsDate.toLocal().toString();  
}
```

In this case, the `toLocal` method is used to obtain a string formatted to the month, day, and year format.

Timing the presentation

The presenter will want to keep to their allotted time slot. We will include a timer in the editor to aid in rehearsal.

Introducing the Stopwatch class

The `Stopwatch` class (from `dart:core`) provides much of the functionality needed for this feature, as shown in this small command-line application:

```
main() {  
    Stopwatch sw = new Stopwatch();  
    sw.start();  
    print(sw.elapsed);  
    sw.stop();  
    print(sw.elapsed);  
}
```

The `elapsed` property can be checked at any time to give the current duration. This is a very useful class as, for example, it can be used to compare different functions to see which is the fastest.

Implementing the presentation timer

The clock will be stopped and started with a single button handled by the `toggleTimer` method. A recurring timer will update the duration text on the screen, as follows:



If the timer is running, the `updateTimer` and the `Stopwatch` in the field `slidesTime` is stopped. No update to the display is required as the user will need to see the final time:

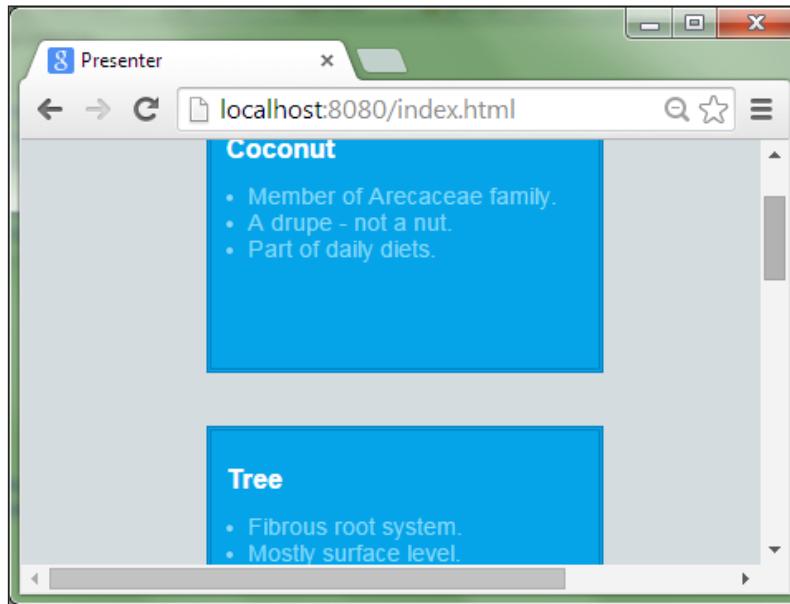
```
void toggleTimer(Event event) {
    if (slidesTime.isRunning) {
        slidesTime.stop();
        updateTimer.cancel();
    } else {
        updateTimer = new Timer.periodic(new Duration(seconds: 1),
        (timer) {
            String seconds = (slidesTime.elapsed.inSeconds %
            60).toString();
            seconds = seconds.padLeft(2, "0");
            timerDisplay.text =
            "${slidesTime.elapsed.inMinutes}:${seconds}";
        });
    }
}
```

The `Stopwatch` class provides properties for retrieving the elapsed time in minutes and seconds. To format this to minutes and seconds, the seconds portion is determined with the modular division operator `%` and padded with the string function `padLeft`.

Dart's string interpolation feature is used to build the final string, and as the `elapsed` and `inMinutes` properties are being accessed, the `{ }` brackets are required so that the single value is returned.

An overview of slides

This provides the user with a visual overview of the slides, as shown in the following screenshot:



The presentation slides will be recreated in a new full screen Div element. This is styled using the `fullScreen` class in the CSS stylesheet located in the `slideShowApp` constructor:

```
overviewScreen = new DivElement();
overviewScreen.classes.toggle("fullScreen");
overviewScreen.onClick.listen((e) => overviewScreen.remove());
```

The HTML for the slides will be identical. To shrink the slides, the list of slides is iterated over, the HTML element object is obtained, and the CSS class for the slide is set:

```
currentSlideShow.slides.forEach((s) {
    aSlide = s.getSlideContents();
    aSlide.classes.toggle("slideOverview");
    aSlide.classes.toggle("shrink");
    ...
});
```

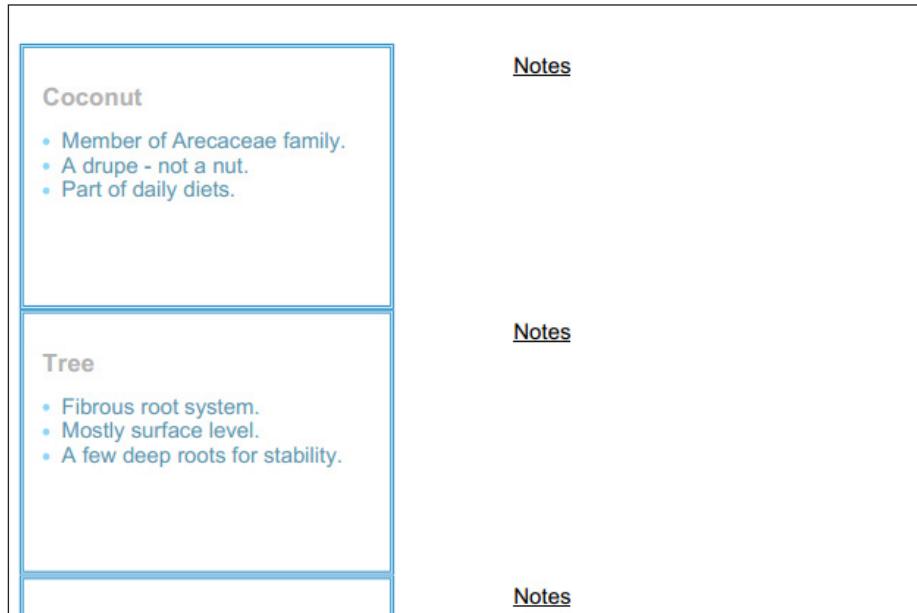
The CSS hover class is set to scale the slide when the mouse enters so a slide can be focused on for review. The classes are set with the `toggle` method, which either adds if not present or removes if they are. This method has an optional parameter:

```
aSlide.classes.toggle('className', condition);
```

The second parameter, named `shouldAdd`, is *true* if the class is always to be added and *false* if the class is always to be removed.

Handout notes

There is nothing like a tangible handout to give presentation attendees. This can be achieved with a variation of the overview display:



Instead of duplicating the overview code, the function can be parameterized with an optional parameter in the method declaration. This is declared with square brackets [] around the declaration and a default value that is used if no parameter is specified:

```
void buildOverview( [bool addNotes = false] )
```

This is called by the presentation overview display without requiring any parameters:

```
buildOverview();
```

This is called by the handouts display without requiring any parameters:

```
buildOverview(true);
```

If this parameter is set, an additional `Div` element is added for the `Notes` area and the CSS is adjusted for the benefit of the print layout.

Comparing optional positional and named parameters

The `addNotes` parameter is declared as an optional positional parameter, so an optional value can be specified without naming the parameter. The first parameter is matched to the supplied value.

To give more flexibility, Dart allows optional parameters to be named. Consider two functions—the first will take named optional parameters and the second will take positional optional parameters:

```
getRecords1(String query,{int limit: 25, int timeOut: 30}) {  
}  
  
getRecords2(String query,[int limit = 80, int timeOut = 99]) {  
}
```

The first function can be called in more ways:

```
getRecords1("");  
getRecords1("", limit:50, timeOut:40);  
getRecords1("", timeOut:40, limit:65);  
getRecords1("", limit:50);  
getRecords1("", timeOut:40);  
  
getRecords2("");  
getRecords2("", 90);  
getRecords2("", 90, 50);
```

With named optional parameters, the order they are supplied in is not important and has the advantage that the calling code is clearer as to the use that will be made of the parameters being passed.

With positional optional parameters, we can omit the later parameters, but it works in a strict left-to-right order, so to set the `timeOut` parameter to a non-default value, the limit must also be supplied. It is also easier to confuse which parameter is for which particular purpose.

Summary

The presentation editor is looking rather powerful with a range of advanced HTML controls, moving far beyond text boxes, date pickers, and color selectors. The preview and overview help the presenter to visualize the entire presentation as they work, thanks to the strong class structure built using Dart `mixins` and data structures using generics.

We have spent time looking at the object basis of Dart, how to pass parameters in different ways, and, closer to the end user, how to handle keyboard input. This will assist in the creation of many different types of applications, and we have seen how optional parameters and true properties can help document code for ourselves and other developers.

Hopefully, you learned a little about coconuts, too!

The next step for this application is to improve the output with full screen display, animation, and a little sound to capture the audience's attention. The presentation editor could be improved, as well—currently, it is only in the English language. Dart's internationalization features can help with this.

4

Language, Motion, and Sound

The sound and music are 50% of the entertainment in a movie.

– George Lucas

A slideshow presenter would be a poor choice of application to create a movie, but even with a smaller audience, there are lessons to be learned from those who have mastered the silver screen.

Simple animations in user interfaces are pleasing to the eye and enhance the overall experience.

Sounds have been present in web browsers for some time now, largely depending on plugins. Most users will be glad that the days of the autoplaying MIDI file are long gone, and most developers will be pleased that there will finally be a standard way to play sound files.

Going fullscreen

A slide show has been in the small screen mode so far and needs to be moved to a web browser equivalent of IMAX—fullscreen mode. The audience doesn't want to see a slideshow editor or desktop—just the slides.

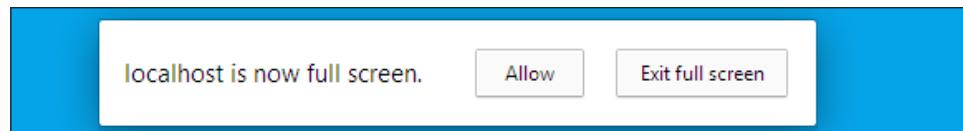


To open the sample code project of this chapter, open the `Presenter` folder in WebStorm and then open the `slideshowapp.dart` file.

Request fullscreen

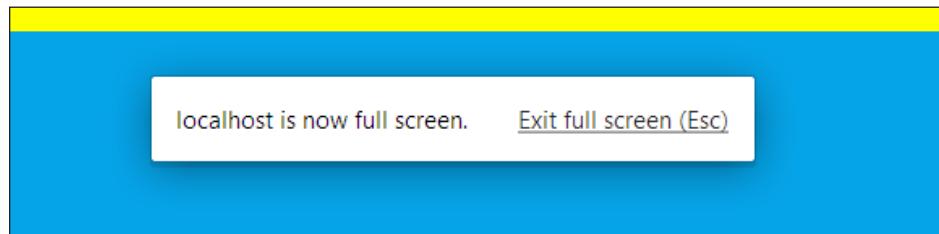
While the trend in web browsers has been to gradually reduce the size of screen controls—a trend accelerated by the Chrome browser—fullscreen web applications are still being accepted by users and standards are settled between the browsers. Lets have a look at the following screenshot:

For security, the web browser will prompt the user to give permission for the website to enter the fullscreen mode, which hides toolbars and status bars that are offscreen.



Once the fullscreen mode is entered, the user is notified of the domain that takes over the entire screen. The web browser also sets the Escape (*Esc*) key as a means for the user to exit the fullscreen mode. After a few seconds, it floats off the screen and can be brought back by moving the pointer back to the top of the screen.

The exact behavior and positioning may vary according to your web browser and operating system. For example, Internet Explorer tends to display notifications at the bottom of the page.



The `requestFullscreen` method is available on most page elements, though typically, `div` will be used as a container. The presenter application calls the `setupFullScreen` method from the constructor of the `SlideShowApp` object. A Boolean flag is used to keep track of the display mode:

```
qs("#btnStartShow").onClick.listen((e) {  
    isFullScreen = true;  
    liveFullScreen  
        ..requestFullscreen()  
        ..focus()  
        ..classes.toggle("fullScreenShow")  
        ..style.visibility = "visible";  
});
```

```
    liveSlideID = 0;
    updateLiveSlide();
}) ;
```

The `onFullscreenChange` event can be listened to so that an individual element can listen to the change in state. For example, a toolbar may not appear when the application is in fullscreen mode.

When you click on `div`, the `isFullScreen` flag is set, `fullscreen` is requested, and the input focus is set on the `liveFullScreen` object. Next, the `.css` file is set on `div` and it is made visible on screen. Finally, `liveSlideID` is set at the start of the presentation and the display is updated with the slide's contents.

Updating the interface for fullscreen

The editor screen will need a button to start the slideshow in fullscreen mode. The display will be an entirely new element and not just an enlarged editor preview.

Fullscreen will be a `div` element that fills the entire window and contains child `div` elements for background, complete with a background image, and the slide that is currently being displayed.

Updating keyboard controls for fullscreen

An event listener similar to the one implemented for the editor screen in the previous chapter is required here. The keyboard controls are set up in the `setupKeyboardNavigation` method of the `SlideShowApp` class:

```
void setupKeyboardNavigation() {
    //Keyboard navigation.
    window.onKeyUp.listen((KeyboardEvent e) {
        if (isFullScreen) {
            if (e.keyCode == 39) {
                liveSlideID++;
                updateLiveSlide();
            } else if (e.keyCode == 37) {
                liveSlideID--;
                updateLiveSlide();
            }
        } else {
            //Check the editor does not have focus.
            if (presEditor != document.activeElement) {
                if (e.keyCode == 39) showNextSlide();
                else if (e.keyCode == 37) showPrevSlide();
            }
        }
    });
}
```

```
        else if (e.keyCode == 38) showFirstSlide();
        else if (e.keyCode == 40) showLastSlide();
    }
}
});
```

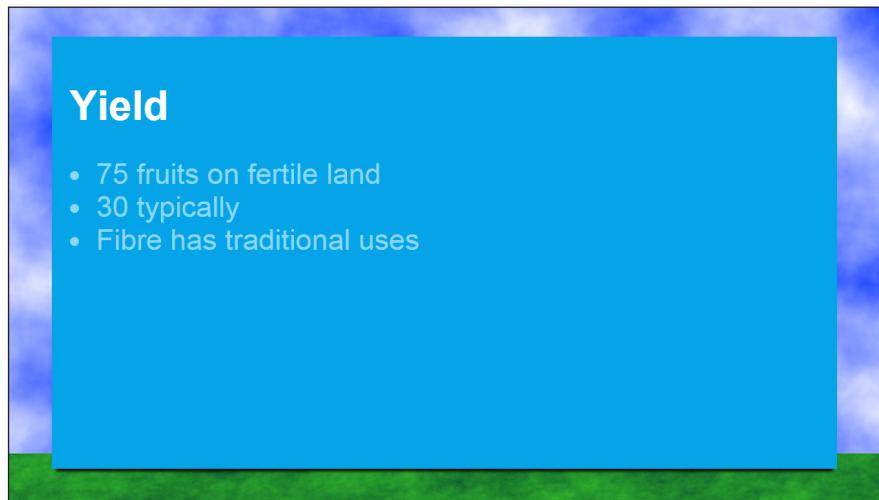
The `isFullScreen` flag is used to determine which mode the presentation application is in. When in the fullscreen mode, `liveSlideID` is incremented or decremented depending on the key pressed and the currently updated displayed slide.

Adding mouse controls

Clicking on the mouse button to advance the presentation is a fairly common feature and our application should support this too. This is implemented in the `setupFullScreen` method:

```
liveFullScreen = qs("#presentationSlideshow");
liveFullScreen.onClick.listen((e) {
    liveSlideID++;
    updateLiveSlide();
});
```

The `onClick` event is listened to on the fullscreen `div` element, and the slideshow is advanced to the next slide when this request is received. Let's have a look at the following screenshot:



Adding metadata

At the time of writing, some of the fullscreen APIs are marked as experimental in the Dart API documentation, and this is made clear in the documentation as well. By the time you read this, the fullscreen APIs should hopefully be stabilized and fully operational.

This is the normal process for new web browser features, which are eventually made stable. The adding of experimental APIs is likely to be a continuous process – it seems very unlikely that web browsers will ever be 100 percent complete! The `onFullscreenChange` Dart API documentation reads:

```
Experimental
ElementStream<Event> get onFullscreenChange
Stream of fullscreenchange events handled by this Element.
```

The source code for this part of the `dart:html` package has several annotations including the `@experimental` marker:

```
/// Stream of `fullscreenchange` events handled by this [Element].
@DomName('Element.onwebkitfullscreenchange')
@DocsEditable()
// https://dvcs.w3.org/hg/fullscreen/raw-file/tip/Overview.html
@Experimental()
ElementStream<Event> get onFullscreenChange =>
fullscreenChangeEvent.forElement(this);
```

Dart has a very flexible annotation system, and new systems can be created to add metadata to most of the components in Dart, such as classes, functions, and libraries. The pattern, as you probably already spotted, is used to start them with a `@` sign followed by a call to the `const` constructor or a reference to a compile-time constant variable.

Creating a custom annotation

Let's create our own `Temporary` annotation that will mark classes as temporary to the design of the application and will allow a label to record which planned build the temporary classes are scheduled to be removed:

```
import 'dart:mirrors';

class Temporary {
  final String removalBuildLabel;
  const Temporary(this.removalBuildLabel);
```

```
String toString() => removalBuildLabel;  
}  
  
@Temporary('Build1')  
class TempWidgetTxt {}  
  
@Temporary('Build2')  
class TempWidgetWithGFX {}  
  
void main() {  
    ClassMirror classMirror1 = reflectClass(TempWidgetTxt);  
    ClassMirror classMirror2 = reflectClass(TempWidgetWithGFX);  
  
    List<InstanceMirror> metadata1 = classMirror1.metadata;  
    print(metadata1.first.reflectee);  
  
    List<InstanceMirror> metadata2 = classMirror2.metadata;  
    print(metadata2.first.reflectee);  
}
```

The `Temporary` constructor allows a label to be set. This field must be `final` (a single-assignment variable or field) if this class is to be used as an annotation. The `main` function calls `reflectClass` to obtain the details of the classes and then accesses this metadata and stores it in a list. The `toString` method allows the metadata to be the output via `print`:

```
Build1  
Build2
```

Annotations are used extensively in the core Dart APIs, and in packages, they are used as the basis of frameworks, for example, web applications. They are not an obscure feature of the core SDK.

Translating the user interface text

My first ever professional software release did not involve any creation of code. It was a language update release for an existing package, and it was just as satisfying to mail out the CD-ROM (this was a while ago!) as something I had coded from scratch. Thankfully, it made me famous in Norway!

The translation of a software application is critical for its acceptance in some markets. This is even more true in the world of web applications. Dart has a package called `intl` that is used to help bring your applications to a global audience.

The `presenter` interface needs to be updated so that it displays the editor in French and Spanish. This will involve extracting the strings that we wish to use, obtaining translations, and integrating them back into the project.

The language will be selectable at runtime by user selection. The current OS interface language may not be the language that the user wishes to use the application in.

Exploring the `intl` package

The `intl` package provides functionality for internationalization and localization of Dart applications. This not only includes the important aspect of translating interface text into other languages but also includes the formatting of text (not every language is formatted from left to right), dates and numbers.

Locating strings to translate

The interface of an application largely consists of buttons, with a few other labels and controls. A phrasebook of the application's interface language can be created using `Intl.message` and stored as static functions in the `PhraseBook` class, which can be found in the `interfacetrans.dart` file. This organizes all the strings that are to be set on the interface elements at runtime using the `value` or `text` properties:

```
class PhraseBook {
    // Navigation controls.
    static String btnFirstSlide() => Intl.message("First",
        name: "btnFirstSlide", desc: "Button label - go to the first
slide.");
    static String btnPrevSlide() => Intl.message("Prev",
        name: "btnPrevSlide", desc: "Button label - go to the previous
slide.");
    ...
}
```

The functions are static so no instance of the class is required to call the function, so, for example, `PhraseBook.btnFirstSlide()` will return the string for the first slide. `Intl.message` has a series of named parameters. The `name` parameter needs to match the function name for the string extraction tools, which are covered in the next section. The `desc` (description) field provides context of where the string is used on the program. This is useful when the translator does not have access to the software that may not even be written yet.

`Intl.message` has extensive features, such as modifying a phrase due to it being plural or referring to a different gender. Plus, it can store more meta information such as example usages of the string to give the translator more context. Refer to the SDK documentation for more details.

Extracting the strings

The English strings for the application are now organized in a class file. Next, we need to extract the strings so that they can be sent out for translation. The `intl` package contains an extraction tool that can be run from the command line.



The next section of this chapter deals with the task of running tools from the command line. Ensure that the Dart SDK `bin` folder is on the executable path for your system so that `pub` and other Dart tools can be executed from anywhere on the command line.

Running commands with Dart pub

`Pub` is a tool with many uses in Dart, and one interesting feature of it is to run command-line tools that are part of Dart packages:

```
pub run packagename:programname
```

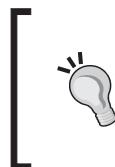
When this command is issued, `pub` will look for `programname.dart` in the packages `bin` folder:

```
pub run intl:extract_to_arb --output-dir=c:\transarb web\lib\interfacetrans.dart
```



Refer to <https://www.dartlang.org/tools/pub/package-layout.html> for more information on the package structure.

Ensure that the target directory already exists. The final parameter is the source file containing the functions with the translation strings.



The **Application Resource Bundle (ARB)** is a JSON-based format that is extensible and directly usable by applications.

You can find out more about the format of the project at <https://code.google.com/p/arb/>.

The JSON format ARB files contain much of the information from the Phrasebook class translated into a dictionary declaration:

```
{
  "btnFirstSlide": "First",
  "@btnFirstSlide":
    { "description": "Button label - go to the first slide.",
      "type": "text",
    ...
}
```

This easy-to-read and portable file format can be used by a range of applications to produce usable translations.

Obtaining translations

Google Translator Toolkit (<https://translate.google.com/toolkit/>) is a powerful online tool that accepts .arb files, among other formats. This is free to use and requires a Google account to login.

The screenshot shows the Google Translator Toolkit interface. On the left, there's a file input field with 'intl_messages.arb' selected, and a dropdown menu set to 'English'. Below it is a list of languages: Arabic, German, Portuguese (Portugal), Italian, Russian, Chinese (Simplified), French, Japanese, and Spanish. A search bar for additional languages is also present. On the right, there's a sidebar titled 'Types of content you can get translated' which lists various document formats like HTML, Microsoft Word, and OpenDocument Text, as well as video, other file types, and ads formats.

One useful feature of Toolkit is that it is tied to Google Translate so that a computer translation of the text is made available. This is the method that is used for the presentation string resources—it does not give perfect translations but is useful for initial testing and the layout of the interface.

Managing translations for even a small project can be a sizeable amount of work. Every new button means new strings, new translations, and more testing!



One alternative to Google Translator Toolkit is the open source project Pootle, which can be found at <http://pootle.translatehouse.org/>.

From my experience I can tell you that translators come from a range of backgrounds, are not always technically minded, and you have to deal with significant time differences. Time to use those important people-focused soft skills!

Each language can be translated separately and the tool allows progress to be tracked of the translation process. The translation file can be shared with other users and translations can be crowd-sourced if desired.

Once the translation is complete for the language, they can be downloaded as .arb files. Let's have a look at the following screenshot:

The screenshot shows the Pootle web interface for translating the 'intl_messages' file. The left pane displays the original English text, and the right pane shows the translated Spanish text. The interface includes a toolbar with 'Comments', 'Show toolkit', 'Save', and a 'Complete' button. The Spanish translation for 'First' is 'Primer'.

Original text:	Translation: English » Spanish	Status:
Message Name: btnFirstSlide Message 1 First	Message Name: btnFirstSlide Message 1 Primer	0% complete, 17 words
Description Button label - go to the first slide.	Description Button label - go to the first slide.	
Message Name: btnPrevSlide Message 2 Prev	Message Name: btnPrevSlide Message 2 Anterior	
Description Button label - go to the previous slide.	Description Button label - go to the previous slide.	
Message Name: btnNextSlide Message 3		

At the bottom, there are tabs for 'Automatic Translation Search' and 'Custom Translation Search'. The 'Automatic Translation Search' tab is active, showing results for 'Primer'.

Integrating the translations text

Now that the translations are safely stored as strings, it is time to bring them back into the Dart project by transforming these files into Dart source code files.

Notice the naming of the ARB files with `_es` and `_fr` that allow the `intl:generate_from_arb` tool to determine which language is in play. In this case, the Dart files are being put directly into the project in the `lib` folder:

```
pub run intl:generate_from_arb
--output-dir=web/lib
web/lib/interfacetrans.dart
web/lang/messages_es.arb web/lang/messages_fr.arb
```

The files created include the central `messages_all.dart` message, which is the main import Dart file. There is also a supporting file created for each language that is being used, `messages_es.dart` and `messages_es.dart`.

Changing the language of the user interface

To make use of the translations, the `messages_all.dart` file needs to be imported into `interfacetrans.dart`. The `intl` package needs to be told which language we want to use, and behind the scenes, it deals with loading the correct resources. The `setInterfaceLang` function in `interfacetrans.dart` carries out this task and is called by the `SlideShowApp` constructor:

```
setInterfaceLang(String langID) {
  initializeMessages(langID).then((_) {
    Intl.defaultLocale = langID;
    addInterfaceText();
  });
}
```

Best of all, the `addInterfaceText` method does not need to change, no matter how many new languages are added.

Adding a language combo box

For users and developers, it is useful to change the language of the interface anytime. We will add a combo box with a handler for the `onChange` event to allow the user to select a language from the list:

```
<select id="interfaceLang">
  <option>en</option>
  <option>fr</option>
  <option>es</option>
</select>
```

All the initialization for the interface has been wrapped in the `setInterfaceLang` method, so the only parameter that is to be passed is the value from the combo box (`SelectElement`):

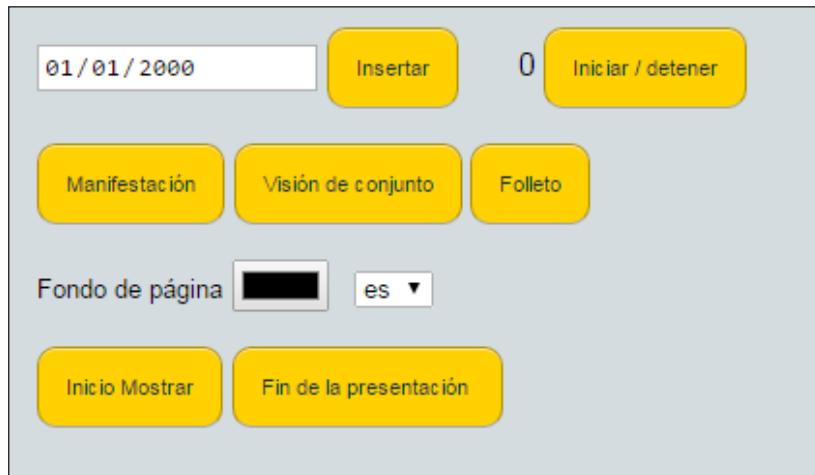
```
setInterfaceLang("en");
qs("#interfaceLang").onChange.listen((e) {
  var lang = qs("#interfaceLang").value;
  setInterfaceLang(lang);
  langInterface = lang;
...
});
```

This is set up with the other listeners in the `SlideShowApp` constructor and triggers the change of the interface requested by the user. The language can be switched to a French user interface with a single click at runtime, and the change occurs almost instantly.



The length of strings can vary wildly from language to language - the button layout allows expansion. Note the differences in button between the Spanish and French translations.

This is a factor to consider when planning the interface layout of your multilingual applications. It is probably best to assume that other languages will be longer and allow some room for expansion.



Advanced as the application is, I am afraid that it is left to the user to translate the demo coconut presentation material!

Working with dates

The ISO 8601 standard for dates is 'Year Month Day', and naturally, no country in the world uses this standard day to day! A typical variation is to have the month or day first, and then the separator character is used between digits in shorthand.

The `intl` package can help provide the appropriate format and translations for each language. Translation strings are required too as the name of the day or month may be required.

Date handling is implemented in the `slideShowApp` class:

```
import 'package:intl/date_symbol_data_local.dart';
```

A field is used to keep a reference to the active `DateFormat` object:

```
DateFormat slideDateFormat;
```

We will consider how this formatter is initialized and kept up to date with the changes made to the users language.

Formatting for the locale

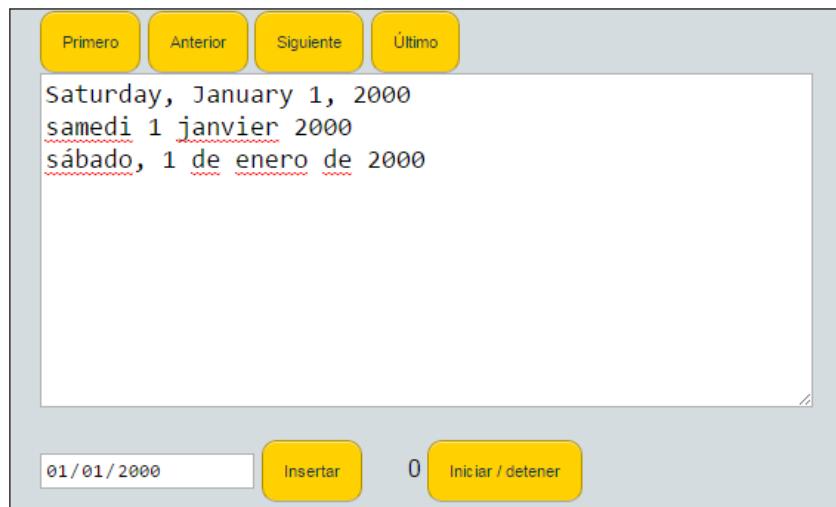
The locale for the date formatter will need to be updated whenever the interface language is changed:

```
qs("#interfaceLang").onChange.listen((e) {
    var lang = qs("#interfaceLang").value;
    setInterfaceLang(lang);
    langInterface = lang;
    initDefaultDate();
});
```

When a locale has been requested, the appropriate `DateFormat` object needs to be created:

```
initDefaultDate() {
    initializeDateFormatting(langInterface, null).then((d) {
        slideDateFormatter = new DateFormat.yMMMMEEEEd(langInterface);
    });
}
```

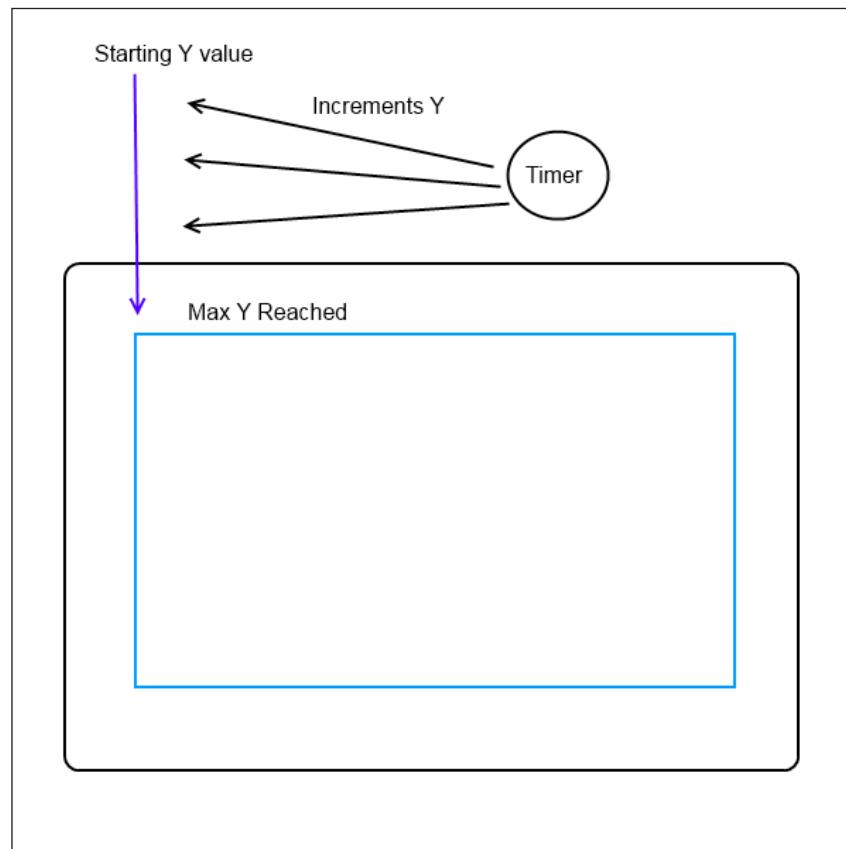
Once this is in place, it is possible to change the interface language using the combo box and get a date in the correct format and language.



As a final thought on this topic, most web applications are not nearly as dynamic in their choice of language. Typically, an interface language selection is a per user configuration option that is set up for each session when the user logs in.

Animating slides

The fullscreen display only has one slide at any time. A *new* slide will be initially placed on the top of the screen and will be gently lowered into view. The *old* slide will, from the presentation viewer's point of view, vanish instantly.



Using a timer

The periodic timer in Dart should be a familiar friend of yours by now. This is used to check whether the slide has reached the final position. If it is not there yet, then the position is updated:

```
new Timer.periodic(new Duration(milliseconds: 50), (timer) {  
    if (isFullScreen && liveSlideY < 0) {  
        liveSlide.style.top = liveSlideY.toString() + "px";  
        liveSlideY += 50;  
    }  
});
```

You may be concerned that the `Timer` object runs all the time, even when a presentation is not running fullscreen. In reality, they are very lightweight and the work that the animation performs is so minimal that it will have a negligible impact.

Playing sound in the browser

This feature was deliberately left until last as continually listening to the same sound effect can become quite irritating! MP3 is probably the best supported format for various web browsers, with one exception – **Dartium**.



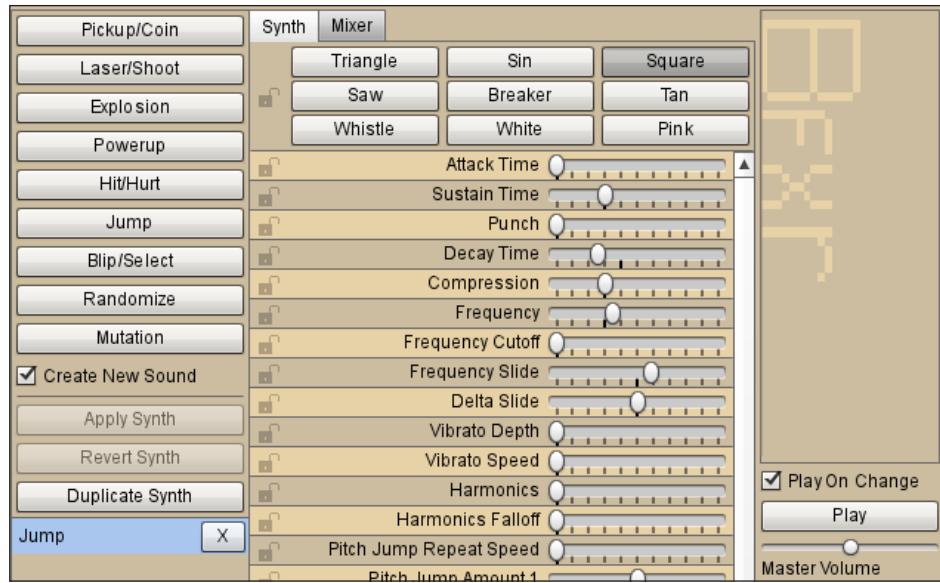
Dartium is built from the Chromium open source project. The project does not include any proprietary codecs including MP3. When creating projects, it is sometimes fine to just use MP3 and test the audio parts of the application in another browser.

For this project, three formats of the same sound file are provided. The OGG format will run from Dartium and most browsers. If you are having trouble, try another format.

Producing sound effects

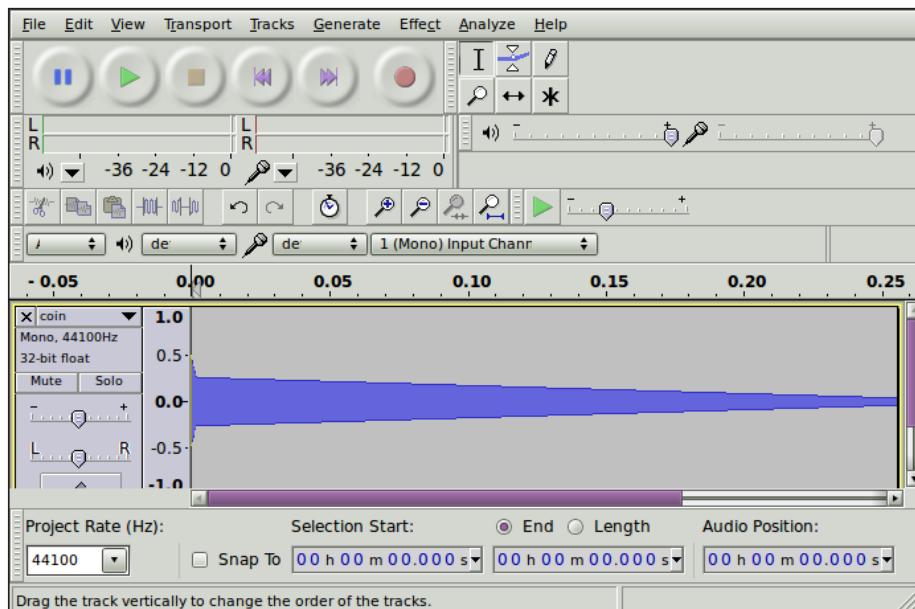
To create a sound effect to play when the slide changes, a tool such as Bfxr can be used. It can be found on the Web at <http://www.bfxr.net/> (where there are links to downloads for standalone versions as well) and is a powerful audio creation tool.

As it was created for use during game jams (time-constrained game coding contests), it is easy to pick up and fast to use. Let's have a look at the following screenshot:



Creating sound files

Audacity (<http://web.audacityteam.org/>) is a popular and stable audio editing and recording package. It is an open source desktop application and is available for all the major OS platforms. Let's have a look at the following screenshot:



You can perform some last minute tweaks to your sound and export it (**File | Export..**) in a variety of formats (such as, MP3, Ogg, and WAV). On some systems, you may need to download an MP3 codec.



Keep an eye on the file size and duration time, though! Ideally, the sound should be around 200ms long.



Loading sounds

The sound is loaded in a method that is called from the constructor of the `SlideShowApp` object, and as you expect with Dart, this is an asynchronous operation:

```
List loadAudio() {  
    slideChange = new AudioElement("snd/slidchange.ogg");  
    slideChange.preload = "auto";  
    return [slideChange.onLoad.first];  
}  
  
playSnd() {  
    slideChange  
        ..play()  
        ..onEnded.listen(done);  
}  
  
done(e) {  
    slideChange.load();  
}
```

The `onLoad` property of `AudioElement` is a stream of load events for this object (it is possible for audio and other media elements to load more than one file).

The `first` property of this stream waits for the first load event and then stops listening to the stream. This is a useful way to ensure that all media, such as images and sounds for a game, are loaded before they are used.

Playing back sounds

Once all that set up is done, actually playing the audio is merely a case of calling the `playSnd` method and ensuring that the speakers are plugged in! The `onEnded` event is listened to and a callback is provided. This deals with a browser quirk that requires the file to be reloaded before it can be replayed.

That quirk serves to remind us that although audio support in browsers has greatly improved, it is geared toward large media playback, and playing short sound effects can be a little more work than it should be.

Here's a website that you can use to help contribute to Dart development with either bugs or feature suggestions:

<http://dartbug.com>

This redirects (at time of writing) you to a Google Code project. The Google Code service is being wound down but will be kept going for some time for the Chromium project. As the Dart team is part of the Chrome team at Google, it is possible that parts of the project will live on both Google Code and GitHub.

If you have an idea or think you have found a bug, it is a good idea to search the existing issues or discuss it on **Google+** (<http://g.co/dartisans>) or a mailing list before raising that issue.



Summary

The presentation application now has enough features for the final slideshow to be pleasing to the eyes and ears of the receiving audience.

Behind the scenes, the interface can be used by those who speak languages other than English. Also, the software is ready for easy translation into more languages with minimal code changes.

We looked at the powerful metadata features of Dart and saw how they are used in the SDK and user applications. We saw how the pub tool allows us to use tools that are in Dart packages.

With the text editor and presenter applications, the client-side story of Dart has been well explored; taking it to the server-side will give us a view of Dart in another habitat!

5

A Blog Server

Forget about someone's resume or how they present themselves at a party. Can they blog or not? The blog doesn't lie.

– Nick Denton

I wish I could remember which was the first blog that I read on the Web, but I do remember reading (and having) diary sections on a home page, long before the term "blog" was invented! Despite the meteoric rise of various social media platforms, blogging has maintained its place.

It is time for us to explore the server side of the Dart story, and a blog gives us great opportunity to explore different aspects of this language. On the server side, new capabilities are made available, as the context is outside the web browser.

The Hello World server example

It is very simple to create a minimal Hello World server in Dart. Open the sample `HelloWorld` file of this chapter in the Dart editor. This project is a command-line project, so the `main.dart` file is placed in the `bin` folder:

```
import 'dart:io';
void main() {
  HttpServer.bind('127.0.0.1', 8080).then(
    (server) {
      server.listen((HttpRequest request) {
        request.response
          ..write('Hello Dart World')
          ..close();
      });
    });
}
```

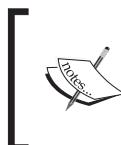
The output is shown in the following screenshot:



The `dart:io` package is important for the server side. It allows us access to features that are not available to the code running in the web browser context due to security, such as filesystem and networking.

Launching the server program does not open up a web browser like in our previous application. You have to manually open a browser, then go to the address and to the port that you have bound to. Also, if a change is made in the server code, the application has to be stopped and restarted for the change to be seen

The `HttpServer.bind` method takes two parameters – an IP address and a port number. The server is then attempted to be bounded to this network configuration, and if this is successful (and it usually only fails if the binding is already in use), the function is then executed.



Network setups vary from computer to computer. The sample code of this chapter uses ports and IP numbers that are generally available; however, you may have to adjust these for your system – this is likely if you have a busy development system!

The returned server object is set to listen to HTTP requests on that port. If any are received, the message `Hello Dart World` is passed to the `response` method of the `request` object, and then, very importantly, the response is closed.

This simple example shows the structure of all HTTP-based server applications – bind to the network and then handle requests. Of course, handling requests and producing quality output is usually a good deal more complicated.

If you are not too familiar with networking, here is a quick recap! **IP (Internet Protocol)** addresses help us identify machines (usually) on the network, and consist of 4 numbers that are separated by a full stop for example 192.168.0.1.

 Most computers have a special IP of 127.0.0.1 only for network communications on that machine. This is a computer's network address that is used to create a connection with itself and cannot be used by remote machines. Other than this distinction, it operates as an equal to any other IP address.

Ports help identify which application or service the connection is attached to. For an analogy, IPs are people's names and ports are the topics of conversation. To get the right information, you want to get both correct!

While the application is still running, go to `main.dart` and change the message, that is the output, by adding `!` to the end of the line. Refresh the browser page and you will find that nothing has changed. The Dart code runs on the server and does not get executed in the browser anymore, so the simple refresh feature to see changes is sadly gone now.

The good news is that starting and stopping a server application is very easy, and we can use any web browser to test the application. Breakpoints can still be placed in the code and will be triggered when the page is served.

A blog server

A blog server needs to respond in the same manner to network requests as any other web server so that it can be used by the existing clients. The main clients to the server will be web browsers, such as Google Chrome and Internet Explorer.

A blog server needs to run as a reliable application on different operating systems. Later in this chapter, we will take a look at how a Dart application can be deployed – helper applications and dependencies are required to achieve this.

Introducing the HTTP protocol

The HTTP protocol is straightforward and lightweight. Generally, a server responds to a request with a set of headers stating that the request was successful or not. It then sends the details of what it is going to send, and the content itself, before closing the connection.

The HTTP protocol requires that all headers must be sent before any content. Dart helps enforce this, and if you try to modify the headers once some content has been sent, `HttpException` will be thrown.

Starting up the server

The `main.dart` entry point for a blog server is similar to the Hello World example. The actual functionality of serving the content is handled by the `BlogServerApp` class, of which a single instance is initialized:

```
import 'dart:io';
import 'package:blogserver/blogserver.dart';

main() {
    print("starting");
    var blogServer = new BlogServerApp();
    print("Starting blog server...");

    HttpServer.bind('127.0.0.1', 8080).then((server) {
        server.listen(blogServer.handleRequest);
    });
}
```

It is good practice to give some feedback to the user when the server has started. This server is single-threaded, with all requests being handled by one process in a serial manner.

Storing the blog posts format

Blog posts are held in a simple text file format, with the filename being the blog post ID and a `.txt` file extension:

```
1st Line - Date
2nd Line - Title
3rd Line until end of file - Post Body
```

A blog post in `1.txt` will look as follows:

```
01/05/2015
Giraffe Facts
<p>
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut
enim ad minim veniam, quis 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, sunt in
culpa qui officia deserunt mollit anim id est laborum.
</p>
```

The content is the standard text `Lorem ipsum`, which has been used by typesetters for centuries.

Reading text files

Now that we are working outside the browser, it is possible to work with the file system more directly. Let's take a look at how to read in a text file, and print out each line to the standard output:

```
void main() {
    File myfile = new File("readme.txt");
    myfile.readAsLines().then(
        (List<String> lines)
        {lines.forEach( (line) => print(line) ); }
    );
}
```

To read in a blog post in our defined format, the list of lines needs a little more processing. This is implemented in the process method that is called from the BlogPost constructor in the `blog.dart` source file:

```
BlogPost(String filename, this._id) {
    File postFile = new File(filename);
    _source = postFile.readAsLinesSync();
    process();
}
```

The file is read using the synchronous version of `readAsLines`, so the process call does not need to be placed in a then handler, unlike the previous example:

```
process() {
    _html = "";
    _date = _source[0];
    _title = _source[1];

    _html = "<h2>$_title</h2><b>$_date</b><br/>";
    _html += "<img src=\"$_id.png\" align=\"left\">";
    _source.sublist(2).forEach((line) => _html += line);
    _html += "<br/><a href=\"post$_id.html\">Permalink</a>";
}
```

The first two lines are selected using the `[]` method. The remaining lines are broken off from the list into a sublist, and are directly used to build up the HTML file that will be the output. The content is assumed to be a valid HTML.

Reading a folder of files

In the project, the `posts` subfolder of the `content` folder contains all the blog content. To obtain a list of all the files to load, the `Directory` object can be used to obtain the list, which can be processed with `forEach`:

```
void _loadPostList() {
    Directory blogContent = new Directory(_pathPosts);
    blogContent.list().forEach((File f) {
        String postFilename =
            path.basename(f.path).replaceAll(".txt", "");
        int id = int.parse(postFilename);
        IDs.add(id);
        postPaths[id] = f.path;
    }).then((v) {
        IDs.sort();
        IDs = IDs.reversed.toList();
    });
}
```

The file paths to the blog posts are stored in `Map`, so that they can be retrieved by the `id` value. `IDs` are stored in a list that can then be ordered (there is no guarantee that the filesystem will supply the list of filenames in any particular order), and so that we can easily retrieve posts with the most recently published post first.

Request handling

The incoming `HttpRequest` object contains a wealth of information about the client—the request made and the network connection to the server:

```
handleRequest(HttpRequest request) {

    if (request.uri.path.endsWith(".html")) {
        _serveTextFile(request);

    } else if (request.uri.path.endsWith(".png")) {
        _servePngFile(request);

    } else if (request.uri.path == "/robots.txt") {
        _serveRobotsFile(request);
    }
    else {
        _serve404(request);
    }
}
```

In the context of a blog server, we are most interested in the **Uri (Uniform resource identifier)** property, more commonly called a URL, of the `request` object that gives the path of the resource that is requested.

For example, if the request sent to the server is `http://127.0.0.1:8080/post6.html?test=false` (the query string is not used by the blog server), the value of `request.uri` would be `/post6.html?test=false` and `request.uri.path` would have the value `/post6.html`.

For the page, image, or file that is being requested, the `request.uri.path` property gives the desired details. The requests for the `robots.txt` file are handled differently than blog posts and images, and this will be detailed in a later section.

Serving text

If a page is requested, we need to provide feedback to the client making the request. Our response should be firstly to tell the client what format of data we will be responding with (the content type), also known as a MIME type. Secondly, we want to let the client know that we have understood the request (`HttpStatus.OK`), and are about to send the requested content:

```
void _serveTextFile(HttpServletRequest request) {
    String content = _getContent(request.uri.path.toString());

    request.response
        .headers.set('Content-Type', 'text/html')
        .statusCode = HttpStatus.OK
        .write("""<html>
<head><title>$BlogTitle</title></head>
<body>
$content
</body>
</html>""")
        .close();
}
```

Once these items are set, the content for the page can be served to the requesting client, using the `response` property of the `request` object. Finally, the `close()` method is called, so that the client knows that the server has finished sending content.

MIME stands for **Multipurpose Internet Mail Extensions**. It is an IETF standard that is used to indicate the type of data that a file contains. This list is continually getting longer as more types of data are shared online. For more details, refer to

<http://www.iana.org/assignments/media-types/media-types.xhtml>.



Robots.txt

People with web browsers are only one type of web users. There are numerous bots and spiders out on the live Internet. To give some guidance to the nonhuman visitors, the `robots.txt` standard tells the bot whether they are welcome, and if it is a search engine, then how to index it. For the purpose of the blog server, we want to welcome search engines to index the content to bring in more readers.

The text content of the response for this will be:

```
User-agent: *
Disallow:
```

The `robots.txt` request needs to be handled and served in the same element as a regular text page:

```
void _serveRobotsFile(HttpServletRequest request) {
    request.response
        ..statusCode = HttpStatus.OK
        ..headers.set('Content-Type', 'text/html')
        ..write(RobotsTxt)
        ..close();
}
```

The page is served with a positive OK HTTP status.

Rendering a single blog post

The blog posts will be served on a single page for the permanent links from Uris, such as `http://127.0.0.1:8080/post6.html`, and the `_getContent` method parses the Uri path to get the ID number of the post:

```
if (path.startsWith("/post")) {
    String idfromUrl =
        path.replaceFirst("/post", "").replaceFirst(".html", "");
    int id = int.parse(idfromUrl);
    return hostedBlog.getBlogPost(id).HTML;
}
```

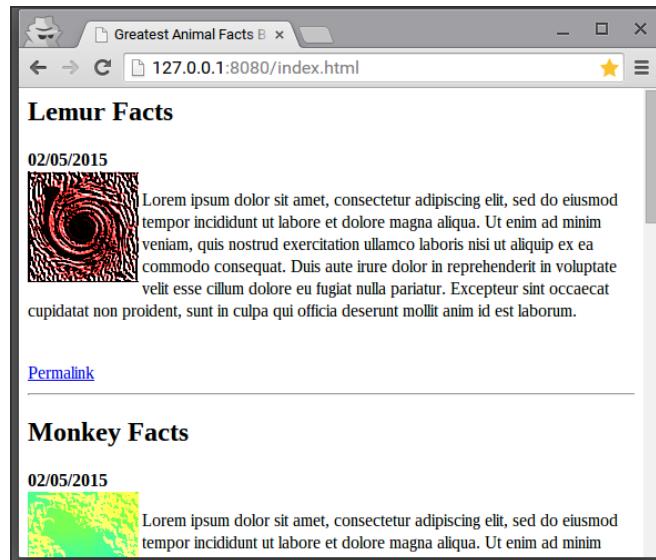
The output is shown in the following screenshot:



Once the ID is extracted, the blog post is retrieved and the HTML file is returned to be served as a text HTML file.

Rendering the index page

A request for the `index.html` page from this web server will show the five most recent blog posts, with the newest one at the top of the page, as shown in the following screenshot:



The blog posts are joined together and served as a single page:

```
String getFrontPage() {
    String frontPage = "";

    IDs.sublist(0, min(5, IDs.length)).forEach((int postID) {
        BlogPost post = getBlogPost(postID);
        frontPage += post.HTML + "<hr/>";
    });

    return frontPage;
}
```

The list of IDs processed previously is used to get the five most recent posts (or less) and the content for the posts is fetched and joined in a single HTML element.

Serving images

The HTTP header information for the PNG file type is the MIME type and the number of bytes in the image. A status code `OK` is also returned:

```
void _servePngFile(HttpServletRequest request) {
    var imgp = request.uri.path.toString();
    imgp = imgp.replaceFirst(".png", "").replaceFirst("/", "");

    image = hostedBlog.getBlogImage(int.parse(imgp));
    image.readAsBytes().then((raw) {
        request.response
            ..statusCode = HttpStatus.OK
            ..headers.set('Content-Type', 'image/png')
            ..headers.set('Content-Length', raw.length)
            ..add(raw)
            ..close();
    });
}
```

Serving an image or other binary files requires a little more information upfront, but once that is done, it is simply a matter of transferring the raw bytes and closing the connection.

Locating the file

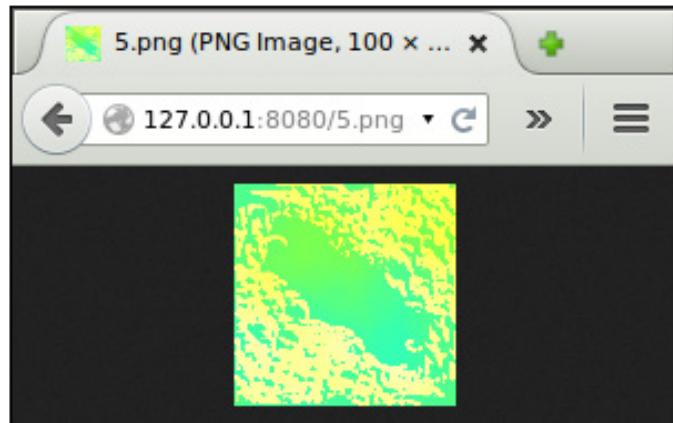
The `getBlogImage` method of the `Blog` class returns the PNG file for display on the blog post. The `Blog` class builds up a list of image filenames when `_loadImgList` is called in the constructor. This is in the same fashion as the previously described `_loadPostList` function:

```
File getBlogImage(int index) {  
    return new File(imgPaths[index]);  
}
```

The ID is used as a key to get a value from a map, and the full file path is returned. The web server's Uri to a file is completely independent of the location of the filesystem, and it is the logic of the application which determines which file is to be delivered to the client.

Serving a single image file

The server is unaware of the complete structure of the web page (HTML and images) served for `index.html`. It simply returns responses to the web browser's HTTP requests in a singular manner:



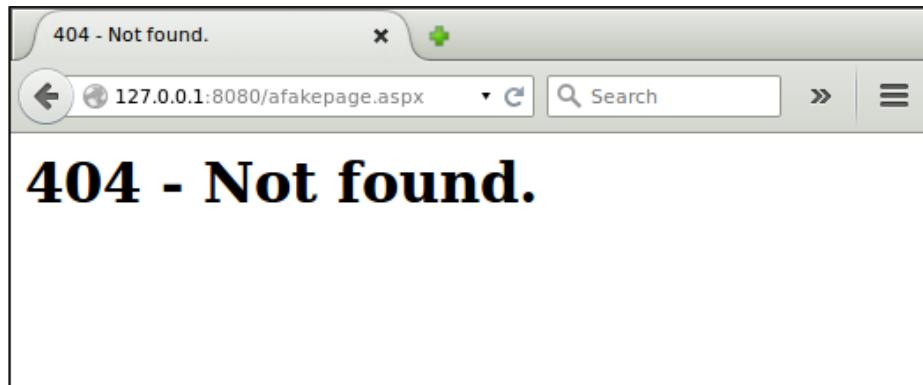
A single image can be requested directly outside the HTML page.

Serving a 404 error

Even nontechnical web users are familiar with the 404 error that indicates a page on a website cannot be found:

```
void _serve404(HttpRequest request) {  
    request.response  
        ..statusCode = HttpStatus.NOT_FOUND  
        ..headers.set('Content-Type', 'text/html')  
        ..write(page404)  
        ..close();  
}
```

This page is a regular text page (the constant string `page404` contains the content), and the client is further informed of the 404 status by setting the status code to `NOT_FOUND`, as shown in the following screenshot:



Let's hope that the blog server does not need to serve many of these pages!

Introducing Dart's server frameworks

The blog server in this project is written directly on top of `dart:io`, so you can see what is going on. As you might expect, there are a number of frameworks for Dart to help write server applications, so that basic tasks, such as serving text files, do not have to be implemented from scratch for every application.

These are all available at the pub package library.

Redstone

Redstone is described as "a server-side micro framework for the Dart platform". It uses simple code annotations to expose Dart functions to the Web. The Redstone Hello World example is very clear, and shows how annotations can hide the complexity from the developer so that they can focus on the application logic:

```
import 'package:redstone/server.dart' as app;

@app.Route("/")
helloWorld() => "Hello, World!";

main() {
  app.setupConsoleLog();
  app.start();
}
```



You can find out more about Redstone at <http://redstonedart.org/>.



Rikulo

Rikulo modestly describes itself as a lightweight – Dart web server. It has a list of interesting features including request routing, template engine, and the **MVC** (**Model-view-controller**) design pattern. Rikulo Stream is the server story and its Hello World example to serve static content is a single line:

```
import "package:stream/stream.dart";
void main() {
  new StreamServer().start();
}
```



You can find out more about Rikulo at <http://rikulo.org/>, where several projects are listed.



Shelf

Shelf claims that it makes it easy to create and compose web servers, and parts of web servers. Its pipeline-based model allows middleware (such as a request logger in its Hello World example) to be added in:

```
import 'package:shelf/shelf.dart' as shelf;
import 'package:shelf/shelf_io.dart' as io;

void main() {
    var handler = const shelf.Pipeline().addMiddleware(shelf.
logRequests())
    .addHandler(_echoRequest);

    io.serve(handler, 'localhost', 8080).then((server) {
        print('Serving at http://${server.address.host}:${server.port}');
    });
}

shelf.Response _echoRequest(shelf.Request request) {
    return new shelf.Response.ok('Request for "${request.url}"');
}
```



Find out more about Shelf at <https://github.com/dart-lang/shelf>, and as this URL suggests, this package is from the Dart team.



Deployment

After a long tough software development project, having issues to get the final product running on the live system can be extremely stressful if there is a last minute hitch. It is good to think about deployment early in the development process, so that the application is suitable for the target environment.

We will take a look at how to deploy Dart on two common environments—Windows and Unix. Of course, every individual system can vary on the details, so you may have to check for equivalent settings or programs on your system and take into account any currently running applications, such as web servers.

Dependencies

The Dart SDK itself is the key dependency for any Dart server application. The application is run via the `dart` command-line application, with the `main.dart` script passed as a parameter.

This is straightforward for an interactive session; however, for a server application, we need to run the application when there is no user logged into the machine.



The Dart SDK is available as a separate download (not including developer tools) from
<https://www.dartlang.org/downloads/>.



Deploying on Unix

On a Linux or other Unix-like system, web applications can be launched within a console application called `screen` (lowercase `s`). The `screen` command allows the creation of other console sessions that can be left running in the background once the main session has ended.

A `screen` application can be created – the application is started, then the `screen` detached from, and the main session is ended. The web application is then left running. If a change has to be made, the detached screen can be reattached (this can be months or years later!) for any adjustments; for example, installation of a new version or configuration change.

This can be used on hosted systems where shell access (usually SSH) is allowed directly on a server.

Using the `screen` command

Log in to an interactive shell on your system as an appropriate user. To check whether `screen` is installed, run the following command to get the version and description:

```
Screen
Screen version 4.01.00devel (GNU) 2-May-06
```

```
Copyright (c) 2010 Juergen Weigert, Sadrul Habib Chowdhury
Copyright (c) 2008, 2009 Juergen Weigert, Michael Schroeder
...
```

If the program is not installed, use your package manager to install the package, simply called `screen`.

Launching a screen

Once screen application is running, press the *Enter* key and you will be at Command Prompt again. Press *Ctrl+a*, and the version screen will be displayed, showing us that we are not in a normal shell.

Use `cd` to go to the `BlogServer` folder, and start the Dart blog server:

```
dart bin/main.dart
```

Next, press *Ctrl+a*, *Ctrl-d* and you will see a message similar to the following one:

[detached from 398 pts-2.localhost]

You should be back in your regular terminal now. Run the command `top` (or any other program that lists the processes that are running on the system) to get a list of processes, and you will see Dart running in the background.

Run a web browser and navigate to the `index.html` page. The blog will be served from the editor debugging session.

To reconnect to this screen, use the command `screen -r` and you will be back in the session.



The `screen` command is a powerful tool, but it can be a little confusing to remember which screen is in play. It is recommended that you commit keyboard shortcuts to memory for navigation.

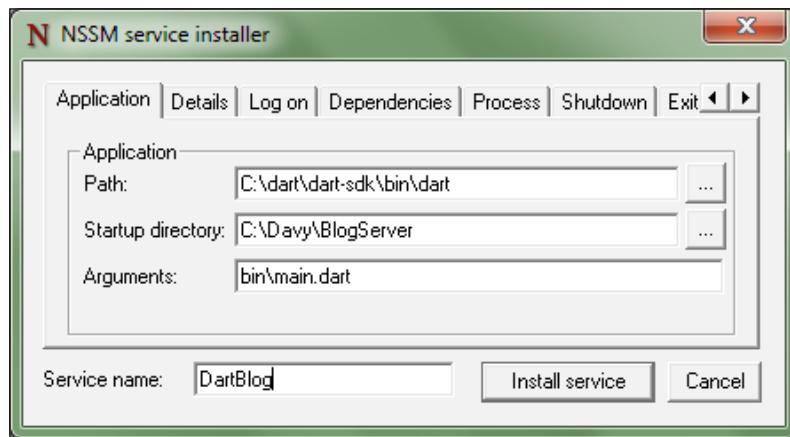
Deploying on Windows

On Windows, server applications are usually run as Window Services, or hosted via Internet Information Services in the case of .Net applications.

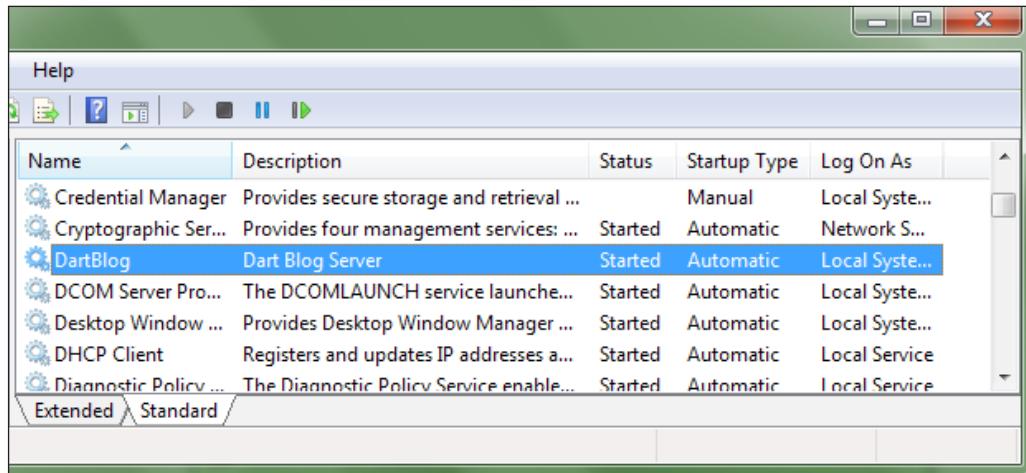
It is a relatively simple operation to place the Dart `bin/main.dart` command into a batch file so that it runs with a double-click on interactive sessions. It would be best to remove this requirement for a user to be logged in if, for example, the machine is unexpectedly rebooted, or the application crashes.

Using the NSSM tool

The free tool **NSSM** (**N**on-**S**ucking **S**ervice **M**anager) (available at <http://nssm.cc/>) has a range of features to manage services. One of these is to install any application as a service, as shown in the following screenshot:



The DartBlog service is installed and set as automatic startup, and is run under the **Local System** account:



Note that the service is stopped when it first installs. Depending on what the application does, the service will need to run under an appropriate user account, such as writing to a file, or performing other operations.

Using a Microsoft solution

Some system administrators may be happier if they use a more official Microsoft-based solution.

As an alternative, Microsoft provides a tool called **srvany**, which is service that is used to launch other applications. It is an older tool (dating back to NT4 Resource Kit), but is commonly used for this purpose. For more information, refer to <https://support.microsoft.com/en-us/kb/137890>.

Load testing

It is good to know the limits of an application long before you ever reach them. The `HttpClientRequest` function can be used to create requests to the blog server, and the `HttpClientResponse` function can be used to receive incoming data from the blog server.

The `statusCode` property will be checked to ensure that the page request was successfully handled by the server.

Building a simple load tool

A simple benchmark for the blog server would be the time taken to serve 1000 files. The `getUrl` method triggers the actual request with the first `then` clause, closing the request to the server. The following `then` clause handles the actual response from the server.

This method can be used to monitor a live website and perhaps trigger a notification if a status other than `HttpStatus.OK` is received:

```
import 'dart:io';

main() {
  print("Starting...");

  var url = "http://127.0.0.1:8080/index.html";
  var hc = new HttpClient();
  var watch = new Stopwatch();
  int attemptedRequests = 1000;

  print("Starting testing...");
  watch.start();
```

```
for (int i=0;i<attemptedRequests;i++)
{
    hc.getUrl(Uri.parse(url))
        .then((HttpClientRequest request) => request.close())
        .then((HttpClientResponse response) {
            if (response.statusCode==HttpStatus.OK)
                print("$i, ${response.statusCode}, ${watch.elapsed.
inMilliseconds}");
        });
}
```

The Stopwatch class can be used to measure the time taken and reported to standard output via a print statement. As the responses arrive asynchronously, the status is printed after each response. The request number, status code, and elapsed time are printed with a comma between each value, so that the data can easily be manipulated in a spreadsheet application.

Try putting a print statement directly after the loop has finished and you will see the print run before the first response is received from the server (or soon after).

On my modest Linux laptop, the server was able to serve `index.html` 2000 times in 3.8 seconds—not too bad! Try experimenting with the request number; however, you are likely to hit a limit on open files, as this simple benchmark fires many simultaneous requests.

Summary

We saw how the `dart:io` package can help build an HTTP server very quickly, and how frameworks can make this even easier. Using some simple processing of the request string, we can do more than serve the files flatly from the filesystem, such as serving the most recent entries on the front page.

The freedom of being out of the browser allows the manipulation of the filesystem. Despite not being on the Web, there is still good support for HTTP operations, which we used to test the performance of the server. Dart server applications can easily be installed on industry-standard host operating systems in a standard manner.

Our blog server is off to a good start. To progress, it needs to do more for the author, the reader, and the growing number of intelligent web crawlers and bots out on the Internet. We will provide an editor to create posts, cache requests to serve pages faster, and expose the posts in other data formats.

6

Blog Server Advanced

One reason I encourage people to blog is that the act of doing it stretches your available vocabulary and hones a new voice.

– Seth Godin

Blogs (a shortened form of the term "web logs") grew from simple homepages and made it faster to get content online for casual users. They also took care of some of the metainformation and publishing that allows content to spread around the Internet and can be found by machines and human consumers.

On the server-side, Dart has greater access to the filesystem and is not limited by the capabilities of a browser. This chapter explores these features more as we add more features to the blog server.

Logging

Web servers such as **Apache** and **IIS** track a visitor's web browser usage, which helps webmasters understand how visitors use their site and how many people view the pages. This good practice also helps web developers find problems in their site and helps the developers of a web server software diagnose issues.



For general-purpose logging in your Dart applications, take a look at the `logging` package, which is authored by the Dart team and is available at <https://pub.dartlang.org/packages/logging>.

Writing text files

With `dart:io`, we already read files from the file system, and writing operates in a similar manner, as follows:

```
import 'dart:io';

void main() {
    File myfile = new File("example.txt");
    myfile.writeAsStringSync("Hello Word!");
}
```

This demonstrates the synchronous version of the function, with `writeAsString` being the asynchronous version.

Open the `log.dart` file in this chapter's project. For this, take a look at the following code snippet:

```
void log(HttpRequest request) {
    String entry = getLogEntry(request);
    File logstore = new File("accesslog.txt");
    logstore.writeAsStringSync(entry, mode: FileMode.APPEND);
}
```

To record the web access request to the disk, the log entries will have to be accumulative so that the entries are written with the `APPEND` mode passed as a named parameter.

Extracting request information

The `HttpRequest` object and `headers` property expose metadata from the request to the client. For example, the `headers` property can give the host and port number. The `user-agent` identifies which program or web browser was used when making the request, as shown in the following code snippet:

```
String getLogEntry(HttpRequest request) {

    //USER_AGENT
    HttpHeaders headers = request.headers;
    String reqUri = "${headers.host},${headers.port}${request.uri.
    toString()}";
    String entry =
        " ${request.connectionInfo.remoteAddress.address}, $reqUri,
${headers[HttpHeaders.USER_AGENT]}\r\n";

    return (new DateTime.now()).toString() + entry;
}
```

The `remoteAddress` getter (IP address) can be used to track who was browsing and from which part of the world. Log entries to request the `index.html` blog would look as follows:

```
2015-05-17 17:03:48.496 127.0.0.1, 127.0.0.1,8080/index.html,
[Mozilla/5.0 (X11; CrOS x86_64 6812.88.0) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/42.0.2311.153 Safari/537.36]
2015-05-17 17:03:49.177 127.0.0.1, 127.0.0.1,8080/6.png, [Mozilla/5.0
(X11; CrOS x86_64 6812.88.0) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/42.0.2311.153 Safari/537.36]
2015-05-14 18:52:45.690 127.0.0.1, 127.0.0.1,8080/favicon.ico,
[NetSurf/2.9 (Linux; x86_64)]
```

Browsers will request the page that is browsed to by the user and also other assets such as `favicon.ico` (the small square graphic that is often shown in the URL bar of a web browser or on a bookmark).

A blog editor

A blog will have an administration section that will allow us to update the blog using the Web. This will be accessible through the address `http://localhost:8080/admin`, using the `admin` login as the username and `Password1` as the password (both case sensitive).



Never ever ever use such a simple login in a real application!

The admin section of the blog's website will consist of web forms. The forms will use the `post` method so that the blog server will have to detect this correctly in the `_handleRequest` method of the `BlogServerApp` class to handle the request. This is exposed by the `method` property of the `request` object, as shown in the following code snippet:

```
if (request.method == 'POST') {
    _handleFormPost(request);
}
```

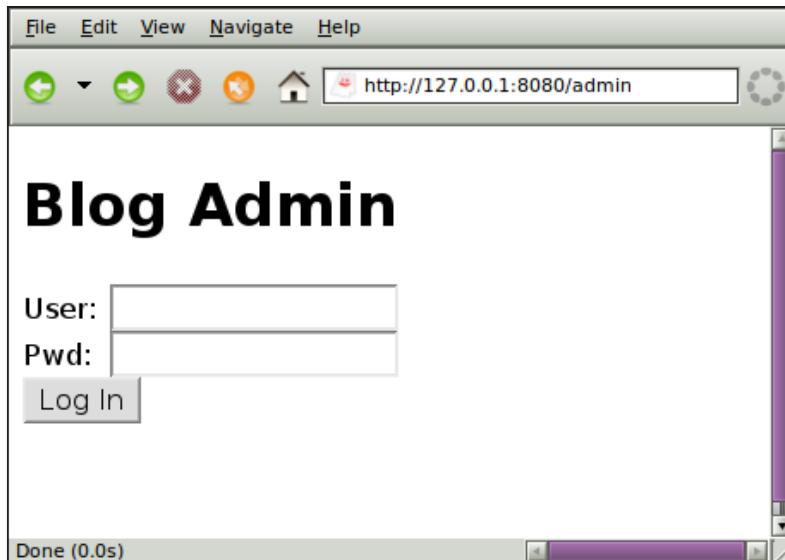
`POST` requests still have a URI associated with them, allowing a decision over how to handle the input. For detailed information, let's take a look at the following code snippet:

```
void _handleFormPost(HttpServletRequest request) {
    if (request.uri.path == "/login") _performLogin(request);
    if (request.uri.path == "/add") _performNewPost(request);
}
```

The handling of the login process and the new blog post are rather different.

Password protection

The /admin path serves the web form that allows the entry of the username and password required to access the admin features, as shown in the following screenshot:



The username and password will be verified, and if they are correct, the form to add a blog post will be shown to you. Let's take a look at the following code snippet:

```
void _performLogin(HttpServletRequest request) {
    request.listen((List<int> buffer) {
        String page = _checkAdminLogin(buffer);

        request.response
            ..statusCode = HttpStatus.OK
            ..headers.set('Content-Type', 'text/html')
            ..write(page)
            ..close();
    }, onDone: () => request.response.close());
}
```

The `checkAdminLogin` method performs the checking of the login details and returns either the web form or a message stating that the login attempt has failed.

Encryption

Security is an important feature of any administration feature of a web application. We will not want to store the password on the site in plain text! A typical way to store the password is a one-way hash, which makes the password hard to crack even if the attacker has the hash value.

The `crypto` package provides a range of encryption options. The `SHA` algorithm will be used for this purpose for the blog, as follows:

```
String _checkAdminLogin(List<int> buffer) {
    var sha = new SHA256();
    sha.add(buffer);
    var digest = sha.close();
    String hex = CryptoUtils.bytesToHex(digest);
    String page = "";

    if (hex != expectedHash) {
        page = wrongPassword;
    } else {
        page = addForm;
    }

    return page;
}
```

Conveniently, the input for the hash is a `List<int>` list, and this is exactly what is received from the web form input into the request's `listen` handler. The `SHA256` algorithm simply takes a body of data and then has the `close` method called to return the calculated hash.



SHA stands for Secure Hash Algorithm. The 256 refers to the word size. If you have ever installed a package on a Linux system, then you have used **SHA-256**. The crypto-currency Bitcoin also uses SHA for some of its operations.

As both the username and password have to be correct, the entire input from the form can simply be hashed and compared against a previously calculated value. For easy storage, the hash is converted to a hexadecimal string, as follows:

```
const String expectedHash = "bdf252c8385e7c4ae76bca280acd8d44
f85d7fdb56f1fbfb44f5749ff549ba2f6";
```

This is stored as a constant in `content.dart`.

Handling more complex forms

Of course, form handling is not always as straightforward as the login page.

Typically, form elements need to be split apart and handled separately. The served page after logging into the administration page will allow the user to add a new post to the blog. The user will supply the post title and body text and the post ID number and date will be generated by the application, as shown in the next screenshot:

The screenshot shows a web form titled "Add Blog Entry". It contains two input fields: "Title" and "Entry", both represented by rectangular input boxes. Below the "Entry" field is a large text area. At the bottom of the form is a single button labeled "Post".

Processing the form

The `_performNewPost` method calls the `getFormData` utility function to split the data into a more manageable list. This rather low-level web programming is typically wrapped up by most Dart web frameworks, as shown in the following code snippet:

```
List _getFormData(List<int> buffer) {
    var encodedData = new String.fromCharCodes(buffer);
    List pieces = encodedData.split("&");
    List data = [];
    List finalData = [];
```

```
pieces
    .forEach((dateItem) =>
        data.add(dateItem.substring(dateItem.indexOf("=") + 1)));
}

data.forEach(
    (encodedItem) => finalData.add(Uri.decodeQueryComponent(encodedItem)));
}

return finalData;
}
```

The data is first split into data from each control, and then the data is decoded using the `Uri.decodeQueryComponent` component to the raw data before it is returned to the caller.

Saving data to a disk

The data has been processed to a manageable list, so now we only need to construct a blog post and save it to the filesystem. This is carried out by the `_performNewPost` method of the `BlogServerApp` method. Let's take a look at the following code snippet:

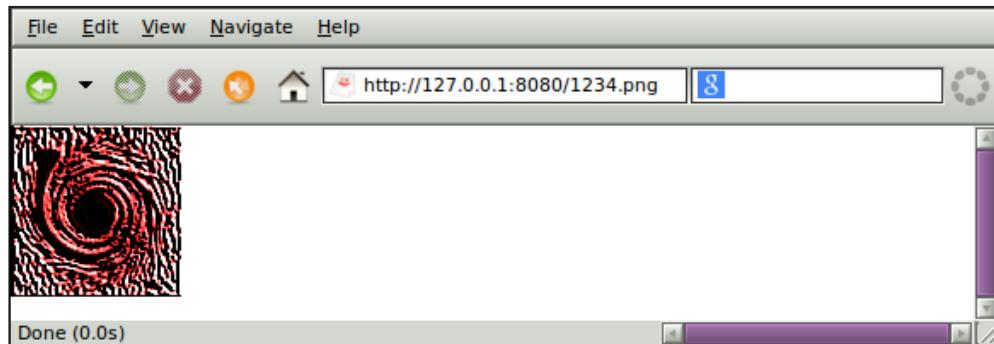
```
List formData = _getFormData(buffer);
String newID = hostedBlog.getNextPostID();
String filename = "$newID.txt";
var p = path.join("content", "posts");
p = path.join(p, filename);

File postFile = new File(p);
String post = BlogPost.createBlogPost(formData[0], formData[1]);
postFile.writeStringSync(post);
```

The `blog` object returns the next free `PostId`, which is used in the file name, and the full path is constructed using `path.join` from the `path` package. The `createBlogPost` static method on the `BlogPost` class is used to add extra data in the defined format. This is written to the constructed path using the synchronous `write` function.

Serving a default graphic

This version of the add form does not allow a user to set a graphic file, which each post requires. If the blog post ID is 8, then 8.png will be requested. Rather than not loading any file, a default image (default.png) will be loaded up. Let's take a look at the following screenshot:



If the file is not found on the filesystem (such as the fictitious 1234.png image!), then 6.png is loaded and served instead, as shown in the following code snippet:

```
File getBlogImage(int index) {
    String path;
    if (imgPaths.containsKey(index)) {
        path = imgPaths[index];
    } else {
        path = _pathDefaultImg;
    }

    return new File(path);
}
```

This is implemented by simply testing the `imgPaths` map for the requested `index` variable. If the key is not present in the `imgPaths` map, the default image path is used.

Refreshing the blog

The content of the blog is read when the blog server is started. As this content has now changed with the new blog post, this list must be refreshed using the `BlogServerApp` method called `initBlog`, as follows:

```
initBlog() async {
    _cache = {};
    IDs = new List<int>();
    postPaths = new Map<int, String>();
    imgPaths = new Map<int, String>();
    await _loadPostList();
    _loadImgList();
}
```

The `BlogServerApp` and `_performNewPost` methods perform the task of calling `initBlog` before serving the new post, as follows:

```
hostedBlog.initBlog();

req.response
..statusCode = HttpStatus.OK
..headers.set('Content-Type', 'text/html')
..redirect(new Uri(path: "post$newID.html"))
..close();
```

Once the blog is reinitialized, the application serves the new blog post to the user as a single page.

Caching

The serving of content in the first iteration of the web server was rather inefficient—I hope you noticed! On every `GET` request, the blog source file is read from the disk and is processed. To gain some efficiency, we can use a map added to the `Blog` class as a field. Let's take a look at the following code snippet for more information:

```
BlogPost getBlogPost(int index) {
    if (!_cache.containsKey(index)) {
        _cache[index] = new BlogPost(postPaths[index], index);
    }
    return _cache[index];
}
```

The map object uses the blog `index` as a key and the resultant blog post as the value. This cache is kept in the memory so that if the blog server application is stopped and restarted, new requests will not initially come from the cache.



Caching of content in web applications is quite an art. For example, a cache could be put in place at the request level for both entire pages and images. The underlying operating system will also cache files in memory, making operations much faster on subsequent requests.

As ever, a memory cache is just one tool that may or may not improve a particular application. Benchmarking a performance and finding the bottleneck that needs caching is critical for a high-performance application.

The cache is something that must be remembered when the content is refreshed and when a new blog post is added. A simple strategy is to set the `_cache` object to `{ }` (an empty map), relying on the first display of the post to fill the cache.

Watching the filesystem

It is very convenient to update a web form from any computer. It may also be useful to update it purely from the filesystem so that a standard text editor can be used or we can have the blog post content entirely automated.

The `watcher` package provides the functionality needed for this feature. This package is authored by the Dart team and is typically of the Dart philosophy of being a small focused library. The `DirectoryWatcher` class takes a path and notifies via events if any files have added, removed, or modified.

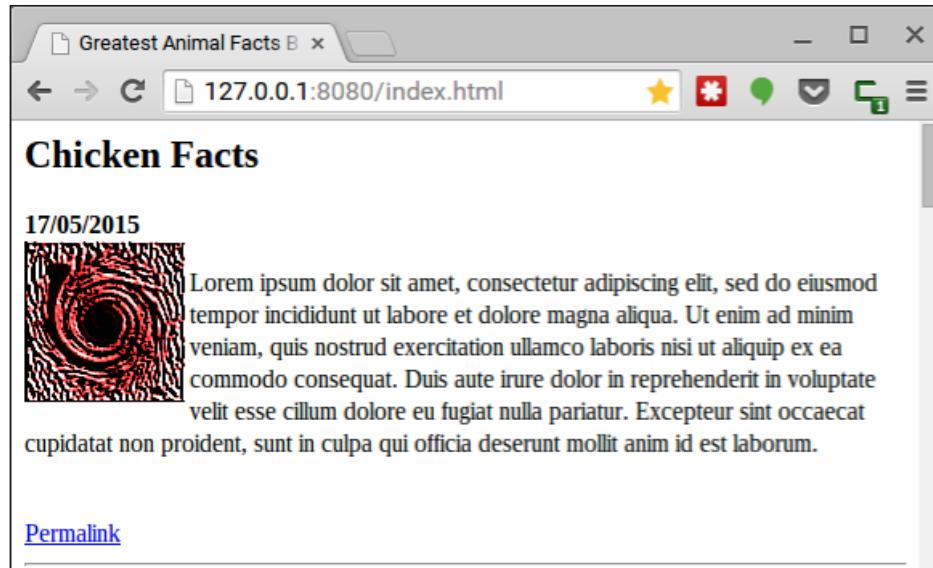
The `Blog` class constructor configures the path that is going to be watched by the `DirectoryWatcher`. Let's take a look at the following screenshot:

```
DirectoryWatcher _changeMonitor;

Blog(this._pathPosts, this._pathImg) {
    initBlog();
    _changeMonitor = new DirectoryWatcher(this._pathPosts);
    _changeMonitor.events.listen((watchEvent) => initBlog());
}
```

The actual details of the change are not important for the blog, so the `initBlog` method will be called to reset the entire site.

In the `content` folder of this chapter's sample code, there is an unpublished blog post with the `7.txt` filename. Start the `main.dart` blog server located in the `bin` folder and go to the `index.html` page. Open the file manager on your system and locate the `7.txt` file. Move this file into the `posts` folder and refresh the web page. A new blog post should appear at the top of the page, as shown in the following screenshot:



Return to the folder and move the `7.txt` file out of the posts folder. Refresh the web browser again and the most recent post on **Chicken Facts** will have gone.

XML feed generation

For blog reader's search and syndication, a blog needs to produce an RSS feed. RSS is an XML specification that lists blog posts in a specified format as separate items that are simple to process.

In late 1990s, there were many attempts to create a syndication format for the Web. Dan Libby and Ramanathan V. Guha from Netscape created the RDF site summary format in 1999 for use in the My.Netscape.Com portal. The RDF specification was refined and eventually became RSS.

The competing (and in my opinion, much superior!) feed format is Atom. Atom was developed to fix the shortcomings of RSS has not gained much traction with simple RSS and web services used for more advanced applications.

As well as blog and news publications, RSS is also established as the standard format for podcasting and its supporting clients and tools.

The full specification is available at <https://validator.w3.org/feed/docs/rss2.html>.

To produce the feed for the blog, posts can easily be iterated as we already have a list of IDs. To produce a valid XML feed, the `xml` package will be used. This is not part of the SDK and is an open source library, which has become a favorite for XML creation and processing in Dart. There was at one time an XML processing package in the SDK, but the `xml` package was improved so quickly that it made more sense to make it the standard package. Let's have a look at the following code snippet for more information:

```
String getRSSFeed() {
    var RssXb = new XmlBuilder();
    RssXb.processing('xml', 'version="1.0"'');

    RssXb.element('rss', attributes: {'version': '1.0'}, nest: () {
        RssXb.element('channel', nest: () {
            IDs.forEach((int postID) {
                BlogPost post = getBlogPost(postID);
                RssXb.element('item', nest: () {
                    RssXb.element('pubDate', nest: () {
                        RssXb.text(post._date);
                    });
                    RssXb.element('title', nest: () {
                        RssXb.text(post._title);
                    });
                    RssXb.element('link', nest: () {
                        RssXb.text("http://127.0.0.1:8080/post${post._id}.
html");
                    });
                });
            });
        });
    });
}

var xml = RssXb.build();
return xml.toXmlString(pretty: true);
}
```

Elements are added in a nested fashion here, which reflects the RSS structure. Once the data structure is put together, the `build` method constructs the XML structure, and finally, it is converted into a string.



For simple XML, it is tempting just to concatenate strings. However, an XML library can provide facilities such as character encoding, validation, and formatting, which are well worth the slightly more verbose code on the smaller use cases.

As it will be viewed by human beings as well as computers, the optional name parameter option `pretty` is set to true for a neatly formatted output, as shown in the following example:

```
<?xml version="1.0"?>
<rss version="1.0">
  <channel>
    <item>
      <pubDate>02/05/2015</pubDate>
      <title>Lemur Facts</title>
      <link>http://127.0.0.1:8080/post6.html</link>
    </item>
  ...

```



The RSS feed is served to `http://127.0.0.1:8080/feed.xml` and can be viewed directly in most web browsers.

Serving the RSS

The RSS feed can be served as any other text file, though the headers will need to be set to assist the client application. Let's take a look at the following code snippet for more information:

```
void _serveRSSFeed(HttpServletRequest request) {
  request.response
    ..statusCode = HttpStatus.OK
    ..headers.set('Content-Type', 'application/rss+xml')
    ..write(hostedBlog.getRSSfeed())
    ..close();
}
```

The content type is set to `application/rss+xml`, which triggers the XML display in the browser. Otherwise, it may attempt to display the content as HTML.

The JSON feed generation

Serving a **JSON (JavaScript Object Notation)** feed is a useful alternative to XML, as it requires the transfer of less data and is often easier and faster to deal with rather than dealing with XPath. The `getJSONFeed` method is part of the `Blog` class, shown as follows:

```
String getJSONFeed() {  
    List posts = new List();  
    IDs.forEach((int postID) {  
        BlogPost post = getBlogPost(postID);  
        Map jsonPost = {};  
        jsonPost["id"] = post._id;  
        jsonPost["date"] = post._date;  
        jsonPost["title"] = post._title;  
        jsonPost["url"] = "http://127.0.0.1:8080/post${post._id}.html";  
        posts.add(jsonPost);  
    });  
    return JSON.encode(posts);  
}
```

This method follows the same approach as RSS, with the main data structure being a list this time and the items being entries in that list. The list can be converted to JSON using the `JSON.encode` method from `dart:convert`:

```
[  
...  
{"id" : 6,  
 "date" : "02/05/2015",  
 "title" : "Lemur Facts",  
 "url" : "http://127.0.0.1:8080/post6.html"},  
...  
]
```

The JSON feed is served to `http://127.0.0.1:8080/feed.json` and can be viewed directly in most web browsers.

Serving the JSON

The serving of JSON requires not only the correct type to be specified, but also the security needs to be dealt with for the consumption of this data feed from another web domain:

```
void _serveJsonFile(HttpServletRequest request) {
    request.response
        ..statusCode = HttpStatus.OK
        ..headers.set('Content-Type', 'application/json')
        ..headers.add("Access-Control-Allow-Origin", "*")
        ..headers.add(
            "Access-Control-Allow-Methods",
            "POST,GET,DELETE,PUT,OPTIONS")
        ..write(hostedBlog.getJSONFeed())
        ..close();
}
```

The content type is set to `application/json`, which triggers the XML display in the browser. The access control (CORS) is set to allow any origin and a set of HTTP verbs.

 **Cross-origin resource sharing (CORS)** is a technology that permits restricted resources, such as scripts to be requested from other web domains. For example, a news site wants to request the headlines in the JSON format from a blog to display them.

By default, resources can only be loaded from the same domain. CORS allows specific resources, and actions (HTTP verbs) on those resources such as GET, POST, and so on to be shared. So, we may be happy to share our headlines, but not our login validation scripts.

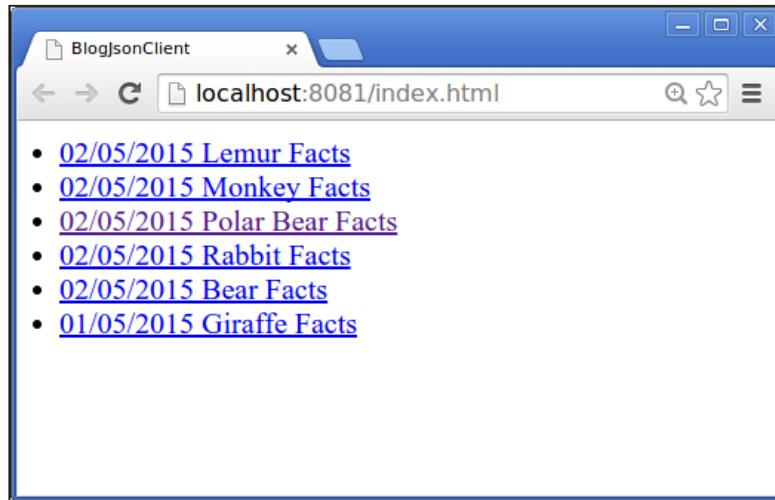
Consuming the JSON feed

JSON has certainly established itself as the modern web data format, being lightweight to both produce and consume. To explore the JSON feature from end to end, we will detour briefly from the server-side and return to the client-side Dart in order to exercise the blog's JSON data feed. For this, take a look at the following code snippet:

Open the example project `BlogJsonClient` of this chapter. This a simple web project which acts as a client to the Blog server JSON feed. For this client to operate, the blog server must also be running.

```
void main() {
    var jsonSrc = "http://127.0.0.1:8080/feed.json";
    HttpRequest.getString(jsonSrc).then((String data) {
        List decoded = JSON.decode(data);
        decoded.forEach((post) {
           querySelector('#output').children.add(new LIElement()
                ..append(new AnchorElement()
                    ..href = post['url']
                    ..text = "${post['date']} ${post['title']}"));
        });
    });
}
```

The consumer of the JSON feed does not need to be aware of the CORS settings in any way. It simply makes an HTTP request to the appropriate URL to receive the JSON data or other resource. The request must meet the CORS requirements, such as the origin or HTTP verb to receive the desired response. Let's take a look at the following screenshot:



Here, the `HttpRequest` class has a static method to return a remote resource as `Future<String>`, which is handled asynchronously in the `then` clause.

JSON is then iterated and the `LI` elements (**List Items**) are added to the DOM structure of the web page.

Static generation

Web hosting facilities vary greatly from provider to provider. The ability to run programs and install software is not universally available. Static generation of websites has become more popular with the ability to run small simple virtual machines on the cloud.

The advantages are very clear, there is no maintenance on software or a database to maintain, making the system very secure and also very fast to serve. It is not suitable for all applications, though. Blogging is one where static generation may make sense.

Freezing the website

In this chapter's project, there is a `staticgen.dart` file in the `bin` folder. This is a standalone program that generates a static HTML file of the front page of the blog. Note that the `main` function is marked with the `async` keyword and is missing the `void` keyword.

Introducing the `await` and `async` keywords

The `await` and `async` keywords were introduced in Dart 1.9.4 in order to simplify asynchronous operations. Consider the following call to a method that returns `Future`:

```
obj.method().then(handlerFunction)
```

This is fine for the post part, but what if things get more complicated and `handlerFunction` returns a future too?

```
obj.method().then(handlerFunction).then(handlerFunction2);
```

Things are starting to get complicated already – debugging is not straightforward. Ideally, we would want to deal with one part of the chain at a time and hold up the execution of statements until a desired operation is complete. This is what `await` allows:

```
var f1 = await obj.aMethod();
var result = await f1.aMethod();
```

Functions and methods to be called with `await` return `Future`. They must also be declared as `async` in the method's header, as does the function using the `await` call:

```
class Foo{
  Future<int> aMethod() async {
    return await aFunction();
  }
}
```

The `async` and `await` keywords simplify the code, particularly if in-line functions are being used, and they also make the asynchronous code much more readable.

Joining file paths

Most computer users have hit the issue with forward and backward slashes being used on paths in Unix-style (`/home/davymitchell/`) and Windows operating systems (`c:\users\davymitchell\`):

```
var contPath = path.join(Directory.current.path, "content");
var srcPath = path.join(contPath, "posts");
var imgPath = path.join(contPath, "img");
```

The `path` package provides a cross-platform method `join` to join folder and file names.

Creating an output folder

The output files will be placed in a folder named `staticoutput` on the current path. The existence of this folder will be tested, and if it is not present, the folder will be created. Let's take a look at the following code snippet:

```
main() async {
    var contPath = path.join(Directory.current.path, "content");
    var srcPath = path.join(contPath, "posts");
    var imgPath = path.join(contPath, "img");

    Blog staticBlog = new Blog(srcPath, imgPath);
    await staticBlog.initBlog();
    print(Directory.current);

    // Create output directory if it does not exist.
    var outDir = new Directory('staticoutput');
    var staticPath = path.join(Directory.current.path, 'staticoutput');
    if (!await outDir.exists()) {
        outDir.create();
    }
}
```

```
// Write out main page.  
var outPath = path.join(staticPath, "index.html");  
var pageContent = staticBlog.getFrontPage();  
File indexPage = new File(outPath);  
await indexPage.writeAsString(pageContent, mode: FileMode.WRITE);  
  
// Return success exit code.  
exit(0);  
}
```

The outcome of the asynchronous method exists and is waited upon before the program continues.

Generating the front page

In *Chapter 5, A Blog Server*, there is a version of Blog Server called `initBlog` from the constructor. It has been moved to a post object creation initialization call so that `await` can be used. Methods and functions marked as `async` cannot be called from constructors. The use of `async` simplifies the `_loadPostList` method, as there is much less nesting, as shown in the following code snippet:

```
_loadPostList() async {  
    print("Loading from $_pathPosts");  
    Directory blogContent = new Directory(_pathPosts);  
  
    var postsSrc = await blogContent.list();  
  
    await postsSrc.forEach((File f) {  
        String postFilename = path.basenameWithoutExtension(f.path);  
        int id = int.parse(postFilename);  
        IDs.add(id);  
        postPaths[id] = f.path;  
    });  
  
    IDs.sort();  
    IDs = IDs.reversed.toList();  
}
```

Previously, the posts were loaded and the blog was set up asynchronously.



If you are looking for some good blogs on Dart programming, a good place to start is <http://www.dartosphere.org/>, which is a blog aggregator or planet-style website with news and resources from various sources.

Writing the static version

Now that the page content is available at a determinable time, it can be written out on a disk using the `writeAsString` method. The `FileMode.WRITE` permission will overwrite any existing file, hence is suitable for updates. Let's take a look at the following code snippet:

```
var outPath = path.join(staticPath, "index.html");
var pageContent = staticBlog.getFrontPage();
File indexPage = new File(outPath);
indexPage.writeAsStringSync(pageContent, mode:FileMode.WRITE);

exit(0);
```

The last line returns an exit code to the OS when the program completes, with 0 being the convention for success. This function does not wait for any pending operations, closing the program immediately.

Load testing revisited

The initial load testing application was rather limited, and though of use, it did not give a realistic picture with so many requests thrown at once at the server application without waiting for a response.

Updating the load tester

The new version of the load testing application will make a single HTTP call and await the result before calling the next. This takes place in the `main.dart` source file. Note that the `main` function itself is now marked as `async`. The `await` command is used in the calling loop of the `main` function, as follows:

```
main() async {
  print("Starting...");

  var url = "http://127.0.0.1:8080/index.html";
  var hc = new HttpClient();
  var watch = new Stopwatch();
  int attemptedRequests = 200;
```

```
print("Starting testing...");  
watch.start();  
  
for (int i = 0; i < attemptedRequests; i++) {  
    await callWebPage(hc, url, i, watch);  
}  
  
watch.stop();  
print("${watch.elapsed.inMilliseconds}");  
}
```

The `callWebPage` method needs to be marked as `async` too, as `await` will be used twice:

```
callWebPage(HttpClient webClient, String targetURL, int requestNumber,  
           Stopwatch watch) async {  
    HttpClientRequest request;  
    HttpClientResponse response;  
    request = await webClient.getUrl(Uri.parse(targetURL));  
    response = await request.close();  
    print("$requestNumber, ${response.statusCode}, ${watch.elapsed.  
inMilliseconds}");  
}
```

The two operations of unknown duration, the URL fetch and closing of the response, are waited upon before the output to the screen is processed.

Summary

The server side opens up many facilities of the filesystem to save the content directly from the application, supplying content for web servers and metadata, such as log files.

Dart has the facilities to create, serve, and consume the modern web formats in the web client and on the server. We saw how quickly a specialized web server can be put together. You are now very familiar with using Dart packages outside the core SDK to implement vital application features.

The asynchronous facilities allow responsive applications to be written, and `await/async` helps us write clear code and bring asynchronous tasks together so that we operate them in a synchronous manner.

Our next, and largest, project will be end-to-end Dart. In this chapter, we explored how to build on the server aspects. Instead of the local filesystem, we will reach out to databases and web services to build a real-time display.

7

Live Data Collection

"It is a capital mistake to theorize before one has data."

- Sherlock Holmes

Most people in the IT world are familiar with the phrase **Garbage In, Garbage Out (GIGO)**. If bad data goes into a system, the output will be wrong too. A source of good data should always be used, and systems need to be robust enough to handle users or other systems' 'garbage'. Specifications and standards can help, but there will always be exceptions and gray areas.

One of the most baffling statements I have heard in my career was from a customer with a software data input issue: 'we have our own ISO standard'!

Kicking off the earthquake monitoring system

The next series of chapters in the book will be built around a single project from several different perspectives. A real-world data source with earthquake information about the planet we live on will be tracked, recorded and displayed.

It is quite ambitious and there will be a lot of Dart along the way. It is surprising how much activity there is in the Earth's crust—perhaps you could blog or write a presentation on seismology using the output of our projects so far!

The project for this section is called `QuakeMonitorFS`—make sure that you select the project with the name ending FS and not DB. The FS version saves to the filesystem while the DB version saves to the database. We will first look at the filesystem storage version, and then move on to the database version.

Introducing the data source

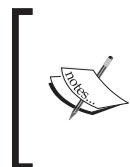
The data source for the application is a web service provided by the **United States Geological Survey (USGS)** Earthquake Hazards Program. There is a wealth of information on their website <http://earthquake.usgs.gov/> including some continually updated data feeds from all around the world.

The data feeds that are of most interest for Dart programmers are in a particular JSON format.

Exploring the GeoJSON format

GeoJSON is a format for recording geographic data structures, which includes support of geometric definitions, such as points and polygons. For example, the location of an earthquake's epicenter (point) and the area affected (polygon). Part of the GeoJSON can look as follows:

```
{  
  "type": "Point",  
  "coordinates": [122.3818333, 45.0686667, 12.9]  
}  
,  
  "id": "uw61022191"
```



The previous example is a small excerpt from the feed that reports all earthquakes in the past hour (http://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_hour.geojson). Take a moment to look at this feed and get a feel for the type of data this format contains.

To kick off this project, we will create a data-monitoring application that will record information from this feed. As this program will be running over a long period, we will want to add a logging feature to ensure it is running smoothly.

Fetching and recording the data

The main function in `bin/main.dart` will prompt the data to be downloaded each minute using an instance of the `DataMonitor` class. There are 1440 minutes in a day and the data files average at around 2k, so 3 megabytes of disk space a day is practical:

```
main() async {  
  setupLogging();  
  
  DataMonitor quakeMon = new DataMonitor();
```

```

Duration updateInterval = new Duration(seconds: 60);

new Timer.periodic(updateInterval, quakeMon.fetchData);
}

```

The main function is marked `async`, which gives you a clue that the implementation will make use of the `await` feature. The call to `setupLogging` will be covered in the next section of this chapter.

The implementation of `DataMonitor` is in `lib/quakemonitorfs.dart`, which contains the key method `fetchData`:

```

fetchData(Timer timerFetch) async {
    try {
        var outDir = new Directory('data');
        if (!await outDir.exists()) {
            outDir.create();
        }

        log.info("Calling web service...");
        HttpClientRequest req = await geoClient.getUrl(Uri.
parse(dataUrl));
        HttpClientResponse resp = await req.close();
        String latestFilePath = path.join('data', latestFilename);
        await resp.pipe(new File(latestFilePath).openWrite());
        String fileContents = await new File(latestFilePath).
readAsString();

        var dataset = await JSON.decode(fileContents);

        int newId = int.parse(dataset['metadata']['generated'].
toString());

        if (idPreviousFetch != newId) {
            idPreviousFetch = newId;
            var f = new File(path.join('data', '${newId}.txt'));
            await f.writeAsString(dataset.toString());
            log.fine("Saved $newId - ${dataset.toString()}");
        }
    } catch (exception, stacktrace) {
        log.severe("Exception fetching JSON.", exception, stacktrace);
    }
}

```

Nearly every other line has an `await!` The program only has to process information every minute or so, therefore it does not require a responsive structure and a more linear approach is sufficient, though the code is still asynchronous.

The remote resource is processed first and the response closed off. The JSON string returned can be put straight into a file. Once this is completed, the `fileContents` variable is populated with the data from the file, which is then parsed by the JSON to return an object into the dataset.

The `Id` of the response is checked so that the same response is not saved twice. Note that the generated `Id` may be different, while the content is the same. The feed is more of a snapshot than a steady stream.

The data is saved off to the data folder using the id of the fetch as a text (`.txt`) file. To give some feedback, the id and JSON are written to the log file via the call to `qlog.fine`.

Logging

I reached many points of difficulty in my early software projects where the only thing left to do was to resort to logging, to figure out what was going on. Soon I grew to enjoy logging and made it a key point when starting a new project. It is an extra pair of eyes to help you test—especially when you are not looking, such as when running a data collector continuously.

The Dart package `logging` provides facilities for logging, however it does not do anything useful with the output, or to put it another way, the handling of the storage of the logging is entirely in the hands of the application.

A simple example of logging

Open up this chapter's `Logging` folder, which is a standalone example containing a range of logging:

```
Logger log = new Logger('DataMonitor');

Logger.root.level = Level.ALL;
Logger.root.onRecord.listen((LogRecord rec) {
    String logMessage = '${rec.level.name}\n${rec.time}\n${rec.message}';
    String exceptionMessage = '';
    if (rec.error != null) {
        print("$logMessage\n${rec.error.message}");
        print(exceptionMessage);
```

```
    print(rec.stackTrace) ;
} else {
    print("$logMessage") ;
}
}) ;
```

The `Logger` object is initialized and the `level` property set to `Level.ALL`. Setting this property gives control over the detail of logging occurring. The custom handler for an incoming log message simply prints it to standard output. Other possibilities are writing to disk, database, or a network socket:

```
log.info("This is my first logging program in Dart");
log.fine("Level 1 detail.");
log.finer("Level 2 detail.");
log.finest("Level 3 detail.");
log.shout("Something bad.");
```

The `Logger` class has numerous methods for logging out at different levels, and this also helps produce readable code. By having levels of logging, this can assist with analysis of the logs file. For example, if an application's logs are being looked at to get a general overview of what is going on, the `info` level of logging would be examined. If a low-level bug is occurring, the approach would be to check the logging down to the `finest` level. Each method also allows the passing of an `Exception` and a stack trace:

```
try {
    throw new Exception("We have a problem!");
} catch (exception, stackTrace) {
    log.severe("Something really bad.", exception, stackTrace);
}
```

The name passed to the `Logger` constructor, in this case `DataMonitor`, identifies a unique object. Any other scope can declare a `Logger` object with the same name, and the same instance will be returned.

Data monitor logging

For the `QuakeMonitorFS`, the application launch and the calls to the web service will be recorded. Network connections are not perfect, so we will want to know if the program has not been able to access the web service. Lets have a look at the following code snippet:

```
Logger log;

setupLogging() {
```

```
log = new Logger('DataMonitor');
Logger.root.level = Level.ALL;
Logger.root.onRecord.listen((LogRecord rec) {
    var entry = '${rec.level.name}\t${rec.time}\t\t${rec.message}\n';
    File logStore = new File("datamonlog.txt");
    logStore.writeAsStringSync(entry, mode: FileMode.APPEND);
});
log.info('Earthquake Monitor - Data fetcher. Starting...');
```

The logging is set up in `bin/main.dart` and the output appended to a text file. Note that this is not saved to the data sub-folder, as it may get lost with all the data files.

In the `fetchData` method of the `DataMonitor` class, the `Logger` object is set up in the constructor, and key events will be noted. The different levels of logging are used (info, fine, severe, and so on), which would allow an operator of the program to adjust the level of logging to the desired level.

Saving to the database

The data collector application currently writes to the filesystem. While this is useful for archiving, it makes it difficult and slow to access and query any sort of analysis or display.

If you are not experienced with databases, here is a very rapid introduction! Databases are collections of data in a structured form. These are accessed and managed via a database management system. You may have heard of **SQL Server**, **MySQL**, **Hadoop** or **Oracle**.

Databases are typically accessed using a language called SQL to read and write records from the database. Data is usually organized into tables, and each table is made up of records. Each table has a set columns defined as data types – it is similar to variables in a programming language. SQL is strong on sorting, grouping, and working with sets of data.

SQL can be stored in the database in a number of forms, with the most common being a stored procedure that can add/ remove and update data, and perform some processing.

Connecting to a database almost always requires a login and may be on a remote computer. This is sometimes referred to as a connection string. It behaves much like a network connection.



Installing a database system

There are numerous database systems available and you probably already have your favorite one! For the purposes of this book, the database used will be **PostgreSQL** as it is free, open source, and available for all the major operating systems. Also it is one of the easiest databases to install, so we can spend more time writing Dart.

PostgreSQL is a powerful, open-source, object-relational database system with a 15-year heritage behind it.

It is a first-class product and is used by big names in industry and government, such as Yahoo and the United Nations. It complies strongly with SQL language standards and can cope with database tables up to a maximum size of 32 terabytes.

PostgreSQL can be downloaded from <http://www.postgresql.org/> for Mac, Linux, and Windows, and is available via most Linux distribution package managers. Also on the PostgreSQL download page, there are numerous virtual machine images available that just need to be downloaded and turned on.

The examples for this book assume that the database installation is the same machine that the software is run from. Adjusting the connection string will allow connection to remote machines.

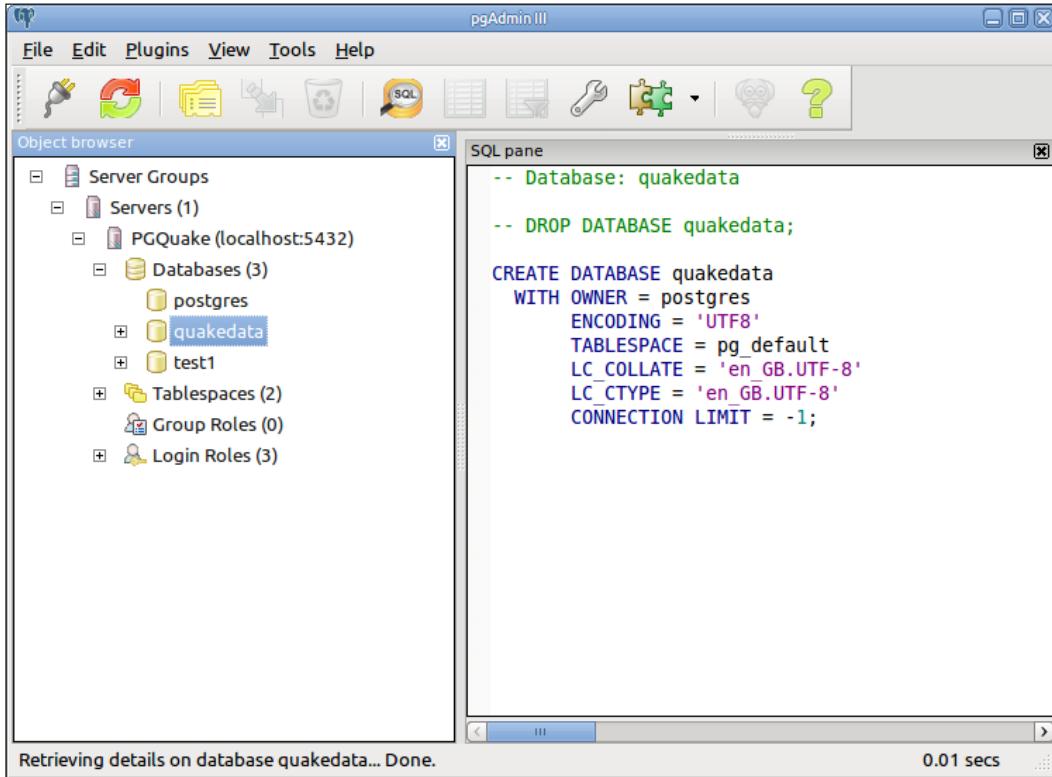
Using PostgreSQL from Dart

The package `postgresql` provides a very capable PostgreSQL database driver to use from the Dart program. It is a third-party package that is well regarded, and has had contributions from the Dart team.

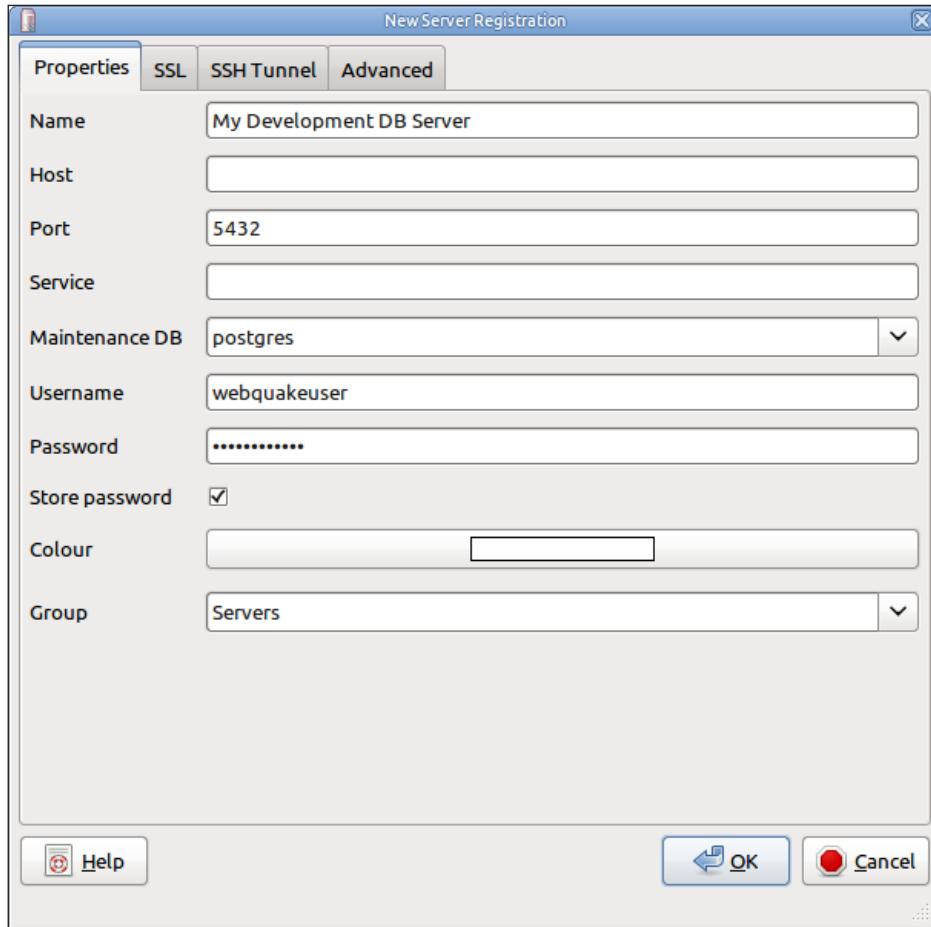
The project for this section is called `QuakeMonitorDB` and the fetching is identical to the filesystem version.

Introducing the pgAdmin GUI

For a graphical tool to access the database, use pgAdmin (also available for Mac, Windows and Linux) <http://www.pgadmin.org/download/>

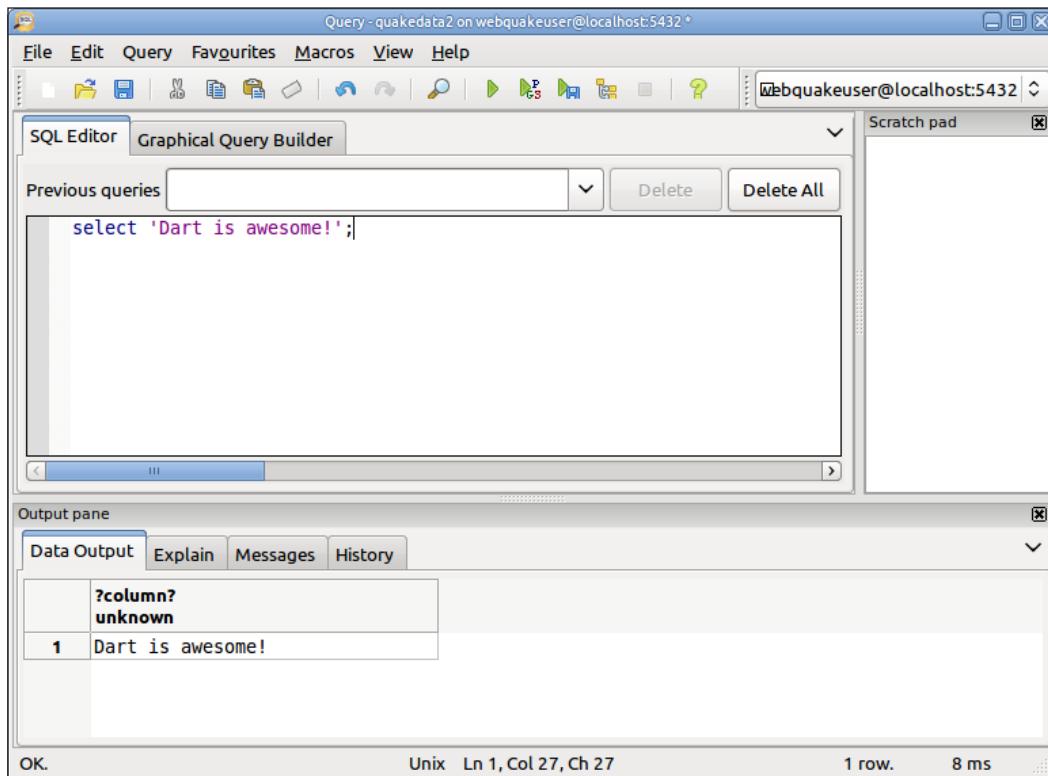


The **File** menu has the option **Add Server...** where a connection to the database server can be made.



Live Data Collection

Once a connection is available, it can be selected in the tree in the pane on the left. To issue commands to the database, select it from the database list for that server, and choose **Query tool...** from the **Tools** menu.



The query tool allows SQL queries to be run against the connected database, and displays the output and results. Enter a query, such as `select 'Dart is awesome!';`, and then click on **Execute** in the **Query** menu.

If you prefer, PostgreSQL also has a powerful built-in command line client called `psql`.

Creating the database and login

The database for the application will be called `quakedata`. The login name used will be `webquakeuser` and the password will be `Coco99nut` (case sensitive and without quotes):

```
CREATE DATABASE quakedata
    WITH OWNER = postgres
        ENCODING = 'UTF8'
        TABLESPACE = pg_default
        LC_COLLATE = 'en_GB.UTF-8'
        LC_CTYPE = 'en_GB.UTF-8'
        CONNECTION LIMIT = -1;
```

The SQL script for creating the database is called `CreateDatabase.sql` and can be found in the SQL sub-folder of the sample code for this chapter:

```
CREATE ROLE webquakeuser LOGIN
    ENCRYPTED PASSWORD 'md504a29d543c1492922e76af320cae3190'
    SUPERUSER INHERIT CREATEDB NOCREATEROLE NOREPLICATION;
```

The SQL script for creating the user is called `CreateUser.sql`.

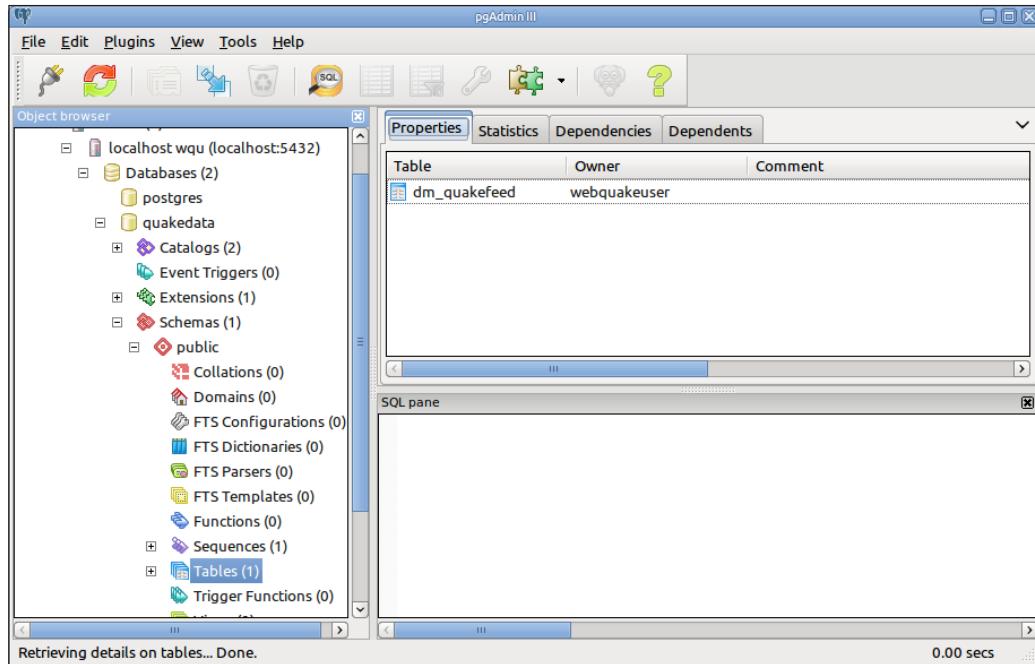
Defining the table

The main table will be a simple storage of the JSON field, with two fields of metadata, and the SQL to create it can be found in `CreateTable.sql`:

```
CREATE TABLE dm_quakefeed (
    quake_id serial PRIMARY KEY,
    geojson text NOT NULL,
    modified_date TIMESTAMP default CURRENT_TIMESTAMP
);
```

Live Data Collection

This SQL defines a table with three fields. The `quake_id` field will be a unique identifier for each record, `geojson` will store the data retrieved from the service, and `modified_date` will record when the insertion into the database was made. Lets have a look at the following screenshot:



The empty database is now configured and ready to be connected to Dart in order to save the data. The table can be found in the tree in the left-hand side panel `quakedata | Schemas | public | Tables`.

Inserting data

The first step is to create a connection string to connect to the PostgreSQL server. The class `DaoQuake` in `lib/quakedao.dart` handles the database access. This is called by the revised `DataMonitor` class's `fetchData` method, as shown here:

```
if (idPreviousFetch != newId) {  
    idPreviousFetch = newId;  
  
    //Save to database.  
    await daoGeoJson.storeJson(dataset.toString());  
  
    log.fine("Saved to db $newId - ${dataset.toString()}");  
}
```

The `daoGeoJson` instance handles the database interaction. The rest of the data fetching, such as accessing the web service is identical to the file-based version:

```
storeJson(String jsonString) async {
    var dbConn;
    try {
        dbConn = await connect(uri);
        await dbConn.execute(
            'insert into dm_quakefeed (geojson) values (@geojson)', {
                'geojson': jsonString
            });
    } catch (exception, stacktrace) {
        log.severe("Exception storing JSON.", exception, stacktrace);
        print(exception);
        print(stacktrace);
    } finally {
        dbConn.close();
    }
}
```

The first thing to take note of is that the implementation is wrapped in `try{}`, `catch{}` and `finally{}`. This is because we want a robust solution, even on error conditions. Databases fail and connections fail so the insert may not take place, so we catch exceptions. Once the `dbConn` object is connected to the database server, it runs the SQL statement to store the `jsonString`.

The application needs to manage system resources too, so we always want to close the database connections, which like memory and disk space, are finite resources. The `finally` clause is always executed after the `try`, `catch` and `except` blocks are complete:

```
String uri = 'postgres://webquakeuser:Coco99nut@localhost:5432/
quakedata';
```

The connection to the database is defined in the `uri` connection string, which identifies the server, user name, password, and database. Typically this would be stored in a configuration file.

Running the program

Once the program has been run for a few minutes, data will start to build up in the database. This can be viewed in pgAdmin in the SQL editor, which can be run by clicking the SQL icon on the toolbar:

```
select * from dm_quakefeed;
```

This simple select statement will show the table contents:

The screenshot shows a software interface with a title bar "Output pane". Below it is a tab bar with four tabs: "Data Output" (which is selected), "Explain", "Messages", and "History". The main area is a table with the following data:

	geojson text	modified_date date	quake_id integer
1	{type: FeatureCollection, metadata: {generated: 14326618570}}	2015-05-26	18
2	{type: FeatureCollection, metadata: {generated: 14326619430}}	2015-05-26	19
3	{type: FeatureCollection, metadata: {generated: 14326619910}}	2015-05-26	20
4	{type: FeatureCollection, metadata: {generated: 14326620980}}	2015-05-26	21
5	{type: FeatureCollection, metadata: {generated: 14326621580}}	2015-05-26	22
6	{type: FeatureCollection, metadata: {generated: 14326621810}}	2015-05-26	23

Maintaining a database

Setting up a database and writing the accompanying application is only a small part of running a database. Like a garden, they need tending and pruning to get the best results.

We will take a look at writing a maintenance script for the data-monitoring application that will report on the data and clear out the data.

Open the `bin/maintenance.dart` file in the editor.

Managing command line arguments

To switch between these tasks, a command line argument will be used. The Dart `main` function receives a list of Strings as its arguments:

```
import 'package:QuakeMonitorDB/quakedao.dart';

main(List<String> arguments) {
    if (arguments.length == 0 || arguments.length > 1) {
        print("Please use either -info or -delete.");
        return 0;
    }

    if (arguments[0] == "-info") {
        var daoGeoJson = new DaoQuake();
        daoGeoJson.displayInfo();
    } else if (arguments[0] == "-delete") {
        var daoGeoJson = new DaoQuake();
        daoGeoJson.deleteRecords();
```

```
    }

    return 0;
}
```

The program skeleton uses the incoming argument to take the appropriate branch. If too few or two many arguments are passed, a simple message is presented before ending the program.

Retrieving data

The `-info` option will return the following data from the database:

Value	Description
Count	Number of records in the table
MinDate	Earliest date
MaxDate	Latest date

The method is implemented in `lib/quakedao.dart` with a short SQL query to retrieve the summarized data:

```
displayInfo() async {
  var dbConn;
  try {
    dbConn = await connect(uri);

    var query = """select count(*) as Count,
      min(modified_date) as MinDate,
      max(modified_date) as MaxDate
      from dm_quakefeed
    """;
    var results = await dbConn.query(query).toList();
    print("Count : ${results[0][0]}");
    print("MinDate : ${results[0][1]}");
    print("MaxDate : ${results[0][2]}");
  } catch (exception, stacktrace) {
    log.severe("Exception getting info.", exception, stacktrace);
    print(exception);
  } finally {
    dbConn.close();
  }
}
```

The result set is returned from PostgreSQL and converted to a list. Each record returned is a list of the column values for that record:

```
Count    : 269
MinDate : 2015-05-26 00:00:00.000
MaxDate : 2015-05-27 00:00:00.000
```

In this instance, we are returned a single record (`result [0]`) containing a list with the three columns (accessed through indices) that contain the information that we need.

Deleting data

This very severe, yet useful, operation is rather simple to implement:

```
deleteRecords() async {
  var dbConn;
  try {
    dbConn = await connect(uri);

    var query = "delete from dm_quakefeed";
    await dbConn.execute(query);
  } catch (exception, stacktrace) {
    log.severe("Exception getting info.", exception, stacktrace);
    print(exception);
  } finally {
    dbConn.close();
  }
}
```

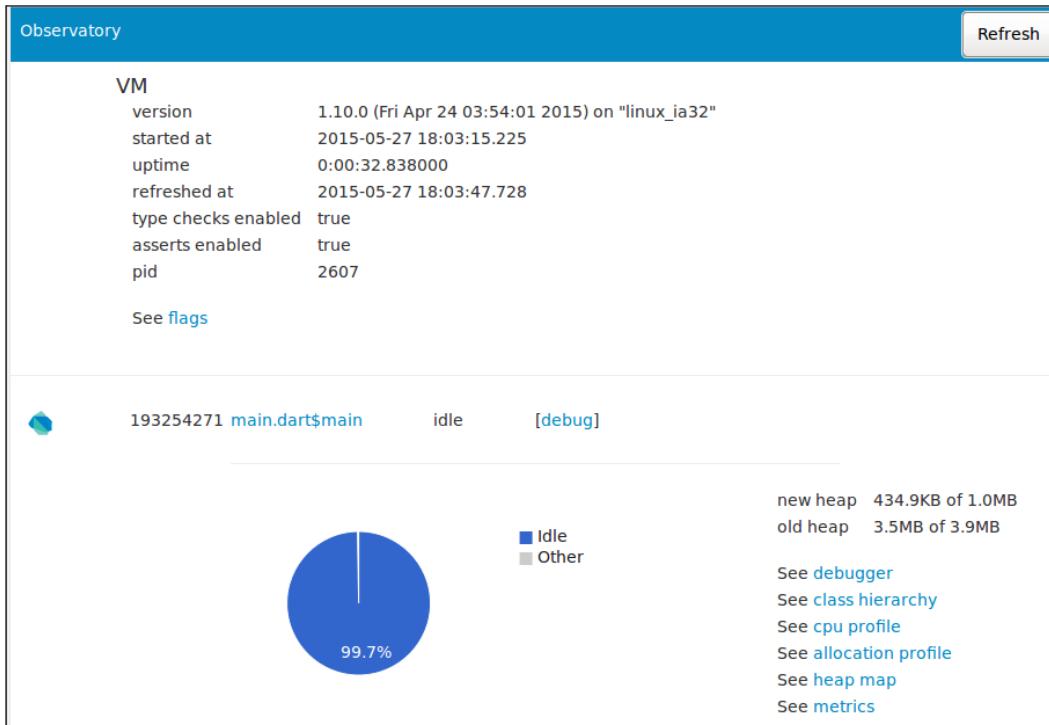
If the maintenance program is re-run with the `-info` option used after this is run, the results are as shown, with `null` meaning that no data is available:

```
Count    : 0
MinDate : null
MaxDate : null
```

Remember to use this very, very carefully!

Observing the Dart VM internals

If you have been watching the output window keenly as you have launched Dart projects, you will have noticed references to the Observatory. This tool (which is part of the Dart SDK) allows developers to look inside a running Dart virtual machine, and profile the ongoing activities. This is useful to find out exactly what an application is doing and where bottlenecks exist.



The Observatory supports command line applications and web applications (Dart versions and not the compiled to JavaScript output) running in Dartium. To enable it, simply pass one of the many Observatory command line parameters—see the SDK documents (<https://www.dartlang.org/tools/dart-vm/>) for a full listing:

```
dart --enable-vm-service bin/main.dart
```

Launch the `main.dart` program with the Observatory attached, and browse to the address `localhost:8181` (note that this port number can change though it can be specified in the launch command parameter `--enable-vm-service=<port>/<IP address>`):

```
chromium.exe index.html
```

Launch `index.html` with the Observatory attached. The address to put into the web browser will be shown:

```
Observatory listening on http://127.0.0.1:34935
```

The URLs can be opened in any web browser, the most important feature is the Refresh button, that appears and re-samples the data that is being displayed.

While you are waiting for the data monitor to collect data, take some time to explore the inner workings of the Dart VM as it runs the code.

Unit testing

If you are a Test Driven Development advocate you are probably a bit concerned that we have not written any unit tests so far. That is about to change as we introduce the Dart unit test package, which is simply called `test` (you may see older example code online that uses the older `unittest` package).

In the sample code for this chapter, there is a sub-folder called `Unittest` that contains a unit test in `bin/main.dart`:

```
library Unittestdemo.test;

import 'package:test/test.dart';

void main() {

  test('HelloWorldTest', () {
    expect(1+1, 2);
  });
}
```

This defines a new test called `HelloWorldTest` and the actual test can be carried out by the `test` in function. The result of the test can be validated using the `expect` method. The library contains an extensive range of matchers to check a result, for example, `isResult`, `isList`, `isNull`, `isTrue` and `isNonPositive`.

Unit Testing focuses on testing a small part of an application at a time. Tests are defined in code, so that they can be run easily and quickly. To ensure they run quickly, true unit tests should not use external resources such as the network or files.



For the code behind an application to be easily unit tested, the design has to allow for the objects to be tested in isolation. For example, all IO may be wrapped into a simple class so that a unit test can be written that uses a fake, or mock, version of the IO class.

This impacts the design but makes sure the code is tested all the way through development. Tests can even be written before the development starts.

Running unit tests

Unit tests are run like any other Dart command line program from your chosen IDE/Editor or from the command line. For instance, from the `Unittest` folder the test can be run as `dart bin/main.dart`:

```
00:00 +0: HelloWorldTest
00:00 +1: All tests passed!
```

The default output for the test runner includes color characters (which may not be supported in your IDE or shell):

```
00:00 +0: HelloWorldTest
00:00 +0 -1: HelloWorldTest
  Expected: <33>
  Actual: <2>

  package:test      expect
bin/main.dart 9:5  main.<fn>

00:00 +0 -1: Some tests failed.

Unhandled exception:
Uncaught Error: Dummy exception to set exit code.
#0      _rootHandleUncaughtError.<anonymous closure> (dart:async/zone.dart:886)
#1      _asyncRunCallbackLoop (dart:async/schedule_microtask.dart:41)
#2      _asyncRunCallback (dart:async/schedule_microtask.dart:48)
#3      _runPendingImmediateCallback (dart:isolate-patch/isolate_patch.dart:96)
#4      _Timer._runTimers (dart:isolate-patch/timer_impl.dart:392)
#5      _Timer._handleMessage (dart:isolate-patch/timer_impl.dart:411)
```

The output shown gives the names of the test run and the overall result:

```
00:00 +0: HelloWorldTest
00:00 +0 -1: HelloWorldTest
  Expected: <33>
    Actual: <2>

  package:test      expect
  bin/main.dart 9:5  main.<fn>

00:00 +0 -1: Some tests failed.
```

Of course, it doesn't always go well and detail is needed to go and investigate any issue. As well as showing the failing test detail, an exception is thrown so the process running the tests will exit with an error.

Writing unit tests for the data monitor

To get a head start on the next part of the project, we will put together some tests for a class to deal with the contents of the JSON. This is in the `test` folder of the `QuakeMonitorDB` project. The Dart Analyzer will show some issues with these files:

```
Warning:(16, 15) Undefined class 'GeoUpdate'
```

This is because we are writing our tests first, in true **TDD (Test Driven Development)**.

The file `test/qmdb_test.dart` contains the test cases, and `test/data.dart` has some sample data for our tests. Having a fixed set of data is typical for unit testing, as it gives predictable results for the test and does not rely on external resources:

```
test('Object create.', () {
  var o = new GeoUpdate();
  expect(o, isNotNull);
});
```

The first test is a simple creation of the object with no parameters. It is useful to break tests down to this level, as when future changes to the code makes the tests fail, it will be easier to identify at what point the failure occurred.

The sample data for testing is declared as `const`.
 What's the difference between `final` and `const`?
 That's a good question!



A `final` variable is single-assignment and is required to have an initializer. No further changes can be made.

A `const` object is frozen and completely immutable at compile time.

The keywords `final` and `const` provide valuable hints to the compiler about what a variable is, and how it will be used, which can aid performance.

Grouping tests together

It is always good to be organized, and as we may end up with hundreds, if not thousands of unit tests in a large project, they can be grouped together:

```
group('GoodJSON', () {
  var o;
  setUp(() {
    o = new GeoUpdate(sampleJson);
  });
  test('T1. Simple load.', () => expect(o.src, equals(sampleJson)));
  test('T2. Blank load', () {
    o = new GeoUpdate("");
  });
});
```

To run tests, there is sometimes a common set of steps to initialize test objects. The test package provides this facility via the `setUp` function, which is run before the tests in the group.



There are many more convenient functions and features in the Dart test package, which is constantly being improved. You can follow along and contribute to its development on GitHub.

<https://github.com/dart-lang/test/>

The final group of tests focus on `Features`, which is the name used for the main earthquake entries in the geoJSON feed:

```
group('Features', () {
  var o;
  setUp(() {
```

```
    o = new GeoUpdate(sampleJson);
  });
test('Test 1', () => expect(o.features.length, equals(5)));
test('Test 2', () => expect(1, equals(1)));
});
```

Developing tests and the implementation is usually an iterative affair during the development of a project. As the second test in this groups shows, it is easy to put in a placeholder test. Tests help ensure that the final implementation of the code has an upfront design, is loosely coupled, and also provides an easy way to automatically retest a range of functionalities when a change is made.

Contrast running a set of unit tests in a few seconds covering many test cases, taking a few minutes to run a full web application, browsing to a page, entering data, checking the result, and only having covered a single test case.

Examining the test results

Running the unit tests produces results that show all the tests are failing. This is expected as we have not written the implementation yet.

We are not ending the chapter on a low note! What we have done is considered the design, defined some requirements for the object, and created a test suite to validate the solution as we create it.

Summary

Serious applications need robustness, and you will now have a grasp of unit testing in Dart to aid in the rapid production of quality software. We have also covered how to log vital operation details, categorized to the correct level, for long-term monitoring of a system.

Connecting Dart to a serious database is critical for many applications, and you can now carry this out using an industrial-strength database system. You now have the skills to connect to a remote web service to collect real data.

In the next chapter, our exploration of database systems will cover extracting the data into our application, and we will mine this constantly updating data store to populate a rich data grid view of the incoming data, which will be updated in real time.

8

Live Data and a Web Service

Data is a precious thing and will last longer than the systems themselves.

– Tim Berners-Lee

For many today, it is hard to imagine that, at one time, most personal and work computers did not connect to a wider network. Not that long ago, a spontaneous network stream of instant notifications to a device in your pocket was unthinkable.

Newsfeeds, streams, and updates are the standard of web applications today. It is not enough for data to be accessible or discoverable. It has to reach out to the user, ideally even before they know they want it.

Freeing the data

The data that was collected by the data monitor is rather trapped in the database, and while the database system PostgreSQL has a reasonable display in the GUI of pgAdmin, a web page would be more pleasing, and viewable by a wider audience.

Sharing the data to be used by a client? I hope JSON sprung immediately to your mind! The end goal of this phase of the project is a web page that we can visit to view the earthquake information in a friendly manner. The page should update itself smoothly and the data should be made available in a consumable manner for future expansion.

Open the project in the QuakeMonitorDB folder in the sample code for this chapter.

Reworking the data collector

The iteration of the GeoJSON data collector in the previous chapter was a straightforward capture of data to the database. In order to make the data easier to handle for other applications, the program will be improved to make a table of individual features listed.

Adding a new data table

The separated features in the database will be stored in a new table named, `dm_quakefeatures`, as follows:

```
CREATE TABLE dm_quakefeatures
(
    qfeat_id text NOT NULL,
    geojson text NOT NULL,
    modified_date timestamp without time zone DEFAULT now(),
    CONSTRAINT dm_qfeat_pkey PRIMARY KEY (qfeat_id)
)
WITH (
    OIDS=FALSE
);
ALTER TABLE dm_quakefeatures
OWNER TO webquakeuser;
```

The SQL statements for creating the database are found in the `SQL` folder of the sample code.

Filtering the data

The `DataMonitor` class in the file `quakemonitordb.dart` has an updated `fetchData` method, which individually stores the underlying features after storing the raw JSON to the database:

```
GeoUpdate update = new GeoUpdate(JSON.encode(dataset));

log.fine("Features to process ${update.features.length}");

for (int i = 0; i < update.features.length; i++) {
    GeoFeature feature = update.features[i];

    await daoGeoJson.storeFeature(feature.id, feature.toJson());
}
```

The `GeoUpdate` class, which is found in `geoupdate.dart`, takes a JSON string as a parameter in the constructor. The following excerpt from the constructor shows how it extracts and adds the individual features to the collection:

```
GeoUpdate(String json) {
    src = json;
    features = [];

    if (src.length > 0) {
        try {
            Map rawJsonObj = JSON.decode(src);

            if (rawJsonObj["features"] != null) {
                var items = rawJsonObj["features"];

                items.forEach((Map i) {
                    features.add(new GeoFeature(i));
                });
            }
        } catch (exception) {
            print("Error decoding JSON.");
            print("$src");
            print(exception);
        }
    }
}
```

The `GeoFeature` class, which is found in the `geofeature.dart` constructor, takes a `Map` object that is populated with the properties of the feature. The purpose of this is to extract the required information about the feature that needs to be stored for our application, as shown in the following code snippet:

```
GeoFeature(Map newProperties) {
    properties = newProperties["properties"];
    geometry = newProperties["geometry"];
    id = newProperties["id"];
    time = properties["time"].toString();
    title = properties["title"].toString();
    type = properties["type"].toString();
    mag = properties["mag"].toString();
    place = properties["place"].toString();

    url = properties["url"].toString();
    detail = properties["detail"].toString();
}
```

As the preceding excerpt shows, this is not a complicated object that arranges data into convenient fields.

Converting the feature to JSON

To convert the object back to JSON, a specific method will be added, called `toJson`, as shown in the following code snippet:

```
String toJson() {  
    Map out = {};  
    out["properties"] = properties;  
    out["geometry"] = geometry;  
    return JSON.encode(out);  
}
```

The key fields of `properties` and `geometry` are added to `Map` before being passed to the encoding function and returned to the caller.

Improving the data maintenance

Since there is a new table in the database, we will want to update the maintenance program `maintenance.dart` to clear both tables, as shown in the following code snippet:

```
deleteRecords() async {  
    var dbConn;  
    try {  
        dbConn = await connect(uri);  
  
        var query = "delete from dm_quakefeed";  
        await dbConn.execute(query);  
  
        query = "delete from dm_quakefeatures";  
        await dbConn.execute(query);  
    } catch (exception, stacktrace) {  
        log.severe("Exception getting info.", exception, stacktrace);  
        print(exception);  
    } finally {  
        dbConn.close();  
    }  
}
```

This is carried out by using another `delete` SQL statement, and reuses the existing connection to the database.

Storing the single feature

The storage of the feature will take two separate queries. The first will use the ID field to check if it is currently in the table. If it is, then the function will exit without storing the JSON string. Let's have a look at the following code snippet:

```
storeFeature(String featureID, String json) async {
    var dbConn;
    try {
        dbConn = await connect(uri);

        var query = """select count(*) as Count
            from dm_quakefeatures where qufeat_id ='$featureID'
            """;
        var results = await dbConn.query(query).toList();

        if (results[0][0] != 0) return;

        await dbConn.execute(
            'insert into dm_quakefeatures (qufeat_id, geojson) values (@
            qufeat_id, @geojson)',
            {'qufeat_id': featureID, 'geojson': json});
    } catch (exception, stacktrace) {
        log.severe("Exception storing Feature.", exception, stacktrace);
        print(exception);
        print(stacktrace);
    } finally {
        dbConn.close();
    }
}
```

The first query returns the number of records that match the ID. The result returned is converted to a list format, and the value is accessed via `[0][0]`—the first record and first field. If the result is not zero, then the method returns without further action.

The second SQL operation stores the feature in the database by using an `insert` statement.

Running the data application

The application can be run as usual from the command line. You may wish to run the data collector like this in order to save having multiple IDEs open for a long period of time. Let's have a look at the following code snippet:

```
$ cd QuakeMonitorDB/  
QuakeMonitorDB$ dart bin/main.dart  
Starting QuakeMonitor DB version...
```

This shows the launching of the QuakeMonitor program in a terminal. If a close look is taken at the processes running on the system, the resources used by the Dart VM can be seen, as shown in the following screenshot:

Tasks: 143 total, 1 running, 142 sleeping, 0 stopped, 0 zombie											
%Cpu(s): 20.5 us, 5.5 sy, 0.0 ni, 41.6 id, 32.4 wa, 0.0 hi, 0.0 si, 0.0 st											
KiB Mem: 2063184 total, 1891244 used, 171940 free, 175320 buffers											
KiB Swap: 1047548 total, 3048 used, 1044500 free. 872680 cached Mem											
<hr/>											
PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
1587	daftspa+	20	0	446328	105652	43228	S	15.6	5.1	1:10.49	dropbox
1025	root	20	0	175268	52076	33536	S	3.6	2.5	1:07.62	Xorg
1651	daftspa+	20	0	144992	27252	21508	S	2.7	1.3	0:01.47	mate-terminal
2546	daftspa+	20	0	65960	24984	8764	S	1.7	1.2	0:04.36	dart
1911	daftspa+	20	0	147900	127324	8544	S	1.3	6.2	0:41.80	dart
1748	daftspa+	20	0	1073912	290220	23496	S	0.7	14.1	1:50.81	java
2403	root	20	0	0	0	0	S	0.7	0.0	0:03.64	kworker/0:1
7	root	20	0	0	0	0	S	0.3	0.0	0:01.99	rcu_sched

The Dart process launched for the QuakeMonitor measures approximately a mere 64 megabytes, and starts nearly instantly.

Creating the web service

Now that the data collection and storage has been re-factored, it is time to build a web service. In some ways, this is similar to the blog server we created previously. However, we want to be ambitious, and so will plan for future development by starting to build a web API for our data.

The REST architecture style is very popular as it is well suited to JSON and being consumed by client web applications.

REST is an architecture style for web services that has gained acceptance as a less complex alternative to WSDL and SOAP.

REST builds on HTTP and uses the verbs GET, POST, PUT, and DELETE. It can use any Internet format as a data type (XML, images, and so on), but usually uses JSON.

There is no agreed standard for RESTful web APIs, as it is a style rather than a protocol. This means that it is very flexible, but it can also be hard to get an answer to questions such as "is this the right way to do it?". The usual advice is to be pragmatic and follow REST as far as makes sense.

For more information, see <http://www.restapitutorial.com/>.



Using the package rpc

A RESTful API has a number of conventions and expectations. Fortunately, there is an existing Dart package, authored by the Dart development team, that covers most of the implementation detail.

The `rpc` package is available from Pub at <https://pub.dartlang.org/packages/rpc>.

The source code is available from GitHub at <https://github.com/dart-lang/rpc>.



As a bonus, the `rpc` package also includes discoverability features (via Google's Discovery Document, <https://developers.google.com/discovery/v1/reference/apis>), which allow the easy creation of client code in any supporting language.

Initiating the API server

In the `main.dart` file, the server is set up and set to serve on the local machine using port 8080, as shown in the following code snippet:

```
library io_rpc_sample;

import 'dart:io';
import 'package:georestwebservice/georestwebservice.dart';
import 'package:rpc/rpc.dart';

final ApiServer _apiServer = new ApiServer(apiPrefix: '/api',
prettyPrint: true);
```

```
main() async {
    _apiServer.addApi(new Quake());
    _apiServer.enableDiscoveryApi();

    HttpServer server = await HttpServer.bind(InternetAddress.ANY_IP_V4,
8080);
    server.listen(_apiServer.httpRequestHandler);
}
```

The `_apiServer` instance is set up with two parameters. The first sets an expected prefix for the API so that valid URLs will start with the form

`http://127.0.0.1:8080/api`. The second parameter, `prettyPrint`, formats the JSON for easy viewing in the browser or another consuming application that may include a debugger. This is particularly useful when the JSON is heavily nested.

The `_apiServer` instance is then configured with a `Quake` object. This object contains the implementation of the API methods. The `Quake` class definition is located in the file `georestwebservice.dart`, and is heavily annotated. Let's have a look at the following code snippet:

```
@ApiClass(
    name: 'quake',
    version: 'v1',
    description: 'Rest API for Earthquake feature data.')
class Quake {
    ...
}
```

The `@ApiClass` annotation exposes this class as an API on the `rpc` server. This allows separate class to handle the implementation per API name and per version. The different versions are accessed by changing the version or name in the URL that is being requested from the server.

Exposing methods

The simplest method is the classic Hello World example:

```
@ApiMethod(path: 'hello')
QuakeResponse hello() {
    return new QuakeResponse()..result = 'Hello world.';
}
```

The path setting in the attribute defines the name part of the URL. A QuakeResponse object is returned with a string set as the result, as shown in the following screenshot:



The QuakeResponse class is a simple class that is automatically converted to a JSON response by the `rpc` package, as shown in the following code snippet:

```
class QuakeResponse {
    String result;
    QuakeResponse();
}
```

Classes used as a response are required to be concrete (not abstract) and to have a constructor with no parameters. Also, the constructor must not be a named constructor.

Error handling of incorrect requests

The `rpc` package handles any URLs for incorrect methods automatically, and provides an error message in JSON format:



The error handling also covers an exception or other error occurring within the implementation of an API method, as shown here:

```
@ApiMethod(path: 'implementationerror')
QuakeResponse implementationError() {
    throw new Exception();
}
```

This doomed-to-fail method will produce a valid JSON response that the client application can handle:



Serving the latest information

For the grid view data display, the most important URL for the web API will be <http://127.0.0.1:8080/api/quake/v1/latest>:

```
@ApiMethod(path: 'latest')
List<String> latest() {
    DaoQuakeAPI qa = new DaoQuakeAPI();
    return qa.fetchTopTenLatest();
}
```

The `rpc` package handles the conversion to JSON of several common return types. The `List<String>` is probably the most common collection.

Supplying the data

The `DaoQuakeAPI` class is implemented in the file `daoapi.dart` and retrieves the features list from the database, as shown in the following code snippet:

```
Future<List<String>> fetchTopTenLatest() async {
    var dbConn;
    try {
        dbConn = await connect(uri);

        var query = """
            select geojson from dm_quakefeatures order by modified_date desc
            limit 10
            """;

        List<Row> dbResults = await dbConn.query(query).toList();
        List<String> results = new List<String>();
    }
}
```

```
    dbResults.forEach((record) {  
        results.add(record[0]);  
    });  
  
    return results;  
}  
catch (exception, stackTrace) {  
    print(exception);  
    print(stackTrace);  
}  
finally {  
    dbConn.close();  
}  
}
```

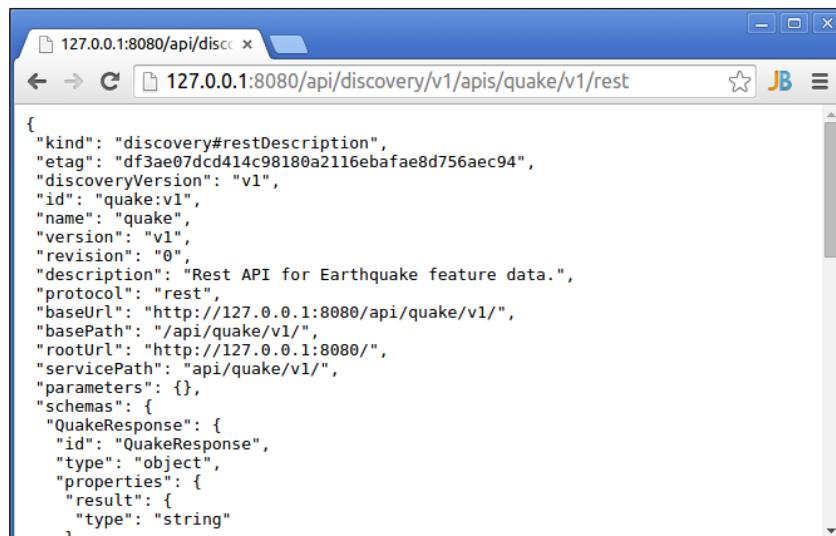
The results are returned as a list of records, and each record returned is a `List` object. This means that `dbr[0]` needs to be specified, as we only require the string in the first field.

Discovering the API

The `rpc` package provides built-in discoverability for the API. This is provided as a JSON definition of the service that is accessed from the API server:

<http://127.0.0.1:8080/api/discovery/v1/apis/quake/v1/rest>

This definition would be consumed by development tools to produce wrapper objects and supporting code in the required language. Let's have a look at the output in the following screenshot:



The header contains identifying information, like the version, and key usage data, such as the base URL for calling methods.

The schemas section summarizes all the data types used by the API, the methods lists, and the available function calls that can be made. In addition, the resources section (empty for this project) would contain the objects that group API methods together.

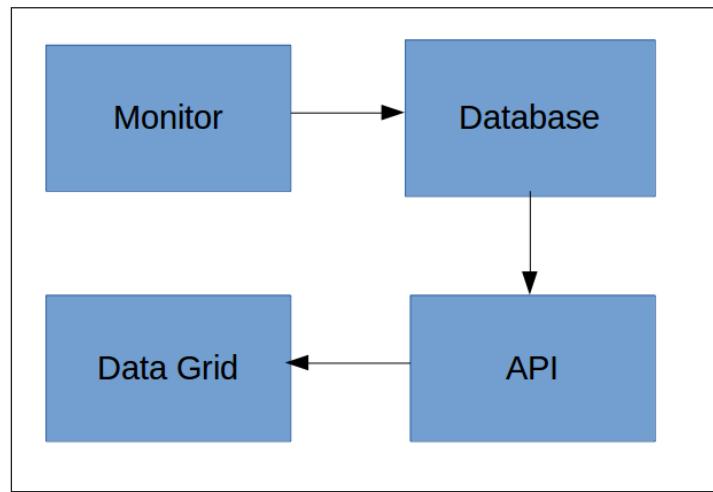
Running the web service

As with the data monitor, you may wish to run the REST web service outside of the development environment. To run from a terminal (command line), enter the following code:

```
$ cd georestwebservice/  
$ dart bin/main.dart  
Starting georestwebservice...
```

Recapping the system so far

We now have two parts of the system in place. The first contacts the live feed and pulls out the breaking information, and the second exposes the collected data via a standard API:



Both of these will need to be running, and it would be advisable to have them running outside the development environment. This varies between operating systems, so use a batch file or shell script as appropriate.

Open a process explorer application (`top` on the Linux command line, Task Manager on Windows, or Activity Monitor on Mac OS) and you will be able to see how efficient the Dart VM is. The applications start immediately and use a relatively small amount of memory.

Consuming application

We now have two parts of the solution: the collected data and the web service to publish it. The next step is to build a client application that will talk to the web service. Initially, we will have a details grid view (or `table` for the HTML-minded) of the ten latest incoming data.

Rather than having the user update the page, the screen will auto-update periodically with the latest earthquake information.

Packaging the grid

The grid display may be useful for other applications, so it will be split off into its own project as the package `webgridview`, as shown here:

```
name: 'GridViewer'  
version: 0.0.1  
description: A grid viewer for GeoJSON data.  
environment:  
  sdk: '>=1.0.0 <2.0.0'  
dependencies:  
  browser: '>=0.10.0 <0.11.0'  
  intl: any  
  webgridview:  
    path: ../webgridview/
```

The package can be referenced in the `pubspec.yaml` as a local package.

Initiating the real-time updates

The main function in `main.dart` will perform an initial update of the grid and then initiate a `Timer` that will update the page on a periodic basis:

```
void main() {  
  performUpdate(null);  
  updater = new Timer.periodic(new Duration(seconds: 10),  
  performUpdate);  
}
```

The periodic timer handler `performUpdate` is provided a `Timer` object as a parameter when it is triggered every 10 seconds. When we are directly calling it, `null` is provided instead of the `Timer` instance.

Performing the update

The two elements being updated are the current time, with a countdown to the next refresh, and the grid of the data. Let's have a look at the following code snippet:

```
void performUpdate(Timer triggerTimer) {
    DivElement outputDiv = querySelector('#timestatus');

    DateTime currentDateTime = new DateTime.now();
    DateFormat timeStamp = new DateFormat("hh:mm a");

    if (triggerTimer != null) {
        secondsToUpdate -= 10;
    } else update DataView();

    if (secondsToUpdate == 0) {
        update DataView();
        secondsToUpdate = 60;
    }

    outputDiv.text =
        "${timeStamp.format(currentDateTime)} - $secondsToUpdate seconds
        until refresh.";
}
```

The `secondsToUpdate` variable is decremented from 60 to 0 and triggers an update every six calls so that the display changes every minute.

Fetching the JSON

The `update DataView` function in the `main.dart` file covers fetching the data and updating the grid view:

```
...
HttpRequest.getString(jsonSrc).then((String data) {
    outputDiv.children.clear();

    List items;
    try {
```

```
    items = JSON.decode(data);
} catch (exception, stackTrace) {
    print(exception);
    print(stackTrace);
}
...

```

The `DivElement outputDiv` is the container for the dynamically updating content, which is entirely cleared at each update. The incoming JSON data is decoded and stored in the `List` object named `items`.

Configuring the grid view control

The data for the grid is provided as a structure of a list of lists. The first list in the structure forms the header of the table. Let's have a look at the following code snippet:

```
...
if (items != null) {
    List allQuakeData = [];
    allQuakeData.add(
        ['Time', 'Magnitude', 'Type', 'Tsunami', 'Place', 'Sig',
        'Link']);
    ...
    items.forEach((String post) {
        Map decodedData = JSON.decode(post);

        List quakeData = [];
        quakeData.add(convertTime(decodedData['properties']['time']));
        quakeData.add(decodedData['properties']['mag']);
        quakeData.add(decodedData['properties']['type']);
        quakeData.add(decodedData['properties']['tsunami']);
        quakeData.add(decodedData['properties']['place']);
        quakeData.add(decodedData['properties']['sig']);
        quakeData.add(decodedData['properties']['url']);

        allQuakeData.add(quakeData);
    });
    outputDiv.append(Gridview.getTable(allQuakeData));
}
...

```

The JSON items are iterated over and the values for the grid are extracted. The only value not used directly is the `time` field, which requires formatting.

Formatting the time

The time provided for each feature in the JSON is an integer number. This is the time recorded as the number of milliseconds since a point in time, usually called the epoch (which is the start of the year 1970 at exactly 00:00:00 1970-01-01). Let's have a look at the following code snippet:

```
String convertTime(int milliTime) {  
    DateTime dt = new DateTime.fromMillisecondsSinceEpoch(milliTime);  
    DateFormat timeStamp = new DateFormat("hh:mm:ss a");  
    return timeStamp.format(dt);  
}
```

The `DateTime` format provides the named constructor `fromMillisecondsSinceEpoch`, which returns a regular `DateTime` object. This can be converted to a string and added to the table for display in the data grid.

Working with date and time

Dates and times are very important data types in all kinds of applications, and the Dart `intl` package has functionality for parsing and formatting dates. Countries have specific formatting and ordering preferences, and, of course, everyone is different and uses multiple formats.

The format used in the grid view may not be to your liking and may be better displayed as a 24-hour format. The project `timesdate` in the sample code for this chapter explores the different date formats that are available. This short command-line program shows some of the options available; the list is quite long, and it is fully documented in the `intl` package documentation. Let's have a look at the following code snippet:

```
DateTime currentTime = new DateTime.now();  
  
print("\nDate and Time Demo");  
printTime(currentTime, "hh:mm a");  
printTime(currentTime, "HH:MM");  
printTime(currentTime, "y");  
printTime(currentTime, "d");  
printTime(currentTime, "M");  
printTime(currentTime, "EEEE");  
...  
...
```

The preceding code renders the following output display, which will vary according to the time at which it is executed:

```
Date and Time Demo
hh:mm a      04:12 PM
HH:MM        16:06
Y            2015
d            13
EEEE        Saturday
```

The function `printTime` formats the input `DateTime` to the desired style. The `padRight` method on the `String` object is used to make a simple table by adding spaces to pad out the string to 12 characters, as shown here:

```
void printTime(DateTime dt, String format) {
    DateFormat timeStamp = new DateFormat(format);
    print("\t${format.padRight(12)} ${timeStamp.format(dt)}");
}
```

The second part of the program allows a date to be entered by using the `stdin.readLineSync` method. The user can enter a blank line to exit. The string is parsed according to the format `yyyy-MM-dd`, and the weekday is displayed if it is successfully parsed:

```
while (true) {
    String userDate = "";
    print("\nPlease enter a date (yyyy-MM-dd): ");

    try {
        DateFormat timeStamp = new DateFormat("yyyy-MM-dd");
        DateFormat outputFmt = new DateFormat("EEEE");

        userDate = stdin.readLineSync();
        if (userDate.length == 0) exit(0);

        print(outputFmt.format(timeStamp.parse(userDate)));
    } catch (exception, stacktrace) {
        print('Error parsing the entered date.');
        print(exception);
        print(stacktrace);
    }
}
```

The parsing is wrapped in `try catch` to deal with an exception raised with a failed parse of a date string.

Building the table

The class `GridView`, located in the file `lib/src/webgridview_base.dart`, prepares a table from the list of lists provided by the calling web page. The static method `getTable` performs the task. As it is static, an instance of the class is not required to call this function. Let's have a look at the following code snippet:

```
static TableElement getTable(List rows) {
    TableElement te = new TableElement();
    rows.forEach((row) => addRow(te, row));
    return te;
}
```

The `addRow` function iterates over the columns in the results. The header row is added as a regular row as the CSS will take care of the special formatting of the first row, as shown here:

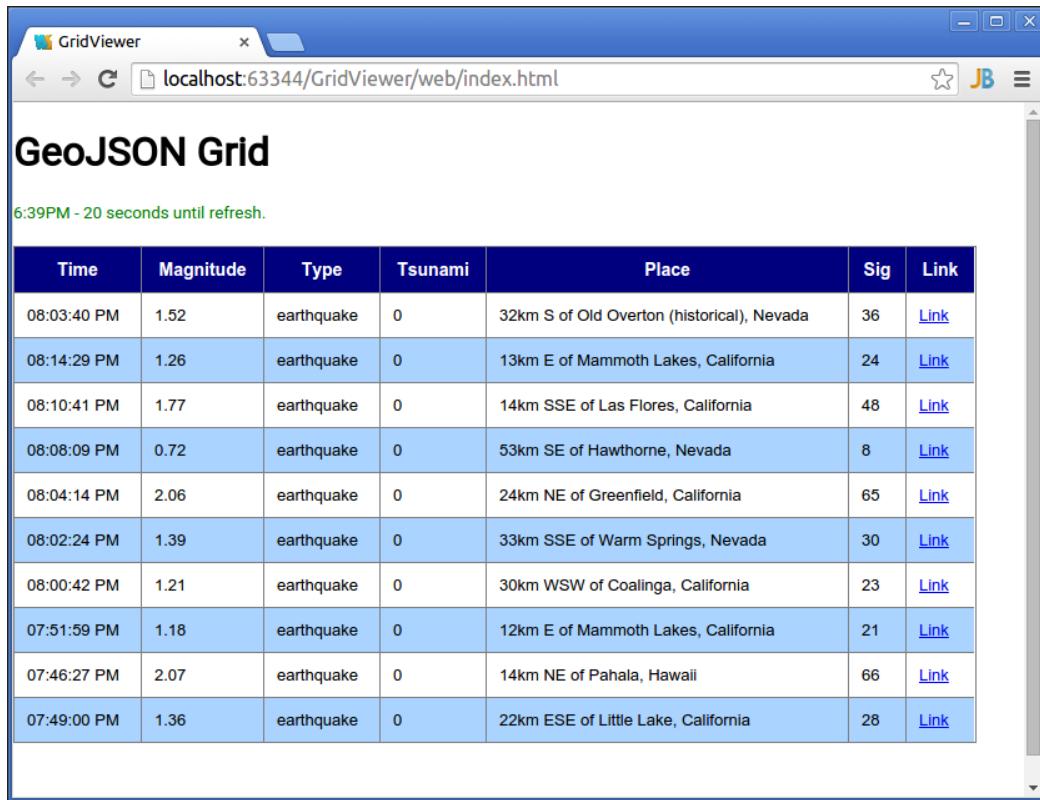
```
static addRow(TableElement table, List cols) {
    TableRowElement tableRow = table.addRow();

    cols.forEach((column) {
        TableCellElement tableCell = tableRow.addCell();
        String content = column.toString();
        if (content.startsWith('http')) {
            content = "<a href=\"$content\">Link</a>";
            tableCell.appendHtml(content);
        } else tableCell.text = content;
    });
}
```

The content is added as text to the table cell, with one special case of strings that start with `http`. These URLs are formatted into HTML with a generic link text, and the `appendHtml` method is used to ensure that the text is not sanitized before being added to the page.

Showing the page

Launching the application in Dartium will show the `index.html` page:



The screenshot shows a web browser window titled "GridViewer". The address bar displays "localhost:63344/GridViewer/web/index.html". The main content area is titled "GeoJSON Grid" and contains the following table:

Time	Magnitude	Type	Tsunami	Place	Sig	Link
08:03:40 PM	1.52	earthquake	0	32km S of Old Overton (historical), Nevada	36	Link
08:14:29 PM	1.26	earthquake	0	13km E of Mammoth Lakes, California	24	Link
08:10:41 PM	1.77	earthquake	0	14km SSE of Las Flores, California	48	Link
08:08:09 PM	0.72	earthquake	0	53km SE of Hawthorne, Nevada	8	Link
08:04:14 PM	2.06	earthquake	0	24km NE of Greenfield, California	65	Link
08:02:24 PM	1.39	earthquake	0	33km SSE of Warm Springs, Nevada	30	Link
08:00:42 PM	1.21	earthquake	0	30km WSW of Coalinga, California	23	Link
07:51:59 PM	1.18	earthquake	0	12km E of Mammoth Lakes, California	21	Link
07:46:27 PM	2.07	earthquake	0	14km NE of Pahala, Hawaii	66	Link
07:49:00 PM	1.36	earthquake	0	22km ESE of Little Lake, California	28	Link

Keep this page open over a period of time and the table will update smoothly as new data becomes available. Events from outside the USA do appear, though less often. It is fairly rare, but the Type column does not always read 'earthquake'.

Summary

The applications created in this chapter have been numerous, but all work together to reach the goal of a live display of data.

The data collector transforms raw live data into a processable form. The REST API was created with the `rpc` package, taking much of the work out of the way. Creating a new web API method is a case of adding an annotated regular Dart method.

The consuming web application showed how to connect to a JSON standard web service and produce high-quality output, updated in real time. The HTML DOM was used to swap in a new table-based display.

The incoming data on earthquakes has proven quite fascinating to watch, but it could be more visual. In the next chapter, we'll create a visualization using the feature geometry data stashed in the database, and look at expanding the API to do more than provide a data feed.

9

A Real-Time Visualization

There's just something hypnotic about maps.

– Ken Jennings

Maps are indeed hypnotic as are flashing indicator lights, which may give you a rough idea of where the earthquake monitor project is going next. It is the logical format to display all that rich geographical data we are collecting.

An animated display is one step up from a static display, and the step beyond animation is to make it interactive. We will look at accessing the web browser features from Dart in order to integrate to the desktop and work with the user's location.

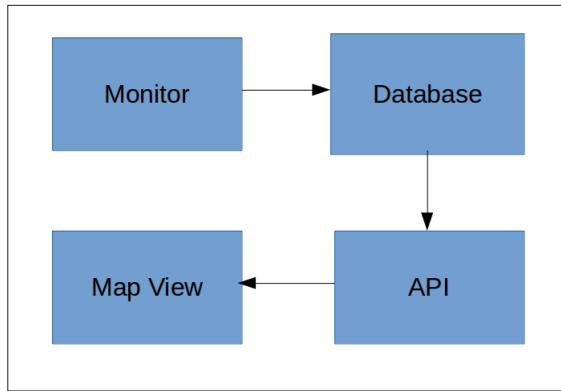
To work in a professional development environment or to share with the community online, we will need to create documentation from our Dart source.

Iteration overview

The grid view gave a very detailed view of the data obtained from the stack of software that collected the data from the web service, was stored in a database, and was then shared from the database via an API.

For this part of the project, we will reuse every part other than a gridview, and build a new visualization using a world map.

Let's take a look at how the map view fits into the overall system:



To ensure the map view has some data, the **Monitor** needs to be running, as does the **API**.

The main project for this chapter is the `mapviewer` project in the sample code bundle.

Application overview

The `mapviewer` project is a Dart web application with an `index.html` entry point that kicks off the application on `main.dart`:

```
main() async {
    quakeMap = new MapDisplay(querySelector('#mapview'), width, height);
    await quakeMap.loadImage();

    featPlotter = new FeaturePlotter(width, height, quakeMap.mapCtx);

    quakeMap.showPopup = showPopup;
    quakeUpdate();

    querySelector('#zoombtn').onClick.listen(zoomMap);
    querySelector('#locatebtn').onClick.listen(locateUser);
    querySelector('#sortbtn').onClick.listen(sortFeatures);

    new Timer.periodic(new Duration(seconds: 60), quakeUpdate);
    new Timer.periodic(new Duration(milliseconds: 100),
        animationUpdate);
}
```

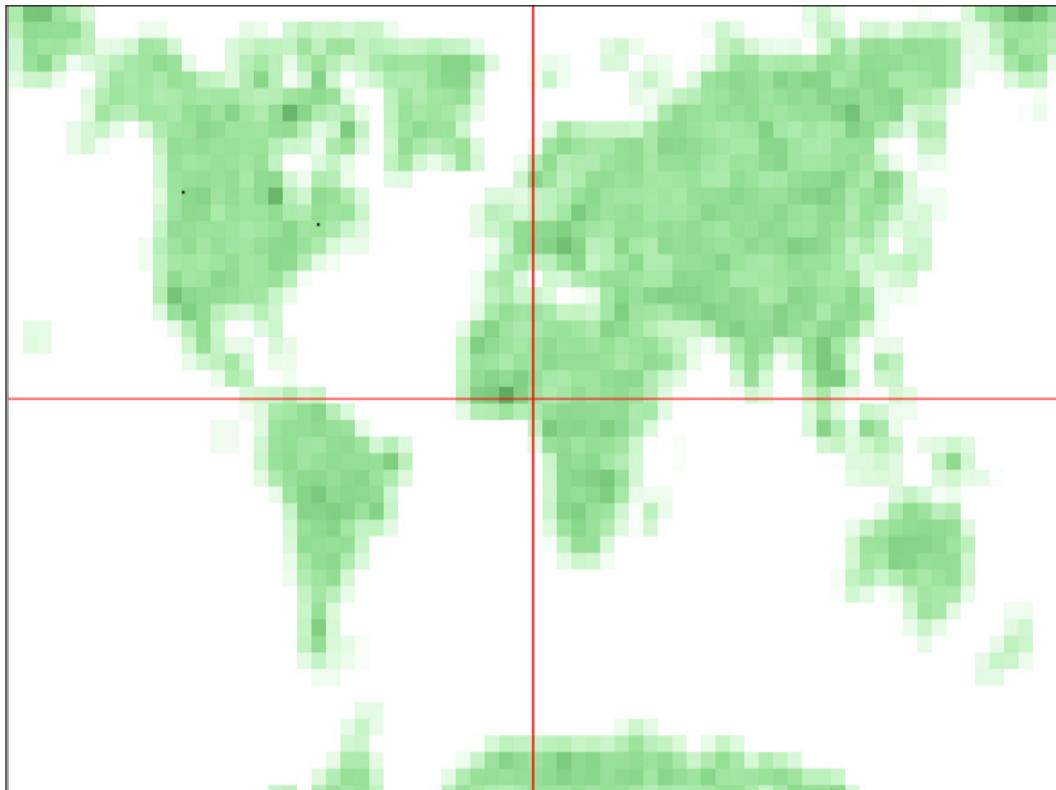
The `MapDisplay` class sets up the map on the web page (on the `div` element with the ID `mapview`) and the image is loaded in. Then, `quakeUpdate()` is called to ensure the initial display of the map and data on the page.

Once the initial display is handled, the **Zoom** button is connected to the `zoomMap` function. Then the two timers have handlers to deal with updating the quake data and the animation display. The `FeaturePlotter` class handles the data and display updates and is used by `quakeUpdate` and `animationUpdate`.

Drawing the map image

The HTML5 canvas will host the live display. The background is a block-like map of the world; feel free to find and use a different version!

Let's have a look at the following screenshot:



The red lines represent 0 degrees of latitude and longitude.

Plotting on the map

The geometry point contains three pieces of data in a list called `coordinates`. They are the **longitude**, **latitude**, and also the **depth** in kilometers of where the earthquake originated:

```
{type: Point, coordinates: [-117.4505005, 34.2316666, 13.31]}
```

To map these coordinates to the map, we use the method `toPoint` in the `FeaturePlotter` class in the file `featureplotter.dart`, which performs the conversion as follows:

```
Point toPoint(double longitude, double latitude) {
    double x = 0.0;
    double y = 0.0;
    double hdeg = width.toDouble() / 360;
    double vdeg = height.toDouble() / 180;

    x = (width / 2) + (hdeg * longitude);
    y = (height / 2) - (vdeg * latitude);

    return new Point(x.toInt(), y.toInt());
}
```

The longitude and latitude is based on the spherical nature of the Earth, and the x axis is 360 degrees wide with 0 being the mid-point. The y axis is 180 with 0 being the midpoint at the equator. In terms of the GeoJSON data, the x values are in the range of 180 to +180 and y is from -90 to 90.

Dart, at the time of writing is at version 1.11 and you may be wondering if it will reach version 2 in the near future. Yes, it will!

During the first Dart Developer Summit, version 2 of Dart was mentioned for the first time by Lars Bak. While it was hinted that there may be some language changes, it was strongly emphasized these would only be made with tools to handle updating existing code. In Bak's words, "this is not Python!" This quote is a reference for the somewhat disruptive move from Python 2 to Python 3.

One key change that has already begun is the moving of some packages out of the SDK. This allows them to update on their own schedule. For example, `dart:html` has to respond more quickly than the SDK update cycle to web browser changes.



Animating the display

The `animationUpdate` handler set up in the `main` function in the `main.dart` file fires the `FeaturePlotter` method `updateDisplay`:

```
void animationUpdate([Timer t = null]) {
    featPlotter.updateDisplay();
}
```

This method in turn calls an update on each map indicator object instance, as shown in the following code snippet. The `userLocation` will be covered later in this chapter:

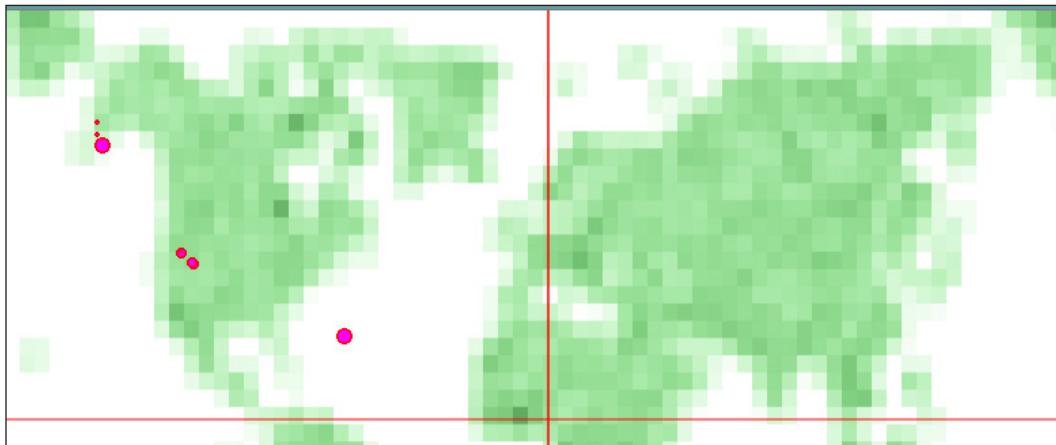
```
void updateDisplay() {
    mapIndicators.forEach((mapIndicator) => mapIndicator.update());
    userLocation.forEach((mapIndicator) => mapIndicator.update());
}
```

The `MapIndicator` class in `mapindicator.dart` handles the display and animation of the feature. The display will be a growing circle point with the maximum size of the point matching the magnitude of the earthquake event. The constructor and `update` method handle the task of keeping the plotted shape within range:

```
MapIndicator(this.x, this.y, this.ctx, this.magnitude) {
    maxWidth = (width + 1) + magnitude * 2;
}

void update() {
    width++;
    if (width == maxWidth) width = 0;
    draw();
}
```

This will cause the point width to grow upwards to the limit before being reset back to a value of 0, as shown here:



The circle is drawn using the arc method of the HTML5 canvas:

```
void draw() {  
    ctx  
        ..beginPath()  
        ..arc(x, y, width, 0, 2 * PI, false)  
        ..fillStyle = colorPrimary  
        ..fill()  
        ..lineWidth = 1  
        ..strokeStyle = colorSecondary  
        ..stroke();  
}
```

The `x` and `y` specify the point where the feature will be drawn and `w` specifies the radius. The next two parameters are the start and end angles, which are measured in radians, and in a circle there are `2 * PI` radians. The constant value for `PI` is taken from the `dart:math` library, which is amongst the imports of this class. The last parameter of this method is a flag to determine whether the arc should be drawn counterclockwise or not.

Fetching the data

The data is fetched as an `HttpRequest` in the same manner as in the grid view project, and the fields we are interested in are put into the list `quakepoints` before being filtered into the `geoFeatures` list:

```
fetchFeatureList() async {
    geoFeatures.clear();

    String data = await HttpRequest.getString(jsonSrc);
    List items;
    List quakePoints = [];

    try {
        items = JSON.decode(data);
    } catch (exception, stacktrace) {
        print(exception);
        print(stacktrace);
    }

    if (items != null) {
        items.forEach((String post) {
            Map feature = JSON.decode(post);

            List quakedata = [
                feature['geometry']['coordinates'][0],
                feature['geometry']['coordinates'][1],
                feature['properties']['mag'],
                feature['properties']['place'],
                feature['properties']['type'],
                feature['geometry']['coordinates'][2]
            ];

            quakePoints.add(quakedata);
        });
    }

    quakePoints.where((qp) => qp[4] == 'earthquake').forEach((qp) {
        geoFeatures.add(qp);
    });
}
```

All the items returned are added to the quake points list, but we only want to add the earthquake ones to the geoFeatures list. The list may contain other types such as quarry blast.

Using a where clause, we can filter the list an item at a time. The list item is run through the function, and the list item at index number 4 is compared against the string earthquake.

This is a very convenient method for getting only the required data and best of all it is available on any iterable object. On top of this, there are number of variants such as firstWhere and lastWhere that help to quickly get a reference to the data that matches a criterion.

Updating the map indicators

Once a list of geoFeatures is available, it can be used to construct a list of mapIndicator objects that will appear on the map. The entire list is cleared out each time so that the number of indicators on the map does not build up too much over time. Let's take a look at the following screenshot:

```
updateData() async {
    await fetchFeatureList();
    mapIndicators.clear();
    geoFeatures.forEach((List feature) {
        Point pos = toPoint(feature[0], feature[1]);
        MapIndicator mi;
        mi = new MapIndicator(pos.x, pos.y, ctx, feature[2].toInt());
        mi.summary = "Magnitude ${feature[2]} - ${feature[3]}";
        mapIndicators.add(mi);
    });
}
```

This function is marked `async` as there is an `await` being used for the `fetchFeatureList` method call. Once the features are available, the `mapIndicators` list is reset and a new one is constructed.

Mouse over popups

To give the user more details about the feature on the map, a popup appears at the top of the screen when the pointer is moved over a map indicator. To get the mouse pointer coordinates, a handler can be added to handle the mouse move events on the canvas element:

```
MapDisplay(CanvasElement mapcanvas, this.width, this.height) {
    mapctx = mapcanvas.getContext("2d");
    mapcanvas.onMouseMove.listen(mouseMove);
}
```

This is carried out in the `MapDisplay` constructor and the method handles the triggering of the implementation function. This function is a field in the class that is set by the page using the map display:

```
void mouseMove(MouseEvent e) {
    if (showPopup != null)
        showPopup(e.client.x - 50, e.client.y - 90);
}
```

The implementation of the `showPopup` function is in `main.dart`. It displays the initially hidden popup above the map and then populates it with the feature details:

```
void showPopup(int x, int y) {
    bool inHotspot = false;

    if (zoomed) {
        x = x ~/ 2;
        y = y ~/ 2;
    }

    featPlotter.hotspotInfo.forEach((Rectangle<int> key, String value) {
        if (key.containsPoint(new Point(x, y))) {
            inHotspot = true;
            querySelector('#popup').innerHTML = value;
        }
    });

    if (inHotspot)
        querySelector('#popup').style.visibility = "visible";
}
```

If the map display is zoomed in, then the coordinates must be updated; this is performed with the `~/` division operator, which returns an integer value. This is faster than the equivalent `(x/2).toInt()` and gives the required whole numbers for plotting:

4.8 - 50km WSW of Purac, Philippines.

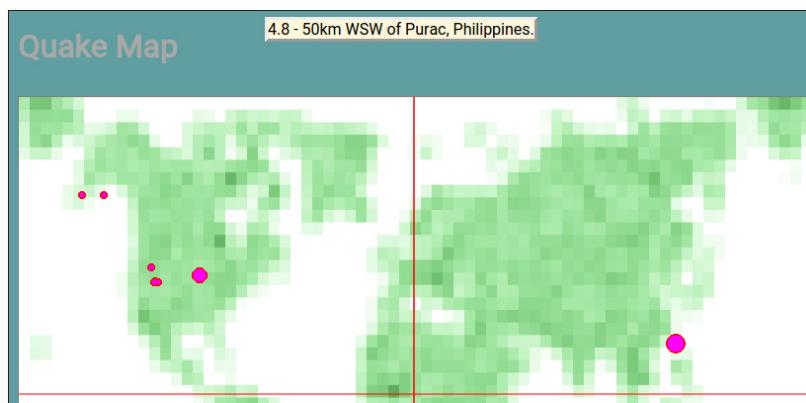
The Map containing the `hotspotInfo` is iterated over with a convenient `forEach` loop, giving both the dictionary key and the respective value. The current point of the mouse relative to the top-left corner of the canvas element is tested against each rectangle that defines the hotspot (the hotspot is the area on the map that will trigger the popup).

The hotspots are stored as a list of `Rectangle<int>` objects generated when the data list is being updated by the `FeaturePlotter` class. This class is from the `dart:math` package:

```
void updateHotspots() {
    hotspotInfo.clear();
    mapIndicators.forEach((MapIndicator mi) {
        Rectangle<int> rect = new Rectangle(mi.x - mi.maxWidth,
            mi.y - mi.maxWidth, mi.maxWidth * 2, mi.maxWidth * 2);

        hotspotInfo[rect] = mi.summary;
    });
}
```

The hotspot is defined as a square area around the map indicator when it is at its maximum width. The `summary` property gives the text to be displayed. The screenshot of the quake map is as follows:



Hopefully, this can help you in improving your geographical knowledge and learning more about Dart!

When debugging in Dartium using a REST or other similar web service, it is possible to get extra logging in the console window of the browser's developer tools for XMLHttpRequests.

Bring up the context menu (right-click), and there is an option named **Log XMLHttpRequests** that displays the HTTP requests made by the web page in the console output window:

```

[-121.5371704, 36.8165016, 0.93]
129.91739911111114 177.278328
2.222222222222223 3.3333333333333335
maxWidth 2
XHR finished loading: GET "http://127.0.0.1:8080/api/
quake/v1/latest".

```

This can be invaluable when debugging a project and saves adding logging to every part of the code that makes an HTTP request.

Zooming into the display

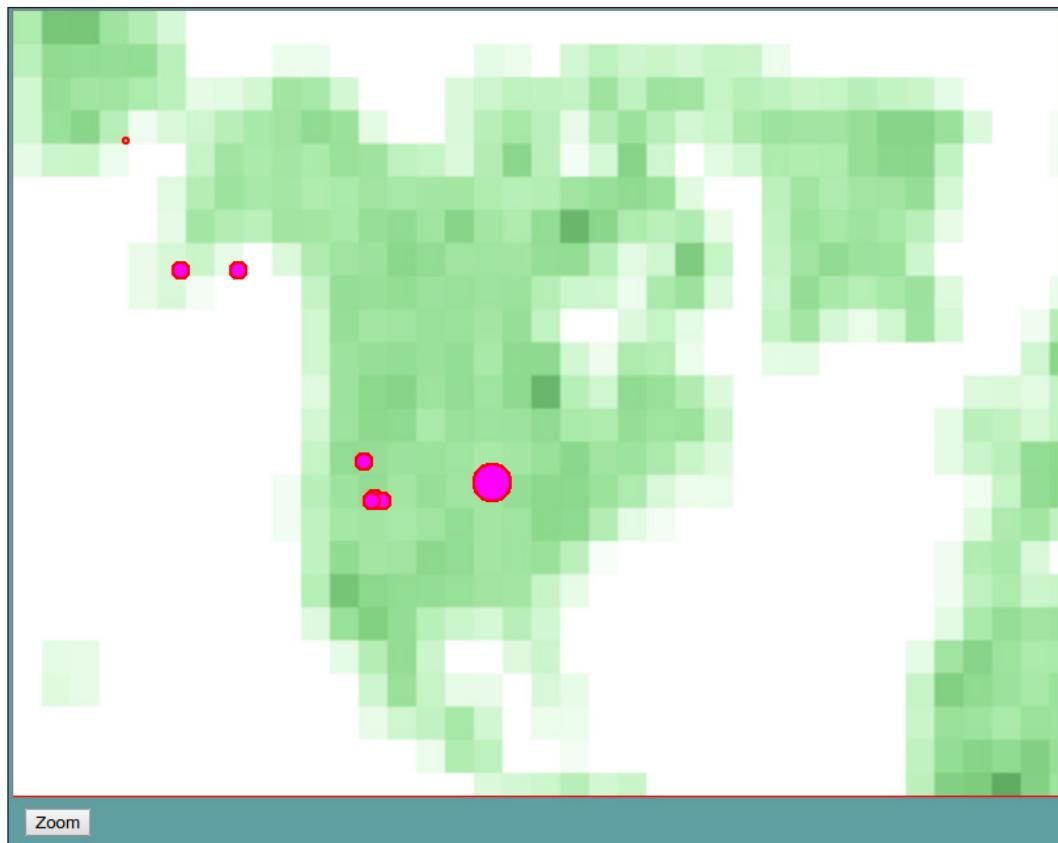
The canvas element has a range of features for 2D drawing, and this includes a scaling feature that we can use to hone in on the most active part of the map, North America. In `main.dart`, a flag exists called `zoomed`, defaulting to `false`, that aids the application in keeping track of the current display. A button is bound to the zoom code implementation:

```

zoomMap(Event evt) async {
  if (zoomed == true) {
    zoomed = false;
    quakeMap.mapCtx.resetTransform();
    quakeMap.mapCtx.scale(1, 1);
  } else {
    zoomed = true;
    quakeMap.mapCtx.scale(2, 2);
  }
  quakeMap.drawMapGrid();
}

```

The canvas is toggled between scaling settings, with a critical call to the method `resetTransform`. Without this, the scaling would not be back to normal. The entire map view must be redrawn immediately so that the user has a responsive experience. Let's have a look at the following screenshot:



If you watch this page for a while, you will find the California and Alaska feature embedded heavily in the data. As the pop-up code handles the change in scale, these are still available. Pressing the **Zoom** button a second time returns to the full world view.



The `print` function is not just for command-line applications but can be used in web applications, too. The output appears in the **Developer Tools** console log window, which can be displayed by right-clicking on the page to bring up the context menu and then clicking `Inspect Element`.

Notifying the user of an update

However impressive a created display is it is unlikely to hold the user's attention forever and they are likely to switch to another application or browser tab. Luckily, HTML5 has a feature that will allow notifications on the desktop.

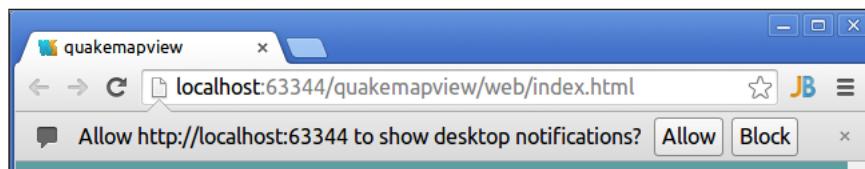
The `main.dart` function `quakeUpdate` checks if there is a significant update to notify the user about. To determine which item in the list to notify the user about, the `lastWhere` method is used. This `List` method finds the last item in the list that matches the criteria defined by the supplied function. In this case, we look at each feature's magnitude value (stored in `gf[2]`) and return `true` or `false` if the value meets the threshold:

```
quakeUpdate([Timer t = null]) async {
    await featPlotter.updateData();
    featPlotter.updateHotspots();
    quakeMap.drawMapGrid();

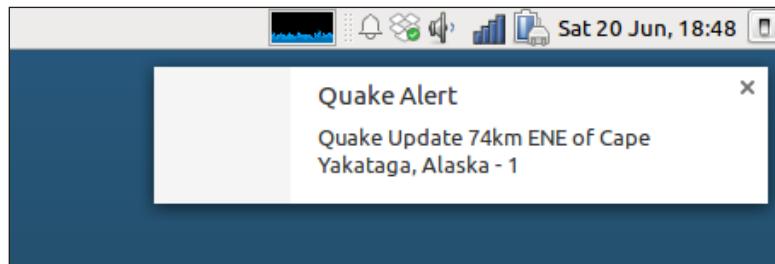
    List notiFeature =
        featPlotter.geoFeatures.lastWhere((List gf) => gf[2] > 1.9);

    String permResult = await Notification.requestPermission();
    if (permResult == 'granted') {
        Notification notifyQuake = new Notification('Quake Alert',
            body: 'Quake Update ${notiFeature[3]} - ${notiFeature[2]}');
    }
}
```

As this feature is potentially intrusive to a user's computer, the web page must ask for permission to show alerts. In Dartium, a prompt is shown at the top of the page requiring a yes or no response:



Once permission has been granted, the final step is to create an instance of the notification class. No further method call is required:

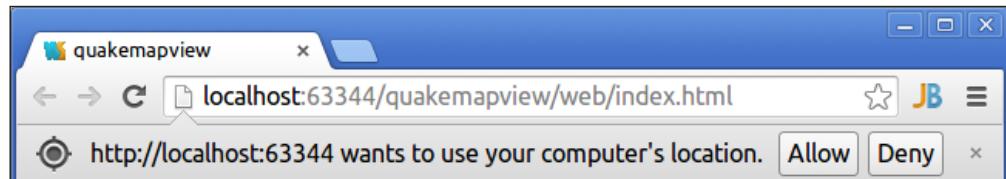


The application is set to notify quite often—you may wish to raise the level at which quakes are notified or you will find yourself dismissing a lot of notifications. The visual display of the notification may vary on different platforms and web browsers.

Plotting the user's location

Most mobile devices are fitted with a geolocation device, or the location can be determined using secondary information such as IP or even falling back to user input. This is exposed for the modern web developer to use in applications via the HTML5 geolocation API at <http://www.w3.org/TR/geolocation-API/>.

As with the desktop notifications, the user must give permission for the page to use their location. A further little hurdle is that location features do not work on Dartium. To try out this feature of the application, use pub build to create the JavaScript version and use another browser:

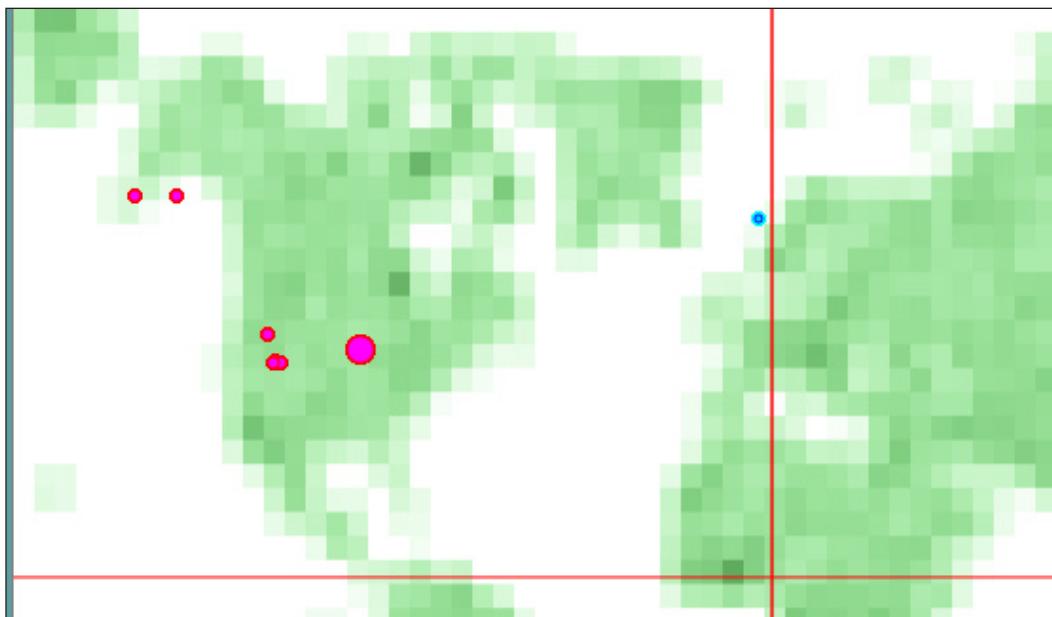


The **Locate** button on the page will attempt to obtain the user's position and draw it on the page by adding it to the featurePlotter class instance:

```
locateUser(Event evt) async {
    Geoposition geoPos = await window.navigator.geolocation.
    getCurrentPosition();
    MapIndicator mapIndicator;
    var pos =
```

```
featPlotter.toPoint(geoPos.coords.longitude, geoPos.coords.latitude);
mapIndicator = new MapIndicator(pos.x, pos.y, quakeMap.mapCtx, 4);
mapIndicator.colorPrimary = "#0000ff";
mapIndicator.colorSecondary = "#00ffff";
featPlotter.userLocation.add(mapIndicator);
}
```

The coordinates are converted in the same manner as the quakes, and the indicator is changed to a blue color so that it will stand out on the map, as shown in the following screenshot:



Once the application is compiled to JavaScript, the geolocation data is made available with user permission and the blue dot indicates the user's location, which in this instance is the relatively earthquake-free United Kingdom.

The Dart VM that runs in the command line and Dartium is not the only manifestation. The Dart team is working on two experimental versions of Dart in other contexts that vary significantly from the main Dart VM. Currently, both operate in a mobile context and focus on concurrency and high performance.

Fletch runs on desktop OSs, but is intended for Android and iOS execution with no JIT. Fletch takes a somewhat opposite approach to the highly asynchronous Dart, favoring user-level threads that are blocked but holding onto just a few resources. Fletch offers an interesting prospect for sharing the same code on different platforms; this would be a major productivity boost for mobile developers.

<https://github.com/dart-lang/fletch>

Sky is Android, only at the time of writing, and ambitiously aims to have highly responsive applications running at 120FPS. To achieve this, it prioritizes the user interface so that it can keep an 8ms response even when other processing is taking place. Currently, it can potentially run wherever the Dart VM can.

https://github.com/domokit/sky_engine/tree/master/sky/packages/sky

The Fletch and Sky prototypes, if successful, are likely to be mobile or server virtual machines and not embedded in a web server.



Sorting the feature list

So far, no use has been made of the third piece of data in the coordinates list, which is the depth of the quake. This data will be presented in the form of a sorted list of the current features when the user presses the **Sort** button. Let's have a look at the following code snippet:

```
void sortFeatures(Event evt) {
  featPlotter.sortFeatures();
  DivElement out = querySelector('#depthDetail');
  out.nodes.clear();
  featPlotter.geoFeatures.forEach((feature) {
    LIElement detail = new LIElement();
    detail.innerHTML = "${feature[5]}km - ${feature[3]}";
    out.nodes.add(detail);
  });
}
```

The list of features is sorted in the method `sortFeatures`, and as Dart knows nothing of the meaning of the contents of the list, we provide a comparator function that decides which feature is deeper by comparing the depth measurement held at index 5 of the list:

```
List sortFeatures() {  
    geoFeatures.sort( (a,b) => a[5] - b[5] );  
    return geoFeatures;  
}
```

The comparator receives two arguments and returns 0 for items that are equal, a negative integer if a is less than b, or a positive integer if a is greater than b. Let's have a look at the following screenshot:



The depths are sorted in ascending order and presented as a simple bulleted list together with the location information.

Documenting Dart code with dartdoc

Documentation is a critical part of software development, and to encourage good documentation of packages, Dart has the tool `dartdoc` (<https://github.com/dart-lang/dartdoc/>), which creates static HTML documentation based on specially formatted code comments. Previous documentation tools for Dart were part of the SDK, while `dartdoc` is a separate project developed by the Dart team.

It can be installed using `pub` at the command line:

```
pub global activate dartdoc
```

We will take a look at a Dart package project to explore the features of `dartdoc`. Open the project `quakerecord` in the WebStorm Editor, and take a look at the file `quakerecord.dart`:

```
library quakerecord;
export 'src/quakerecord_base.dart';
```

This package exports a single file, `quakerecord_base.dart`, and in this file, the class `Processor` is declared:

```
/// A lightweight object to process a raw earthquake feature.
///
/// * Must have valid JSON.
/// * Must be less than 128MB.
///
class Processor {

    // Store for the JSON.
    String _feature;

    /// The state of processing
    bool get processed => true;

    /// The default constructor.
    ///     Processor processor;
    ///     processor = new Processor('{}');
    Processor(String feature) {}

    /// Returns a converted object based on the feature passed into
    /// the constructor [Processor].
    object convert() {
        // A normal comment.
```

```

        return new Object();
    }

    /// Changes feature based on partial data.
    /// *BETA QUALITY*
    ///
    /// 1. Estimates end points.
    /// 2. Transforms polarity.
    ///
    void experimentalNewMethod() {}
}

```

Note the use of the `///` triple slash comment style—this is what `dartdoc` is looking for. The class declaration comments contain an unordered list `*`, the constructor has a section of sample code, the `convert` method has a link `[]` declared to the constructor, and `experimentalNewMethod` has a numbered list and formatting.



For full details of Dart comments and documentation guidelines, see:
<https://www.dartlang.org/articles/doc-comment-guidelines/>

There is also an additional class called `ProcessHelper` in `recordhelper.dart`:

```

/// A helper class to deal with quake feature settings.
class ProcessHelper {

    ///Swaps the data structure.
    void reversePolar() {}
}

```

To transform the comments into a web page, run `dartdoc` on the folder containing `pubspec.yaml`:

```

~/quakerecord/$ dartdoc
Generating documentation for 'quakerecord' into quakerecord/doc/api/
parsing lib/quakerecord.dart...
Parsed 1 file in 15.6 seconds.
generating docs for library quakerecord from quakerecord.dart...
Documented 1 library in 20.0 seconds.
Success! Open quakerecord/doc/api/index.html

```

A `doc` sub-folder is created in the project containing a folder named `api` that holds the documentation. Open the page in your favorite web browser and take a look at the output for the `Processor` class. Let's have a look at the following screenshot:

The screenshot shows a documentation page for the `Processor` class. At the top, there are navigation links: `quakerecord`, `quakerecord`, `Processor`, and `experimentalNewMethod`. Below these, the word `METHOD` is followed by the method name `experimentalNewMethod`. A large button labeled `SOURCE` is present. On the left side, there is a sidebar with the following sections and their contents:

- quakerecord**: `void experimentalNewMethod()`
- quakerecord**: Changes feature based on partial data. *BETA QUALITY*
 - 1. Estimates end points.
 - 2. Transforms polarity.
- Processor**:
 - PROPERTIES**: `processed`
 - CONSTRUCTORS**: `Processor`
 - METHODS**: `convert`, `experimentalNewMethod`

The main content area is titled `Source` and contains the following code:

```
/// Changes feature based on partial data.  
/// *BETA QUALITY*  
///  
/// 1. Estimates end points.  
/// 2. Transforms polarity.  
///  
void experimentalNewMethod(){  
}
```

Finally, you may be wondering where the class `ProcessHelper` from `recordhelper.dart` is located on the generated page. It is not part of the documentation as it is not exported from the package, and because of this, is considered to be a private implementation detail that the consumer of the package does not need to know about.

Summary

We have seen how visualization can be made from raw data and rendered in animated form with a client using the HTML5 canvas. This can also be made interactive with popups and zooming facilities.

HTML5 can be used to provide desktop notifications and utilize the user's geolocation as part of the application. High quality formatted and standard documentation for the quake map viewer project was generated with the SDK tool `dartdocgen`.

We have explored data structures and had a look at convenient sorting and iterating facilities.

Moving on from this project, the data can still be presented in a more detailed manner across a date range. We will look at how Dart can be used to build a staple of businesses and other applications – printable reports. Also, we will build upon the API to allow direct entry of geographical information rather than relying on an external data feed.

10

Reports and an API

Man is still the most extraordinary computer of all.

– John F. Kennedy

A report is a common output of computer systems no matter what the application type, be it a business application or a game. Although alerts, dashboards, and widgets have been developed for decades, they do not seem able to compete with the effectiveness of a report, which has a defined scope (for example, the previous six weeks of data) and can be converted into and executed in many other document formats on other devices.

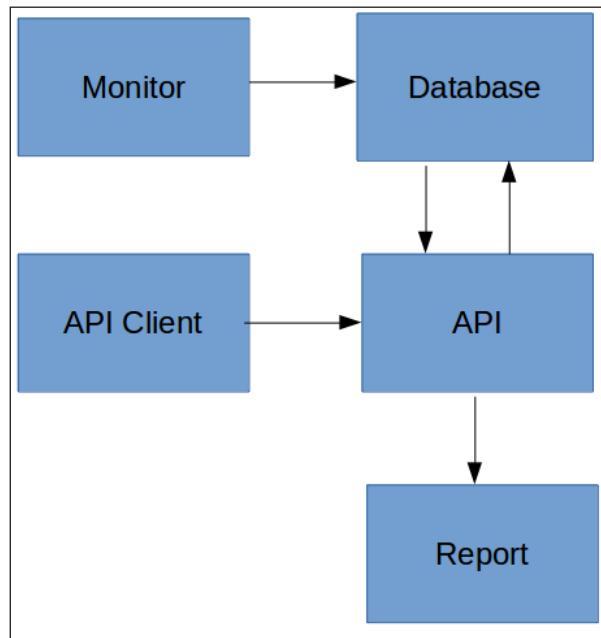
The digestion of data output from computer systems still has a human factor, even though computers have advanced a lot since the days of U.S. President Kennedy. You may wonder how long it will be before the insight of an expert is matched by a big data analysis or genuine **Artificial Intelligence (AI)**.

An API is a common feature of applications, especially those that are in a cloud that expands the usefulness of a system and drives the integration.

Recapping the earthquake system

The earthquake monitoring system will be further explained in this chapter. This time, the REST API will be expanded to add new data from another source, and a website will be created in order to provide a user interface for report generation. The API-accepting input opens up a wider variety of data sources, including the generated test data for development environments.

The reports will primarily be in HTML format, all though we will also look at other formats. The **Comma-separated Values (CSV)** format continues to persist as a user requirement on most systems, simply because it is flexible and easy to use in a spreadsheet and in other applications. Let's take a look at how the API client and reporting fit into the overall system:



This project will have many parts when completed, which is typical of many systems. Not to scare any new programmers, but this is actually a fairly simple system. If you don't believe me, consider a modern social network website that receives text, photographs from many sorts of device, and API clients. Now imagine the number of subsystems and management reports it would require!

Advancing the REST API

This API will be far more useful and will provide us with the ability to add data so that the current web data feed is not the only data source.

The sample code in the `georestwebservice` project (of this chapter) is an updated version of the API with the new `recordFeature` method in the `georestwebservice.dart` file:

```
@ApiMethod(path: 'record', method: 'POST')
QuakeResponse recordFeature(QuakeRequest request) {
    DaoQuakeAPI quakeAPI = new DaoQuakeAPI();
    quakeAPI.recordFeature(getFeatureAsJSON(request));

    QuakeResponse quakeResponse = new QuakeResponse();
    quakeResponse.result = "1";

    return quakeResponse;
}
```

This method will use the standard HTTP POST verb in order to receive the input from the client application. As the `rpc` package wraps the entire method and composes and sends error responses, there is no need for error handling in this method. If, for example, something goes wrong while storing a result in the database, the client will receive an error message.

The following `getFeatureAsJSON` function, that is found in the `helpers.dart` file, converts the incoming `QuakeRequest` object into a JSON string:

```
String getFeatureAsJSON(QuakeRequest request) {
    String feature = jsonData;
    feature = feature.replaceAll("MAG", request.magnitude.toString());
    feature = feature.replaceFirst("TIME", request.time.toString());
    feature = feature.replaceFirst("LAT", request.latitude.toString());
    feature = feature.replaceFirst("LONG", request.longitude.
        toString());
    return feature;
}
```

The `jsonData` string is a template for the GeoJSON feature. The `String` class has numerous useful methods that are used to match strings, and these are used to generate the final string that is returned from the function. The `replaceAll` method is used to replace every occurrence of a string; in this case, for the magnitude that appears as a value and in the text description. The `replaceFirst` method is used to replace the first occurrence of a string and is used in this function for the values that appear only once in the string.

The following API method's parameter is a simple class that is declared in the same file that contains four fields:

```
class QuakeRequest {  
  
    @ApiProperty(required: true)  
    int time;  
  
    @ApiProperty(required: true)  
    double magnitude;  
  
    @ApiProperty(required: true)  
    double longitude;  
  
    @ApiProperty(required: true)  
    double latitude;  
}
```

The fields are annotated, which allows the `rpc` package to handle the marshaling of data through the REST interface.

Passing parameters to the API

One method of providing input to a REST API is via the calling URL, and this will be used to get a specified number of entries from the database using the following URL:

`http://127.0.0.1:8080/api/quake/v1/recent/100`

The implementation involves extending the pattern of the path with a curly bracket syntax (these correspond to the `String` parameters in the function):

```
@ApiMethod(path: 'recent/{count}')  
Future<List<String>> recent(String count) async {  
    DaoQuakeAPI quakeAPI = new DaoQuakeAPI();  
    return await quakeAPI.fetchRecent(int.parse(count));  
}
```

The `int.parse` method is used to convert the `count` string into a database query parameter.

Posting on the API

In the `georestwebservice` project in this chapter, the source code is an updated version of the API with the new `recordFeature` method in the `daoapi.dart` file, which is as follows:

```
Future<List<String>> recordFeature(String json) async {
    var dbConn;
    DateTime time = new DateTime.now();
    String featureID = time.millisecondsSinceEpoch.toString();
    List<String> result = new List<String>();

    try {
        dbConn = await connect(uri);

        await dbConn.execute(
            'insert into dm_quakefeatures (qfeat_id, geojson) values (@
qfeat_id, @geojson)',
            {'qfeat_id': featureID, 'geojson': json});
    } catch (exception, stacktrace) {
        print(exception);
        print(stacktrace);
    } finally {
        dbConn.close();
    }
    result.add(featureID);
    return result;
}
```

This method will create a `featureID` string for the incoming feature's details, which is then inserted in to the `dm_quakefeatures` table together with the supplied data from the client. The generated `featureID` string is then returned to the calling application for future reference.

Connecting to an API client

The client will be a command-line application that is used to add new features to the database using the new REST API method.

This project is a command-line application and uses the `dart:io` package, which has similar functionality to `dart:html` to work with an HTTP request, as shown in the following code:

```
main() async {
    print("API client");

    HttpClient client = new HttpClient();
    HttpClientRequest request = await client.postUrl(Uri.parse(apiUrl));
    request.headers.contentType = ContentType.JSON;
    await request.write(JSON.encode(getFeature()));

    HttpClientResponse response = await request.close();
    print("${response.toString()}");

    response.transform(UTF8.decoder).listen((contents) {
        request.close();
        print(contents);
        print("API client - done");
        exit(0);
    });
}
```

The application will post a single-feature JSON string to the API, wait for the response, display the response, and then exit. If multiple features are required, the program can simply be rerun.

Varying the data

The client will provide a generated feature to post to the API when it is run. Rather than having set data, a feature will be randomly generated. To generate random numbers in Dart, we can make use of the `Random` class from `dart:math` as follows:

```
Map getFeature() {  
    var rng = new Random();  
    var now = new DateTime.now();  
    Map feature = new Map();  
    feature['time'] = now.millisecondsSinceEpoch;  
    feature['magnitude'] = rng.nextInt(8) + 1;  
    feature['latitude'] = rng.nextDouble() * 90;  
    feature['longitude'] = rng.nextDouble() * 90;  
    return feature;  
}
```

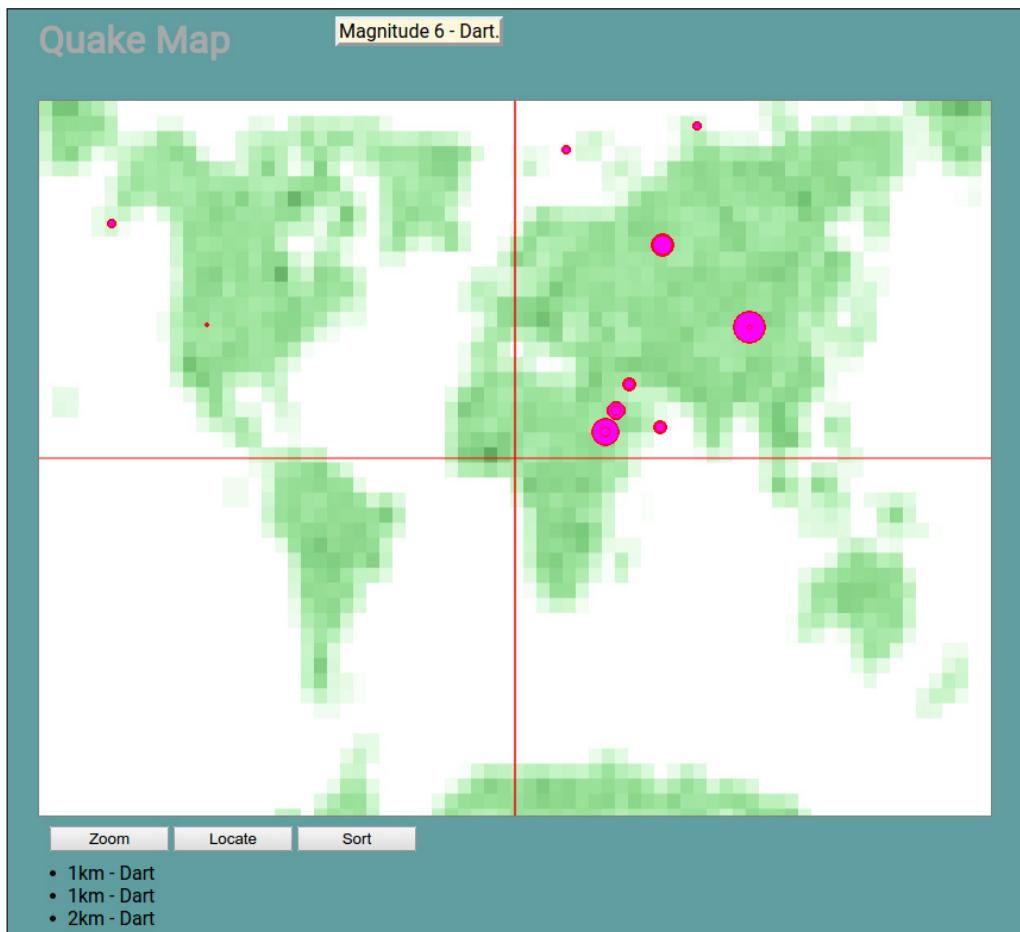
The `Random` class has methods to provide integer and double values, the former allowing the setting of an upper limit. The `nextDouble` method returns a value between 0 and 1, so this value can simply be multiplied by the maximum desired value. The time will be set to the current time.

The `dart:math` package contains a wide range of functions, and if you browse the documentation, you will see that most of the types are not integer or double, but number. In Dart, integer and double are the subtypes of a number object, and the number object implements the basic operators (+, *, / and so on).

The Dart specification contains arbitrary precision integers, which results in a difference in behavior between JavaScript and Dart compiled to JavaScript when dealing with very large numbers. This applies to numbers outside the -253 to 253 range, so this point is probably the only concern that we have for very advanced and unusual applications!

Returning to the map

To view the generated data, the quake map view from the previous chapter can be used. Let's have a look at the visual representation of the generated data in the following screenshot:



Once the command-line API client has been run a number of times, the map will soon be populated. Try adjusting the inputs for the generated data to cover more of the map, as the current numbers are skewed toward the top-right corner.

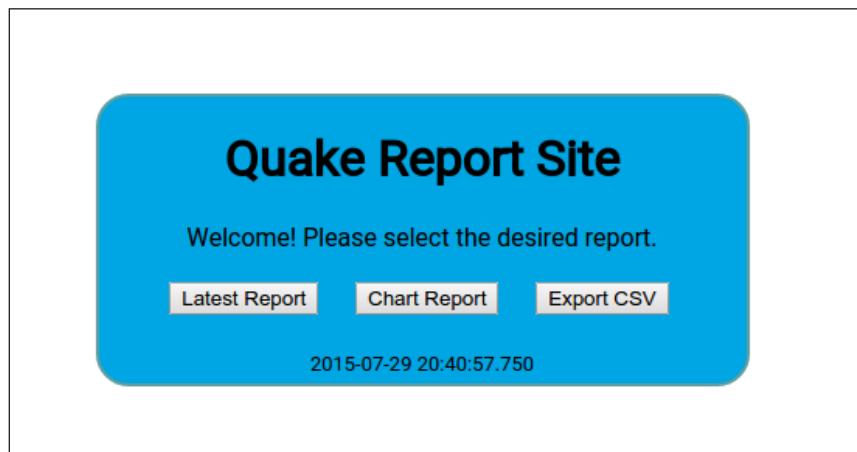
Reporting on the data

There are numerous existing systems that can be used to create reports, Microsoft **SQL Server Reporting Services (SSRS)**, that has a large install base. Due to Dart's support for the most ubiquitous formats, such as XML, and its ability to work with industry standard databases, solutions built with Dart can certainly be integrated.

The reporting feature that we will consider here is a pure Dart implementation. For many applications, reporting even a lightweight reporting feature and an export can greatly increase the usefulness. The added advantage here is the ease of deployment, even in enterprise situations.

The ReportSite project

Users will access the reports via a small website that will allow them to select each report type. This website is implemented in the `ReportSite` project, and the `main.dart` file sets up the initial page with event handlers for the button, as shown in the following figure:



The reports are filled with data from the REST API methods, so ensure that this is running if you wish to view reports. They are rendered in HTML and displayed on the same web page. The following excerpt from the `report_latest.dart` file shows how this is implemented:

```
querySelector('body').setInnerHtml(rep.output.toString(),  
treeSanitizer: new ReportSanitizer());
```

This is a straightforward method call with the HTML string for the page, so why is a second parameter required? This is a security measure to ensure that the user-supplied content does not claim any undesired HTML string or script when being rendered in the display. Consider the following code snippet:

```
class ReportSanitizer implements NodeTreeSanitizer {  
    void sanitizeTree(Node node) {}  
}
```

The `ReportSanitizer` class is a **No Operation (NOP)** implementation, as the HTML is known to be from a trusted source.

Report classes

A report is made of a series of objects in a list, and the page elements are to be based on a similar interface so that the new page elements can be added to the package without having to change the display or export functions.

The page elements are implemented in the `report_elements.dart` file, which is part of the `reports` package project.

The core class will be called `Element`, and as it is to be used as an interface for other classes, it will be declared as follows:

```
abstract class Element {  
    void setContent(var content);  
    String toHtml();  
}
```

This class is declared as `abstract` and provides no implementation for the methods.

To demonstrate this, we will consider the `Title` page element class, which implements the element (the methods have the `@override` annotation, which provides feedback to code analysis tools and other developers that the method is an implementation of a base class) as follows:

```
class Title implements Element {  
  
    String content;  
  
    Title(this.content) {}  
  
    @override  
    void setContent(var content) {  
        String input = content.toString();  
        content = input;  
    }  
}
```

```
}

@Override
String toHtml() {
    return "<h1>$content</h1>";
}

}
```

If a method is not provided an implementation, then the Dart analyzer will warn you of 'Missing concrete implementation'.

The other core page elements follow the same pattern. To create a report, only a few lines of code are required, which are as follows:

```
var rep = new Report('Sample');
rep..addSection(new Title("Sample Report"))
    ..addSection(new Paragraph("This is a paragraph"))
    ..addSection(new Pagebreak())
        ..addSection(new Paragraph("This is a paragraph too"));
```

The `Report` object can be found in the `reports_base.dart` file in the `Reports` project, with its most important `generate` method being generated. This method iterates all the page element objects to build up the HTML report that is stored in the `output` field, as follows:

```
bool generate() {
    output = new StringBuffer();
    allContent.forEach((content) {
        output.write( content.toHtml() );
    });
    generatedTimestamp = new DateTime.now();
    return true;
}
```

A `StringBuffer` object is used to build up the HTML report. This class is more efficient than appending a number of strings.

Strings are stored internally as an immutable format in Dart (as in most programming languages), and updating a string really entails building an entirely new string object. The `StringBuffer` object only creates a `String` object when the `toString` method is called.

Creating a printable report

The data is retrieved from the REST API and displayed in a tabular fashion by using the HTML Table element. In the `performLatest` method, JSON is retrieved from the web service using the following URL:

```
http://127.0.0.1:8080/api/quake/v1/recent/100
```

This returns a JSON list of the requested length (or less, if the database does not have sufficient records), as shown in the following code snippet:

```
[  
  {"\\"properties\\":{\\"mag\\":1.04,\\"place\\":\\"3km SSW of San  
  Bernardino, California\\",\\"time\\":1440707640360,\\"updated\\"  
  :1440707771997,\\"tz\\":-420,\\"url\\":\\"http://earthquake.usgs.gov  
  /earthquakes/eventpage/ci37234439\\",\\"detail\\":\\"  
  http://earthquake.usgs.gov/earthquakes/feed/v1.0/  
  detail/ci37234439.geojson\\",\\"felt\\":null,\\"cdi\\":null,  
  \\"mmi\\":null,\\"alert\\":null,\\"status\\":\\"automatic\\",  
  \\"tsunami\\":0,\\"sig\\":17,\\"net\\":\\"ci\\",\\"code\\":\\"37234439\\",  
  \\"ids\\":\\"ci37234439\\",\\"sources\\":\\"ci\\",\\"types\\":\",  
  general-link,geoserve,nearby-cities,origin,phase-data,  
  scitech-link,\\"},\\"nst\\":27,\\"dmin\\":0.09553,\\"rms\\":0.11,  
  \\"gap\\":62,\\"magType\\":\\"ml\\",\\"type\\":\\"earthquake\\",  
  \\"title\\":\\"M 1.0 - 3km SSW of San Bernardino, California\\",  
  \\"geometry\\":{\\"type\\":\\"Point\\",\\"coordinates\\":[-  
  117.30233,34.1023331,10.84]}}",  
  ...  
]
```

Once the data is retrieved, the report's element objects are created and the HTML for the report details are inserted as a paragraph by using the following code:

```
...  
String json = await HttpRequest.getString(jsonSrc);  
List<String> items = JSON.decode(json);  
DateTime dt = new DateTime.now();  
  
Report rep = new Report('Latest');  
rep  
  ..addSection(new Title("Latest Quake Data Report"))  
  ..addSection(new Paragraph(  
    '<div style="align:center"></div>'))  
  ..addSection(new Paragraph("Report Generated At : ${dt}"))  
  ..addSection(new Pagebreak());  
  
var table = buildFeatureTable(items);  
rep
```

```
..addSection(new Paragraph('<table id="reporttable">' + table +
'</table>'))
..addSection(new Pagebreak())
..addSection(new Notes("Looks like a busy day!"))
..generate();
...
```

The buildFeatureTable loops over the incoming JSON data to create the rows for the table, as shown in the following screenshot:

The screenshot shows a report titled "Latest Quake Data Report". At the top left is a circular logo with a green and blue gradient background, containing the text "Quake Research". Below the logo, the text "Report Generated At : 2015-08-01 16:51:20.045" is displayed. A table titled "Type" is shown below, listing seven entries of earthquake data with columns for Type, Time, Place, and Magnitude.

Type	Time	Place	Magnitude
earthquake	2015-07-28 19:12:04.535	Dart	4
earthquake	2015-07-28 19:12:03.051	Dart	6
earthquake	2015-07-28 19:12:01.735	Dart	2
earthquake	2015-07-28 19:12:00.253	Dart	2
earthquake	2015-07-28 19:11:58.750	Dart	3
earthquake	2015-07-28 19:11:57.152	Dart	5
earthquake	2015-07-28 19:11:53.941	Dart	1

The report is displayed in a continuous manner on screen. Switching to the print preview will show the page breaks between the sections that are created using **Cascading Style Sheets (CSS)**. The `div` element is added to the report with the `page-break-after:always` style by the `Pagebreak` page element.

Charting the data

A good visualization can make a great deal of difference to the person analyzing the data. To achieve this, we can use the `modern_charts` package, which provides a range of chart options. The chart report is implemented in the `report_chart.dart` file in the `ReportSite` project.

This chart will be based on the 20 most recent entries to the earthquake database using the following URL:

```
http://127.0.0.1:8080/api/quake/v1/recent/20
```

The first step to create a chart is to build up a dataset from the incoming JSON string as follows:

```
List dataset = [['Place', 'Magnitude']] ;  
  
items.forEach((String featureJSON) {  
    Map feature = JSON.decode(featureJSON) ;  
    String place = feature['properties']['place'] ;  
    place = place.substring(place.lastIndexOf(",") + 1) ;  
  
    if (place.length > 5) place = place.substring(0, 5) + ".";  
    var mag = feature['properties']['mag'] ;  
    dataset.add([place, mag]) ;  
});  
DataTable table = new DataTable(dataset) ;
```

The first item in the dataset contains the number and names of the series to be plotted. The place name is processed so that the long and specific place names can fit on the axis of the chart. It is then added to the list with the magnitude, as shown in the following code:

```
Map options = {  
    'colors': ['#3333cb'] ,  
    'series': {'labels': {'enabled': true}}}  
};
```

The `options` object contains a range of details for the plotting of the chart, as follows:

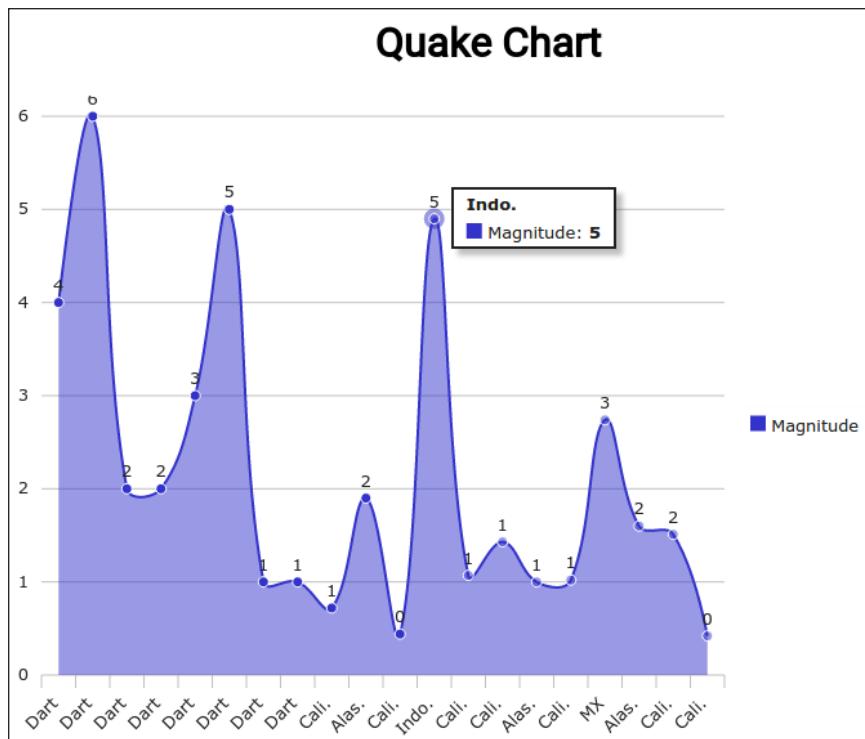
```
DivElement container = newDivElement() ;  
container  
    ..id = "chartContainer"  
    ..style.width = "90%"  
    ..style.height = "90%" ;
```

```
var Title = new HeadingElement.h1();
Title.text = "Quake Chart";
document.body.nodes
    ..clear()
    ..add(Title)
    ..add(container);

LineChart chart = new LineChart(container);
chart.draw(table, options);
```

Once the dataset is ready, the HTML elements for the page can be put together, and the `LineChart` object that is created is given a reference to the object to add it on the page; in this case, a `div` element container (refer to the `modern_chart` package documentation for complete details).

The output of the previous code is shown in the following chart:



You may find working in Dartium to be an odd experience. It is a good browser, but you may prefer your daily web browser, such as Internet Explorer, Firefox, or Chrome, with your carefully-managed settings and collection of extensions. Using **Dart-to-JavaScript (dart2js)** before testing on other browsers interrupts the edit and refresh cycle. Ideally, Dart would run in any web browser.

 Dartium is still going to be around for a long time; however, there is a project underway by the Dart team called the Dart development compiler. This new compiler has two goals, the first being to allow a smooth edit and refresh development experience on all browsers. A modular (incremental) compilation is used to aid a fast edit/refresh cycle. Find out more about the compiler project at https://github.com/dart-lang/dev_compiler.

The second goal of the compiler is to create a human-readable JavaScript format so that the output can easily be debugged. Initially supporting a subset of the Dart language that is statically type checked. The type system has a different approach to dart2js that is to allow more direct and readable mapping of the Dart language to JavaScript. If you are developing in Firefox, then you will want to be able to debug your Dart code in it so that you do not have to step through the obfuscated code.

Exporting to CSV

CSV is a format that just keeps going thanks to its easy import into spreadsheets. The implementation of this report is found in the `report_csv.dart` file, and it retrieves the 50 most recent entries using the following URL:

`http://127.0.0.1:8080/api/quake/v1/recent/50`

 CSV is a text format, and the fields are cleared of any comma characters so as not to have extra delimiters.

Let's consider the following code:

```
performExport(MouseEvent event) async {
    String json = await HttpRequest.getString(jsonSrc);
    List<String> items = JSON.decode(json);
    String csvexport = "Type, Time, Place, Magnitude\r\n";

    items.forEach((String featureJSON) {
        Map feature = JSON.decode(featureJSON);
        String type = feature['properties']['type'];
```

```

String place = feature['properties']['place'];
var mag = feature['properties']['mag'];
var time =
    new DateTime.fromMillisecondsSinceEpoch(feature['properties']
['time']);
String row = "$type, $time, ${place.replaceAll(",","")}, $mag\\
r\\n";
csvexport += row;
});

querySelector('body').setInnerHTML('<pre>$csvexport</pre>',
treeSanitizer: new ReportSanitizer());
}

```

The plain text CSV is set on the page with a `<pre>` HTML tag to ensure that the web browser does not change the intended formatting.

Let's have a look at the following screenshot:

	Type	B	C	D
1	Type	B	C	D
2	earthquake	28/07/15 19:12	Dart	4
3	earthquake	28/07/15 19:12	Dart	6
4	earthquake	28/07/15 19:12	Dart	2
5	earthquake	28/07/15 19:12	Dart	2
6	earthquake	28/07/15 19:11	Dart	3
7	earthquake	28/07/15 19:11	Dart	5
8	earthquake	28/07/15 19:11	Dart	1
9	earthquake	27/07/15 20:40	Dart	1
10	earthquake	26/07/15 17:24	13km NE of Borrego Springs California	0.72
11	earthquake	26/07/15 17:03	87km SW of Homer Alaska	1.9
12	earthquake	26/07/15 17:00	7km W of Cobb California	0.44
13	earthquake	26/07/15 16:22	150km NE of Bitung Indonesia	4.9
14	earthquake	26/07/15 16:45	3km W of Muscoy California	1.07

Exporting this data to a spreadsheet and applying some formatting produces a very usable result for further (human!) analysis.

You may already be aware of some of the existing JavaScript reporting libraries, such as `jspdf`. If the reports package for the quake reports was developed further, could it be used by any web developer? One feature request that the Dart development team has received is the ability to create JavaScript libraries by using Dart. While this is possible by using the JavaScript interoperability, it is not easy to use and develop with Dart for this scenario.



The Dart development compiler is also being developed to allow the creation of JavaScript libraries that can be used seamlessly in non-Dart applications, where the use of Dart is an implementation detail that does not concern the consuming developer and application.

This will give Dart developers the ability to write first-class libraries that target both the Dart world and the larger JavaScript world, giving the library author a larger audience for their code. This also assists in integrating Dart into the existing code bases without transforming the entire project to a new language, which can be critical in risk-averse environments.

Summary

In these past few chapters we have built a powerful geographic application built on a real-world data source.

The API was expanded to provide data input using multiple methods with the powerful standard `rpc` package, and it continued to work with the relational database. The `math` package was used to generate a range of test data in order to push the limits of the application.

The reporting application shows how server-side Dart can be used to process and extract significant data in useful formats. We looked at how Dart can work with databases to produce reports in standard industry formats with the class and inheritance features of the language being used to build an object model for an extensible reporting system.

The continued development of the Dart language via the Dart development compiler may open further possibilities for web applications on the client side and also the server side. The Dart language can rapidly develop full-featured applications that can meet users' needs and provide a solid, dependable, and dynamic platform for developers.

Module 2

Mastering Dart

Master the art of programming high-performance applications with Dart

1

Beyond Dart's Basics

Dart is a very young computer language with many interesting features. Dart is a class-based, object-oriented language with optional types, and it can help you write very powerful programs. In this chapter, we will cover the following topics:

- Modularity and a namespace
- Functions and closures in different scopes
- Classes and mixins
- Methods and operators

Modularity and a namespace

Complex things are the foundation of our world. To understand the complexity of the things around us, it is necessary to understand the parts that make them up. The evolution of complex things is due to functional and behavioral modularity. Functional modularity is the composition of smaller independent components with clear boundaries and functions. Behavioral modularity is mainly about traits and attributes that can evolve independently.

Modularity is nothing new. Earlier, product manufacturers figured out ways to increase the output and quality of the product, while still managing to reduce the cost pressures. They accomplished this through modularity. Modular design can be seen in automotive industry, buildings, and many other industries. Henry Ford introduced the notion of modularity in his assembly line with standardized and interchangeable parts. As a result, he reduced the production cycles and costs to achieve the mass production of his automobiles. A lot of these concepts are still used by many companies today.

Modularity in software development

Representation of complex things as a set of parts is called decomposition. By analogy, the real-world complex software may be broken into functional parts called *modules*. Each module can be created, changed, tested, used, and replaced separately.

Let's take a look at the benefits of modularity. For the sake of simplicity, we divide them into development and postproduction phases. Each of these phases has its own specific tasks to be solved in the scope of that phase.

The development phase has the following benefits:

- Each module requires less code.
- New features or changes can be introduced to modules in isolation, separate from the other modules.
- Errors can be easily identified and fixed in a module.
- Modules can be built and tested independently.
- Programmers writing the modules can collaborate on the same application.
- The same modules can be reused in many applications.
- Applications have a main module and many auxiliary modules. Each module encapsulates a specific functionality and each one is integrated through loosely coupled communication channels provided by the main module.

The postproduction phase has the following benefits:

- Modules kept in a versioning system can be easily maintained and tested
- Fixed and noninfrastructural changes in a module can be done without affecting other modules

One significant disadvantage of modularity is that it increases complexity when managing many modules, especially when each one is individually versioned, updated, and has dependencies on the other modules.

Modularity in Dart

The Dart language was designed by keeping the modules in mind. Modularity in Dart is realized through packages, libraries, and classes.

A **library** exposes functionality as a set of interfaces and hides the implementation from the rest of the world. As a concept, it's very similar to the separation of concern between objects in **object-oriented programming (OOP)**. Separating an application into libraries helps minimize tight coupling and makes it easier to maintain the code. A library can be implemented as a simple function, a single class, several classes, or a collection of parts representing the entire API of a library. The Dart application is a library as well.

A **package** is simply a directory that contains a `pubspec.yaml` file and may include any number of libraries and resources. The `pubspec.yaml` file contains significant information about the package, its authors, and its dependencies on other packages. Here is a sample `pubspec.yaml` file:

```
name: animation_library
version: 0.1.0
author: Sergey Akopkokhyants
description: Animation library for Web application
dependencies:
  browser: any
```

The real `pubspec.yaml` file can have more fields as specified at <https://www.dartlang.org/tools/pub/pubspec.html>. Before a package can be used, it must be published to a package management system, which is available as an online resource called *pub* at <https://pub.dartlang.org/>. To publish and retrieve packages from pub, we use a utility application of the same name. The `pub` utility uses information about dependencies from the `pubspec.yaml` file to retrieve all the necessary packages from the following locations:

- The recently updated packages at <https://pub.dartlang.org/>
- The Git repository
- The directory in the local filesystem

Dart Editor manages dependencies automatically for you. You can publish your packages right in Dart Editor.

Libraries

A **namespace** is a container for all the members of a library. A namespace is defined by the library name. A library that is implicitly named has an empty namespace. This results in a conflict when trying to import libraries with the same namespaces. Import library namespace conflicts can be easily avoided with a prefix clause (`as`) and a name prefix.

The following is an implicitly named library in which all the resources from `dart:html` are made available within the scope of our library with the prefix `dom`:

```
/**  
 * Implicitly named library.  
 * The dart:core library is automatically imported.  
 */  
import 'dart:html' as dom;  
  
/**  
 * Get [Element] by [id].  
 */  
dom.Element getById(String id) => dom.querySelector('#$id');
```

The library namespace make sense only in the Dart environment.



The code that is compiled in JavaScript loses all the library information.



Dart implements encapsulation through privacy. Each member or identifier of a library has one of the two levels of access: private or public. Private members are visible only inside the library in which they are declared. Conversely, members with a public access are visible everywhere. The difference between them is the underscore prefix (`_`), as shown in the following code:

```
// Animation library.  
library animation;  
  
// Class publicly available everywhere.  
class Animation {  
    // ...  
}  
  
// Class visible only inside library.  
class _AnimationLibrary {  
    // ...  
}  
  
// Variable publicly available everywhere.  
var animationSpeed;
```

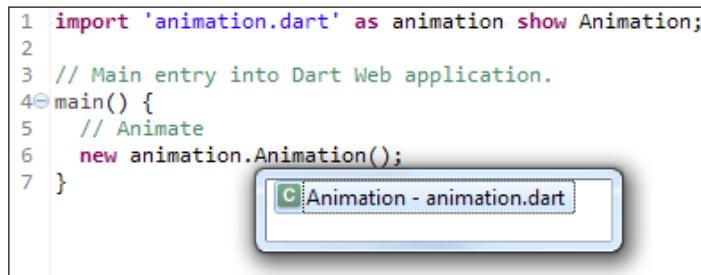
The preceding code shows an animation library with two classes and one variable. The `Animation` class and the `animationSpeed` variable are public, and therefore visible outside the library. The `_AnimationLibrary` class is private and it can be used only in the library.

Public access can be managed with the `show` and `hide` extensions of the `import` statement. Use the following `show` extension with a specific class, which will then be available inside the library in which it is imported:

```
import 'animation.dart' as animation show Animation;

// Main entry into Dart Web application.
main() {
  // Animate
  new animation.Animation();
}
```

The `animation` prefix in the `import` statement defines the namespace to import the `animation.dart` library. All members of the `animation.dart` library are available in the global namespace via this prefix. We are referring to an `Animation` class with the `animation` prefix, as shown here:

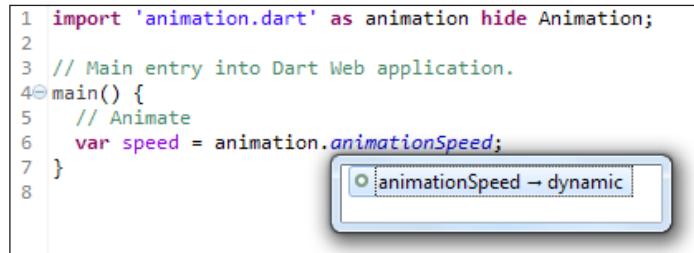


Use the `hide` extension with a specific class, which will then be unavailable inside the library in which it is imported; everything else from the library will be available, as shown in the following code:

```
import 'animation.dart' as animation hide Animation;

// Main entry into Dart Web application.
main() {
  // Animate
  var speed = animation.animationSpeed;
}
```

Now we hide the `Animation` class, but all the other public members in the namespace `animation` are still available, as seen in the following screenshot:



```
1 import 'animation.dart' as animation hide Animation;
2
3 // Main entry into Dart Web application.
4 main() {
5     // Animate
6     var speed = animation.animationSpeed;
7 }
8
```

A callout box highlights the `animationSpeed` variable, which is annotated with `dynamic`. A tooltip shows the type as `dynamic`.

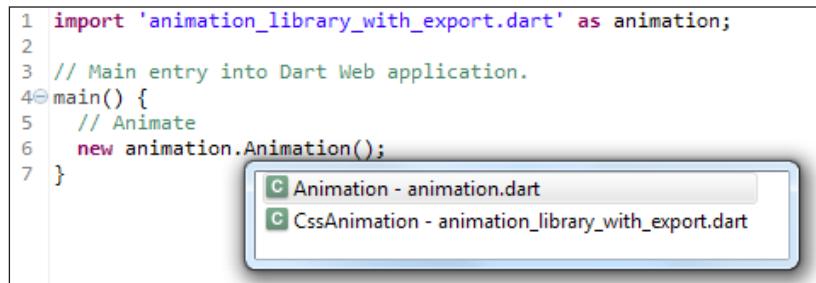
As you can see, the members of the imported library become invisible. This happens because the library exports members from the public namespace. It can be possible to re-export the imported library with the `export` statement if this necessary export statement can be managed with `show` and `hide` as it was for the `import` statement, as shown in the following code:

```
library animation.css;

import 'animation.dart' as animation;
export 'animation.dart' show Animation;

class CssAnimation extends animation.Animation {
    // ...
}
```

The preceding code shows the `animation.css` library. We export the `Animation` class as part of the library namespace. Let's take a look at how we can use them:



```
1 import 'animation_library_with_export.dart' as animation;
2
3 // Main entry into Dart Web application.
4 main() {
5     // Animate
6     new animation.Animation();
7 }
```

A callout box highlights the `Animation` class, which is annotated with `animation.dart`. Another callout box highlights the `CssAnimation` class, which is annotated with `animation_library_with_export.dart`.

There are the exported `Animation` and original `CssAnimation` classes available for use in our main code. Without the `export`, the `Animation` class would be inaccessible in the main code.

Functions and closures in different scopes

I like Dart because everything is an object. Functions are first-class citizens because they support all the operations that are generally available to other types. This means each function have the following properties:

- They can be named by a variable
- They can be passed as an argument to a function
- They can be returned as the result of a function
- They can be stored in data structures
- They can be created in any scope

Let's see where and how can we use functions as usual or as first-class citizens.

Naming functions with a variable

Naming functions by variable means that we can create a reference to a function and assign it to a variable, as shown in the following code:

```
library function_var;

// Returns sum of [a] and [b]
add(a, b) {
    return a + b;
}

// Operation
var operation;

void main() {
    // Assign reference to function [add]
    operation = add;
    // Execute operation
    var result = operation(2, 1);
    print("Result is ${result}");
}
```

Here is the result of the preceding code:

```
Result is 3
```

We have the `add` function and the `operation` variable. We assign the reference of the `add` function to a variable and call the variable as a function later.

Passing a function as an argument to another function

Passing functions as arguments to other functions can be very useful in cases when we need to implement the strategy design pattern to enable the program code to be selected and executed at runtime, as shown in the following code:

```
library function_param;

// Returns sum of [a] and [b]
add(a, b) {
    return a + b;
}

// Operation executor
executor(operation, x, y) {
    return operation(x, y);
}

void main() {
    // Execute operation
    var result = executor(add, 2, 1);
    print("Result is ${result}");
}
```

Here is the result of the preceding code:

```
Result is 3
```

The global executor function from the preceding example can call any function that accepts two arguments. You can see the implementation of the strategy design pattern in the form of anonymous functions passed as parameters of methods in collections.

Returning a function as a result of another function

Sometimes, a function can be returned as a result of another function, as shown in the following code:

```
library function_return;

// Returns sum of [a] and [b]
```

```
add(a, b) => a + b;

// Returns difference between [a] and [b]
sub(a, b) => a - b;

// Choose the function depends on [type]
chooser(bool operation) =>operation ? add : sub;

void main() {
    // Choose function depends on operation type
    var operation = chooser(true);
    // Execute it
    var result = operation(2, 1);
    // Result
    print("Result is ${result}");
}
```

Here is the result of the preceding code:

```
Result is 3
```

This option can be very useful in implementing closures.

Storing a function in data structures

We can store a function in data structures in any collection, as shown in the following code:

```
library function_store;

// Returns sum of [a] and [b]
add(a, b) => a + b;

// Returns difference between [a] and [b]
sub(a, b) => a - b;

// Choose the function depends on [type]
var operations = [add, sub];

void main() {
    // Choose function from list
    var operation = operations[0];
    // Execute it
    var result = operation(2, 1);
    // Result
    print("Result is ${result}");
}
```

Here is the result of the preceding code:

```
Result is 3
```

We have two functions and the array **operations** in our example that stores references to them.

Closures

A function can be created in the global scope or within the scope of another function. A function that can be referenced with an access to the variables in its lexical scope is called a closure, as shown in the following code:

```
library function_closure;

// Function returns closure function.
calculate(base) {
    // Counter store
    var count = 1;
    // Inner function - closure
    return () => print("Value is ${base + count++}");
}

void main() {
    // The outer function returns inner
    var f = calculate(2);
    // Now we call closure
    f();
    f();
}
```

Here is the result of the preceding code:

```
Value is 3
Value is 4
```

We have the `calculate` function, which contains the `count` variable and returns a inner function. The inner function has an access to the `count` variable because both are defined in the same scope. The `count` variable exists only within the scope of `calculate` and would normally disappear when the function exits. This does not happen in this case because the inner function returned by `calculate` holds a reference to `count`. The variable has been closed covered, meaning it's within a closure.

Finally, we know what a **first-class function** is, where we can use them, and how important it is to use closures. Let's move ahead to classes and mixins.

Classes and mixins

We all know it's wasteful trying to reinvent the wheel. It's even more wasteful trying to do it each time we want to build a car. So how can a program code be written more efficiently and made reusable to help us develop more powerful applications? In most cases, we turn to the OOP paradigm when trying to answer this question. OOP represents the concept of objects with data fields and methods that act on that data. Programs are designed to use objects as instances of classes that interact with each other to organize functionality.

Types

The Dart language is dynamically typed, so we can write programs with or without the type annotations in our code. It's better to use the type annotations for the following reasons:

- The type annotations enable early error detection. The static analyzer can warn us about the potential problems at the points where you've made the mistakes.
- Dart automatically converts the type annotations into runtime assertion checks. In the checked mode, the dynamic type assertions are enabled and it can catch some errors when types do not match.
- The type annotations can improve the performance of the code compiled in JavaScript.
- They can improve the documentation making it much easier to read the code.
- They can be useful in special tools and IDE such as the name completion.

The fact that the type annotations were not included in our code does not prevent our program from running. The variables without the type annotations have a dynamic type and are marked with **var** or **dynamic**. Here are several recommendations where the type annotations are appropriate:

- You should add types to public and private variables
- You can add types to parameters of methods and functions
- You should avoid adding types to the bodies of methods or functions

Classes

In the real world, we find many individual objects, all of the same kind. There are many cars with the same make and model. Each car was built from the same set of blueprints. All of them contain the same components and each one is an instance of the class of objects known as `Car`, as shown in the following code:

```
library car;

// Abstract class [Car] can't be instantiated.
abstract class Car {
    // Color of the car.
    String color;
    // Speed of the car.
    double speed;
    // Carrying capacity
    double carrying;

    // Create new [Car] with [color] and [carrying] info.
    Car(this.color, this.carrying);

    // Move car with [speed]
    void move(double speed) {
        this.speed = speed;
    }

    // Stop car.
    void stop() {
        speed = 0.0;
    }
}
```

Objects have methods and instance variables. The `color`, `speed`, and `carrying` are instance variables. All of them have the value `null` as they were not initialized. The instance methods `move` and `stop` provide the behavior for an object and have access to instance variables and the `this` keyword. An object may have getters and setters—special methods with the `get` and `set` keywords that provide read and write access to the instance variables. The `Car` class is marked with the `abstract` modifier, so we can't create an instance of this class, but we can use it to define common characteristics and behaviors for all the subclasses.

Inheritance

Different kinds of objects can have different characteristics that are common with others. Passenger cars, trucks, and buses share the characteristics and behaviors of a car. This means that different kinds of cars inherit the commonly used characteristics and behaviors from the `Car` class. So, the `Car` class becomes the superclass for all the different kinds of cars. We allow passenger cars, trucks, and buses to have only one direct superclass. A `Car` class can have unlimited number of subclasses. In Dart, it is possible to extend from only one class. Every object extends by default from an `Object` class:

```
library passenger_car;

import 'car.dart';

// Passenger car with trailer.
class PassengerCar extends Car {
    // Max number of passengers.
    int maxPassengers;

    // Create [PassengerCar] with [color], [carrying] and
    [maxPassengers].
    PassengerCar(String color, double carrying, this.maxPassengers) :
        super(color, carrying);
}
```

The `PassengerCar` class is not an abstract and can be instantiated. It extends the characteristics of the abstract `Car` class and adds the `maxPassengers` variable.

Interface

Each `Car` class defines a set of characteristics and behaviors. All the characteristics and behaviors of a car define its interface—the way it interacts with the outside world. Acceleration pedal, steering wheel, and other things help us interact with the car through its interface. From our perspective, we don't know what really happens when we push the accelerator pedal, we only see the results of our interaction. Classes in Dart implicitly define an interface with the same name as the class. Therefore, you don't need interfaces in Dart as the abstract class serves the same purpose. The `Car` class implicitly defines an interface as a set of characteristics and behaviors.

If we define a racing car, then we must implement all the characteristics and behaviors of the Car class, but with substantial changes to the engine, suspension, breaks, and so on:

```
import 'car.dart';
import 'passenger_car.dart';

void main() {
    // Create an instance of passenger car of white color,
    // carrying 750 kg and max passengers 5.
    Car car = new PassengerCar('white', 750.0, 5);
    // Move it
    car.move(100.0);
}
```

Here, we just created an instance of PassengerCar and assigned it to the car variable without defining any special interfaces.

Mixins

Dart has a mixin-based inheritance, so the class body can be reused in multiple class hierarchies, as shown in the following code:

```
library trailer;

// The trailer
class Trailer {
    // Access to car's [carrying] info
    double carrying = 0.0;

    // Trailer can carry [weight]
    void carry(double weight) {
        // Car's carrying increases on extra weight.
        carrying += weight;
    }
}
```

The Trailer class is independent of the Car class, but can increase the carrying weight capacity of the car. We use the with keyword followed by the Trailer class to add mixin to the PassengerCar class in the following code:

```
library passenger_car;

import 'car.dart';
```

```
import 'trailer.dart';

// Passenger car with trailer.
class PassengerCar extends Car with Trailer {
    // Max number of passengers.
    int maxPassengers = 4;

    /**
     * Create [PassengerCar] with [color], [carrying] and
     [maxPassengers].
     * We can use [Trailer] to carry [extraWeight].
     */
    PassengerCar(String color, double carrying, this.maxPassengers,
        {double extraWeight:0.0}) : super(color, carrying) {
        // We can carry extra weight with [Trailer]
        carry(extraWeight);
    }
}
```

We added `Trailer` as a mixin to `PassengerCar` and, as a result, `PassengerCar` can now carry more weight. Note that we haven't changed `PassengerCar` itself, we've only extended its functionality. At the same time, `Trailer` can be used in conjunction with the `Truck` or `Bus` classes. A mixin looks like an interface and is implicitly defined via a class declaration, but has the following restrictions:

- It has no declared constructor
- The superclass of a mixin can only be an `Object`
- They do not contain calls to `super`

Well-designed classes

What is the difference between well-designed and poorly-designed classes?
Here are the features of a well-designed class:

- It hides all its implementation details
- It separates its interface from its implementation through the use of abstract classes
- It communicates with other classes only through their interfaces

All the preceding properties lead to encapsulation. It plays a significant role in OOP. Encapsulation has the following benefits:

- Classes can be developed, tested, modified, and used independently
- Programs can be quickly developed because classes can be developed in parallel
- Class optimization can be done without affecting other classes
- Classes can be reused more often because they aren't tightly coupled
- Success in the development of each class leads to the success of the application

All our preceding examples include public members. Is that right? So what is the rule that we must follow to create well-designed classes?

To be private or not

Let's follow the simple principles to create a well-designed class:

- Define a minimal public API for the class. Private members of a class are always accessible inside the library scope so don't hesitate to use them.
- It is not acceptable to change the level of privacy of the member variables from private to public to facilitate testing.
- Nonfinal instance variables should never be public; otherwise, we give up the ability to limit the values that can be stored in the variable and enforce invariants involving the variable.
- The final instance variable or static constant should never be public when referring to a mutable object; otherwise, we restrict the ability to take any action when the final variable is modified.
- It is not acceptable to have the public, static final instance of a collection or else, the getter method returns it; otherwise, we restrict the ability to modify the content of the collection.

The last two principles can be seen in the following example. Let's assume we have a Car class with defined final static list of parts. We can initialize them with Pedal and Wheel, as shown in the following code:

```
class Car {  
    // Be careful with that code !!!  
    static final List PARTS = ['Pedal', 'Wheel'];  
}  
void main() {  
    print('${Car.PARTS}'); // Print: [Pedal, Wheel]
```

```
// Change part
Car.PARTS.remove('Wheel');
print('${Car.PARTS}'); // Print: [Pedal]
}
```

However, there's a problem here. While we can't change the actual collection variable because it's marked as final, we can still change its contents. To prevent anyone from changing the contents of the collection, we change it from final to constant, as shown in the following code:

```
class Car {
    // This code is safe
    static const List PARTS = const ['Pedal', 'Wheel'];
}

void main() {
    print('${Car.PARTS}'); // Print: [Pedal, Wheel]

    // Change part
    Car.PARTS.remove('Wheel');
    print('${Car.PARTS}');
}
```

This code will generate the following exception if we try to change the contents of PARTS:

```
Unhandled exception:
Unsupported operation: Cannot modify an immutable array
#0 List.remove (dart:core-patch/array.dart:327)
...
...
```

Variables versus the accessor methods

In the previous section, we mentioned that nonfinal instance variables should never be public, but is this always right? Here's a situation where a class in our package has a public variable. In our Car class, we have a color field and it is deliberately kept as public, as shown in the following code:

```
// Is that class correct?
class Car {
    // Color of the car.
    String color;
}
```

If the Car class is accessible only inside the library, then there is nothing wrong with it having public fields, because they don't break the encapsulation concept of the library.

Inheritance versus composition

We defined the main rules to follow and create a well-designed class. Everything is perfect and we didn't break any rules. Now, it's time to use a well-designed class in our project. First, we will create a new class that extends the current one. However, that could be a problem as inheritance can break encapsulation.

It is always best to use inheritance in the following cases:

- Inside the library, because we control the implementation and relationship between classes
- If the class was specifically designed and documented to be extended

It's better not to use inheritance from ordinary classes because it's dangerous. Let's discuss why. For instance, someone developed the following `Engine` class to start and stop the general purpose engine:

```
// General purpose Engine
class Engine {
    // Start engine
    void start() {
        // ...
    }

    // Stop engine
    void stop() {
        // ...
    }
}
```

We inherited the `DieselEngine` class from the `Engine` class and defined when to start the engine that we need to initialize inside the `init` method, as shown in the following code:

```
import 'engine.dart';

// Diesel Engine
class DieselEngine extends Engine {
    DieselEngine();

    // Initialize engine before start
    void init() {
        // ...
    }
    void start() {
        // Engine must be initialized before use
    }
}
```

```
    init();
    // Start engine
    super.start();
}
}
```

Then, suppose someone changed their mind and decided that the implementation Engine must be initialized and added the init method to the Engine class, as follows:

```
// General purpose Engine
class Engine {
    // Initialize engine before start
    void init() {
        // ...
    }

    // Start engine
    void start() {
        init();
    }

    // Stop engine
    void stop() {
        // ...
    }
}
```

As a result, the init method in DieselEngine overrides the same method from the Engine superclass. The init method in the superclass is an implementation detail. The implementation details can be changed many times in future from release to release. The DieselEngine class is tightly-coupled with and depends on the implementation details of the Engine superclass. To fix this problem, we can use a different approach, as follows:

```
import 'engine.dart';

// Diesel Engine
class DieselEngine implements Engine {
    Engine _engine;

    DieselEngine() {
        _engine = new Engine();
    }

    // Initialize engine before start
    void init() {
```

```
// ...
}

void start() {
    // Engine must be initialized before use
    init();
    // Start engine
    _engine.start();
}

void stop() {
    _engine.stop();
}
}
```

We created the private `engine` variable in our `DieselEngine` class that references an instance of the `Engine` class. `Engine` now becomes a component of `DieselEngine`. This is called a composition. Each method in `DieselEngine` calls the corresponding method in the `Engine` instance. This technique is called **forwarding**, because we forward the method's call to the instance of the `Engine` class. As a result, our solution is safe and solid. If a new method is added to `Engine`, it doesn't break our implementation.

The disadvantages of this approach are associated performance issues and increased memory usage.

Methods and operators

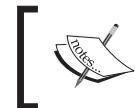
Now that we've introduced well-designed classes, we need to discuss methods.

Checking the values of the parameters before using them

The class constructors, methods, mutators (setters), and operators remove some restrictions on the values that must be passed into their parameters. What will happen if an invalid parameter value is passed to a method? One possibility is that the method will fail with a confusing exception or worse it will succeed but with a wrong result. In any case, it's dangerous not check the parameters of a method before using them. The rule here is to check whether the parameter value is valid as soon as possible. The best place to do that is at the beginning of the method.

The Dart VM can work in a developer-friendly checked mode and a speed-obsessed production mode. We usually use the checked mode when developing our applications. One of the benefits of this mode is the dynamic assertion. We should use the `assert` statement to check whether the parameters of the method are valid before using it. The Dart VM continues the program execution if the Boolean result of the dynamic assertion is `true`, otherwise stops it. This is shown in the following code:

```
/**  
 * Return sum of [a] and [b].  
 * It throws [AssertionError] if any of [a] or [b] equals null  
 */  
sum(int a, int b) {  
    assert(a != null);  
    assert(b != null);  
    return a + b;  
}
```



The `assert` statement has no effect when the program executes in the production mode or is compiled with the JavaScript code.



We must check the validity of the parameters stored in the method for later use. Ignoring this can lead to problems later because an error associated with the parameter can be thrown in a completely different place, making it harder to trace its source. This has serious implications, especially in constructors.

Sometimes, it is important to validate the internal state of a class in the method and generate a special error, as shown in the following code. The typical errors are `StateError`, `RangeError`, and `ArgumentError`.

```
class Car {  
    double petrol;  
  
    /**  
     * Start engine.  
     * That method throws [StateError] if petrol is null  
     * or less than 5 liters.  
     */  
    void startEngine() {  
        if (petrol == null || petrol <= 5.0) {  
            throw new StateError('Not enough petrol');  
        }  
    }  
}
```

Here, we have a `Car` class with the `petrol` variable and the `startEngine` method. The `startEngine` method checks whether there is enough petrol to start the engine; otherwise, it throws an error.



Each time you create a method, think about the restrictions that apply to its parameters.



Well-designed methods

So, now that we've defined well-designed classes, it's time to define well-designed methods. We must remember that methods are part of a class' interface and the following simple rules can make them easier to use and also less error-prone:

- Choose the right method name. Remember, Dart doesn't support method overloading. Instead, we can have different method names or optional parameters.
- Use optional named parameters. This helps programmers to use your methods without the need to remember the position of each parameter.
- Refer to objects in terms of their interfaces over classes as the type of parameters. For example, we have an interface and the class implements that interface. Use the interface as the parameter type of the method instead of a solid one. Don't restrict the solution to a particular implementation.

A car may have the following different types of engines:

```
// Engine interface
abstract class Engine {
    void start();
}

// Diesel engine
class DieselEngine implements Engine {
    void start() {
        // ...
    }
}

// Carburetor engine
class CarburetorEngine implements Engine {
    void start() {
        // ...
    }
}
```

```
}
```

```
// Car
class Car {
    var engine;

    // Car may have any engine
    Car(Engine this.engine);
}
```

It's better to pass the abstract `Engine` class as a parameter of the constructor for the car to prevent any problems in future.

Downloading the example code



You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Summary

This chapter covered some of the most useful advanced features of the Dart language. The Dart language was designed with the modules in mind. Modularity in Dart is realized through packages, libraries, and classes. The code compiled in JavaScript loses all the library information.

Functions are first-class citizens because they support all the operations generally available to other types. A function that can be referenced with an access to the variables in its lexical scope is called a closure.

Programs are designed to use objects as instances of classes that interact with each other to organize functionality. The Dart language is dynamically typed, so we can write programs with or without the type annotations in our code.

In the next chapter, we will talk about generics, errors and exceptions, and annotations and reflection.

2

Advanced Techniques and Reflection

In this chapter, we will discuss the flexibility and reusability of your code with the help of advanced techniques in Dart. Generic programming is widely useful and is about making your code type-unaware. Using types and generics makes your code safer and allows you to detect bugs early. The debate over errors versus exceptions splits developers into two sides. Which side to choose? It doesn't matter if you know the secret of using both. Annotation is another advanced technique used to decorate existing classes at runtime to change their behavior. Annotations can help reduce the amount of boilerplate code to write your applications. And last but not least, we will open Pandora's box through Mirrors of reflection. In this chapter, we will cover the following topics:

- Generics
- Errors versus exceptions
- Annotations
- Reflection

Generics

Dart originally came with **generics** – a facility of generic programming. We have to tell the static analyzer the permitted type of a collection so it can inform us at compile time if we insert a wrong type of object. As a result, programs become clearer and safer to use. We will discuss how to effectively use generics and minimize the complications associated with them.

Raw types

Dart supports arrays in the form of the `List` class. Let's say you use a list to store data. The data that you put in the list depends on the context of your code. The list may contain different types of data at the same time, as shown in the following code:

```
// List of data
List raw = [1, "Letter", {'test': 'wrong'}];
// Ordinary item
double item = 1.23;

void main() {
  // Add the item to array
  raw.add(item);
  print(raw);
}
```

In the preceding code, we assigned data of different types to the `raw` list. When the code executes, we get the following result:

```
[1, Letter, {test: wrong}, 1.23]
```

So what's the problem with this code? There is no problem. In our code, we intentionally used the default `raw` list class in order to store items of different types. But such situations are very rare. Usually, we keep data of a specific type in a list. How can we prevent inserting the wrong data type into the list? One way is to check the data type each time we read or write data to the list, as shown in the following code:

```
// Array of String data
List parts = ['wheel', 'bumper', 'engine'];
// Ordinary item
double item = 1.23;

void main() {
  if (item is String) {
    // Add the item to array
    parts.add(item);
  }
  print(parts);
}
```

Now, from the following result, we can see that the code is safer and works as expected:

```
[wheel, bumper, engine]
```

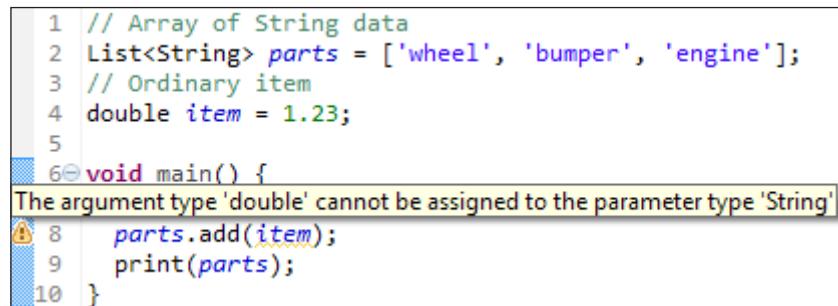
The code becomes more complicated with those extra conditional statements. What should you do when you add the wrong type in the list and it throws exceptions? What if you forget to insert an extra conditional statement? This is where generics come to the fore.

Instead of writing a lot of type checks and class casts when manipulating a collection, we tell the static analyzer what type of object the list is allowed to contain. Here is the modified code, where we specify that `parts` can only contain strings:

```
// Array of String data
List<String> parts = ['wheel', 'bumper', 'engine'];
// Ordinary item
double item = 1.23;

void main() {
    // Add the item to array
    parts.add(item);
    print(parts);
}
```

Now, `List` is a generic class with the `String` parameter. Dart Editor invokes the static analyzer to check the types in the code for potential problems at compile time and alert us if we try to insert a wrong type of object in our collection, as shown in the following screenshot:



The screenshot shows a code editor with the following code:

```
1 // Array of String data
2 List<String> parts = ['wheel', 'bumper', 'engine'];
3 // Ordinary item
4 double item = 1.23;
5
6 void main() {
7     parts.add(item);
8     print(parts);
9 }
10 }
```

A tooltip appears over the line `parts.add(item);` with the message: "The argument type 'double' cannot be assigned to the parameter type 'String'". The line number 8 is highlighted with a blue background.

This helps us make the code clearer and safer because the static analyzer checks the type of the collection at compile time. The important point is that you shouldn't use raw types. As a bonus, we can use a whole bunch of shorthand methods to organize iteration through the list of items to cast safer. Bear in mind that the static analyzer only warns about potential problems and doesn't generate any errors.



Dart checks the types of generic classes only in the check mode.
Execution in the production mode or code compiled to JavaScript
loses all the type information.



Using generics

Let's discuss how to make the transition to using generics in our code with some real-world examples. Assume that we have the following `AssemblyLine` class:

```
part of assembly.room;

// AssemblyLine.
class AssemblyLine {
    // List of items on line.
    List _items = [];

    // Add [item] to line.
    add(item) {
        _items.add(item);
    }

    // Make operation on all items in line.
    make(operation) {
        _items.forEach((item) {
            operation(item);
        });
    }
}
```

Also, we have a set of different kinds of cars, as shown in the following code:

```
part of assembly.room;

// Car
abstract class Car {
    // Color
```

```
    String color;
}

// Passenger car
class PassengerCar extends Car {
    String toString() => "Passenger Car";
}

// Truck
class Truck extends Car {
    String toString() => "Truck";
}
```

Finally, we have the following `assembly.room` library with a `main` method:

```
library assembly.room;

part 'assembly_line.dart';
part 'car.dart';

operation(car) {
    print('Operate ${car}');
}

main() {
    // Create passenger assembly line
    AssemblyLine passengerCarAssembly = new AssemblyLine();
    // We can add passenger car
    passengerCarAssembly.add(new PassengerCar());
    // We can occasionally add Truck as well
    passengerCarAssembly.add(new Truck());
    // Operate
    passengerCarAssembly.make(operation);
}
```

In the preceding example, we were able to add the occasional truck in the assembly line for passenger cars without any problem to get the following result:

```
Operate Passenger Car
Operate Truck
```

This seems a bit far fetched since in real life, we can't assemble passenger cars and trucks in the same assembly line. So to make your solution safer, you need to make the `AssemblyLine` type generic.

Generic types

In general, it's not difficult to make a type generic. Consider the following example of the `AssemblyLine` class:

```
part of assembly.room;

// AssemblyLine.
class AssemblyLine <E extends Car> {
    // List of items on line.
    List<E> _items = [];

    // Add [item] to line.
    add(E item) {
        _items.insert(0, item);
    }

    // Make operation on all items in line.
    make(operation) {
        _items.forEach((E item) {
            operation(item);
        });
    }
}
```

In the preceding code, we added one type parameter, `E`, in the declaration of the `AssemblyLine` class. In this case, the type parameter requires the original one to be a subtype of `Car`. This allows the `AssemblyLine` implementation to take advantage of `Car` without the need for casting a class. The type parameter `E` is known as a bounded type parameter. Any changes to the `assembly.room` library will look like this:

```
library assembly.room;

part 'assembly_line.dart';
part 'car.dart';

operation(car) {
    print('Operate ${car}');
}

main() {
    // Create passenger assembly line
```

```

AssemblyLine<PassengerCar> passengerCarAssembly =
    new AssemblyLine<PassengerCar>();
// We can add passenger car
passengerCarAssembly.add(new PassengerCar());
// We can occasionally add truck as well
passengerCarAssembly.add(new Truck());
// Operate
passengerCarAssembly.make(operation);
}

```

The static analyzer alerts us at compile time if we try to insert the `Truck` argument in the assembly line for passenger cars, as shown in the following screenshot:

A screenshot of a Dart code editor showing a type error. The code is as follows:

```

1 library assembly.room;
2
3 part 'generics_assembly_line.dart';
4 part 'cars.dart';
5
6 operation(car) {
7   print('Operate ${car}');
8 }
9
10 main() {
11   // Create passenger assembly line
12   AssemblyLine<PassengerCar> passengerCarAssembly =
13     new AssemblyLine<PassengerCar>();
14   // We can add passenger car
15   passengerCarAssembly.add(new PassengerCar());
The argument type 'Truck' cannot be assigned to the parameter type 'PassengerCar'
16   passengerCarAssembly.add(new Truck());
17   // Operate
18   passengerCarAssembly.make(operation);
19 }
20

```

The line `17` contains the error message: "The argument type 'Truck' cannot be assigned to the parameter type 'PassengerCar'". The word "Truck" is underlined with a yellow squiggle, and the word "PassengerCar" is underlined with a red squiggle.

After we fix the code in line `17`, all looks good. Our assembly line is now safe. But if you look at the operation function, it is totally different for passenger cars than it is for trucks; this means that we must make the operation generic as well. The static analyzer doesn't show any warnings and, even worse, we cannot make the operation generic directly because Dart doesn't support generics for functions. But there is a solution.

Generic functions

Functions, like all other data types in Dart, are objects, and they have the data type `Function`. In the following code, we will create an `Operation` class as an implementation of `Function` and then apply generics to it as usual:

```
part of assembly.room;

// Operation for specific type of car
class Operation<E> extends Car implements Function {
    // Operation name
    final String name;
    // Create new operation with [name]
    Operation(this.name);
    // We call our function here
    call(E car) {
        print('Make ${name} on ${car}');
    }
}
```

The gem in our class is the `call` method. As `Operation` implements `Function` and has a `call` method, we can pass an instance of our class as a function in the `make` method of the `assembly` line, as shown in the following code:

```
library assembly.room;

part 'assembly.dart';
part 'car.dart';
part 'operation.dart';

main() {
    // Paint operation for passenger car
    Operation<PassengerCar> paint = new
        Operation<PassengerCar>("paint");
    // Paint operation for Trucks
    Operation<Truck> paintTruck = new Operation<Truck>("paint");
    // Create passenger assembly line
    Assembly<PassengerCar> passengerCarAssembly =
        new Assembly<PassengerCar>();
    // We can add passenger car
    passengerCarAssembly.add(new PassengerCar());
    // Operate only with passenger car
    passengerCarAssembly.make(paint);
    // Operate with mistake
    passengerCarAssembly.make(paintTruck);
}
```

In the preceding code, we created the `paint` operation to paint the passenger cars and the `paintTruck` operation to paint trucks. Later, we created the `passengerCarAssembly` line and added a new passenger car to the line via the `add` method. We can run the `paint` operation on the passenger car by calling the `make` method of the `passengerCarAssembly` line. Next, we intentionally made a mistake and tried to paint the truck on the assembly line for passenger cars, which resulted in the following runtime exception:

```
Make paint on Passenger Car
Unhandled exception:
type 'PassengerCar' is not a subtype of type 'Truck' of 'car'.
#0 Operation.call (.../generics_operation.dart:10:10)
#1 Assembly.make.<anonymous
    closure> (.../generics_assembly.dart:16:15)
#2 List.forEach (dart:core-patch/growable_array.dart:240)
#3 Assembly.make (.../generics_assembly.dart:15:18)
#4 main (.../generics_assembly_and_operation_room.dart:20:28)
...
...
```

This trick with the `call` method of the `Function` type helps you make all the aspects of your assembly line generic. We've seen how to make a class generic and function to make the code of our application safer and cleaner.



The documentation generator automatically adds information about generics in the generated documentation pages.

To understand the differences between errors and exceptions, let's move on to the next topic.

Errors versus exceptions

Runtime faults can and do occur during the execution of a Dart program. We can split all faults into two types:

- Errors
- Exceptions

There is always some confusion on deciding when to use each kind of fault, but you will be given several general rules to make your life a bit easier. All your decisions will be based on the simple principle of recoverability. If your code generates a fault that can reasonably be recovered from, use exceptions. Conversely, if the code generates a fault that cannot be recovered from, or where continuing the execution would do more harm, use errors.

Let's take a look at each of them in detail.

Errors

An **error** occurs if your code has programming errors that should be fixed by the programmer. Let's take a look at the following `main` function:

```
main() {
    // Fixed length list
    List list = new List(5);
    // Fill list with values
    for (int i = 0; i < 10; i++) {
        list[i] = i;
    }
    print('Result is ${list}');
}
```

We created an instance of the `List` class with a fixed length and then tried to fill it with values in a loop with more items than the fixed size of the `List` class. Executing the preceding code generates `RangeError`, as shown in the following screenshot:

```
Unhandled exception:
RangeError: 5
#0      List.[]= (dart:core-patch/array.dart:13)
#1      main (file:///C:/Users/sergey/errors vs exceptions/bin/range_error.dart:6:9)
#2      _startIsolate.isolateStartHandler (dart:isolate-patch/isolate_patch.dart:214)
#3      _RawReceivePortImpl._handleMessage (dart:isolate-patch/isolate_patch.dart:122)
```

This error occurred because we performed a precondition violation in our code when we tried to insert a value in the list at an index outside the valid range. Mostly, these types of failures occur when the contract between the code and the calling API is broken. In our case, `RangeError` indicates that the precondition was violated. There are a whole bunch of errors in the Dart SDK such as `CastError`, `RangeError`, `NoSuchMethodError`, `UnsupportedError`, `OutOfMemoryError`, and `StackOverflowError`. Also, there are many others that you will find in the `errors.dart` file as a part of the `dart.core` library. All error classes inherit from the `Error` class and can return stack trace information to help find the bug quickly. In the preceding example, the error happened in line 6 of the `main` method in the `range_error.dart` file.

We can catch errors in our code, but because the code was badly implemented, we should rather fix it. Errors are not designed to be caught, but we can throw them if a critical situation occurs. A Dart program should usually terminate when an error occurs.

Exceptions

Exceptions, unlike errors, are meant to be caught and usually carry information about the failure, but they don't include the stack trace information. Exceptions happen in recoverable situations and don't stop the execution of a program. You can throw any non-null object as an exception, but it is better to create a new exception class that implements the abstract class `Exception` and overrides the `toString` method of the `Object` class in order to deliver additional information. An exception should be handled in a catch clause or made to propagate outwards. The following is an example of code without the use of exceptions:

```
import 'dart:io';

main() {
    // File URI
    Uri uri = new Uri.file("test.json");
    // Check uri
    if (uri != null) {
        // Create the file
        File file = new File.fromUri(uri);
        // Check whether file exists
        if (file.existsSync()) {
            // Open file
            RandomAccessFile random = file.openSync();
            // Check random
            if (random != null) {
                // Read file
                List<int> notReadyContent =
                    random.readSync(random.lengthSync());
                // Check not ready content
                if (notReadyContent != null) {
                    // Convert to String
                    String content = new
                        String.fromCharCodes(notReadyContent);
                    // Print results
                    print('File content: ${content}');
                }
                // Close file
                random.closeSync();
            }
        }
    }
}
```

```
        } else {
            print ("File doesn't exist");
        }
    }
}
```

Here is the result of this code execution:

```
File content: [{ name: Test, length: 100 }]
```

As you can see, the error detection and handling leads to a confusing spaghetti code. Worse yet, the logical flow of the code has been lost, making it difficult to read and understand it. So, we transform our code to use exceptions as follows:

```
import 'dart:io';

main() {
    RandomAccessFile random;
    try {
        // File URI
        Uri uri = new Uri.file("test.json");
        // Create the file
        File file = new File.fromUri(uri);
        // Open file
        random = file.openSync();
        // Read file
        List<int> notReadyContent =
            random.readSync(random.lengthSync());
        // Convert to String
        String content = new String.fromCharCodes(notReadyContent);
        // Print results
        print('File content: ${content}');
    } on ArgumentError catch(ex) {
        print('Argument error exception');
    } on UnsupportedError catch(ex) {
        print('URI cannot reference a file');
    } on FileSystemException catch(ex) {
        print ("File doesn't exist or accessible");
    } finally {
        try {
            random.closeSync();
        } on FileSystemException catch(ex) {
            print("File can't be close");
        }
    }
}
```

The code in the `finally` statement will always be executed independent of whether the exception happened or not to close the `random` file. Finally, we have a clear separation of exception handling from the working code and we can now propagate uncaught exceptions outwards in the call stack.

The suggestions based on recoverability after exceptions are fragile. In our example, we caught `ArgumentError` and `UnsupportedError` in common with `FileSystemException`. This was only done to show that errors and exceptions have the same nature and can be caught any time. So, what is the truth? While developing my own framework, I used the following principle:

If I believe the code cannot recover, I use an error, and if I think it can recover, I use an exception.

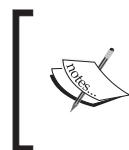
Let's discuss another advanced technique that has become very popular and that helps you change the behavior of the code without making any changes to it.

Annotations

An **annotation** is metadata – data about data. An annotation is a way to keep additional information about the code in the code itself. An annotation can have parameter values to pass specific information about an annotated member. An annotation without parameters is called a marker annotation. The purpose of a marker annotation is just to mark the annotated member.

Dart annotations are constant expressions beginning with the @ character. We can apply annotations to all the members of the Dart language, excluding comments and annotations themselves. Annotations can be:

- Interpreted statically by parsing the program and evaluating the constants via a suitable interpreter
- Retrieved via reflection at runtime by a framework



The documentation generator does not add annotations to the generated documentation pages automatically, so the information about annotations must be specified separately in comments.

Built-in annotations

There are several built-in annotations defined in the Dart SDK interpreted by the static analyzer. Let's take a look at them.

Deprecated

The first built-in annotation is `deprecated`, which is very useful when you need to mark a function, variable, a method of a class, or even a whole class as deprecated and that it should no longer be used. The static analyzer generates a warning whenever a marked statement is used in code, as shown in the following screenshot:

A screenshot of a code editor showing a Dart file. The code includes several `@deprecated` annotations. A tooltip 'KindOfPrice' is deprecated appears over the `KindOfPrice` class definition. The code is as follows:

```
1 part of annotations;
2
3 // Sum [ab] and [b]
4 @deprecated
5 sum(a, b) {
6   return a + b;
7 }
8 * The [KindOfPrice] of product.
9 */
10
11 @deprecated
12 class KindOfPrice {
13   // Kind of price
14 @deprecated
15   String kind;
16
17   // Calculate
18 @deprecated
19   void calculate() {
20     // ...
21   }
22 }
```

Override

Another built-in annotation is `override`. This annotation informs the static analyzer that any instance member, such as a method, getter, or setter, is meant to override the member of a superclass with the same name. The class instance variables as well as static members never override each other. If an instance member marked with `override` fails to correctly override a member in one of its superclasses, the static analyzer generates the following warning:

```

1 part of annotations;
2
3 /**
4  * Type of price.
5 */
6 class PriceType extends KindOfPrice {
7     // Calculate the Price.
8     @override
9     void calculate() {
10         // ...
11     }
12
13     // Try override not exists method
14     @override
Method does not override an inherited method
15     void sum() {
16         // ...
17     }
18 }
19

```

Proxy

The last annotation is proxy. Proxy is a well-known pattern used when we need to call a real class's methods through the instance of another class. Let's assume that we have the following Car class:

```

part of cars;

// Class Car
class Car {
    int _speed = 0;
    // The car speed
    int get speed => _speed;

    // Accelerate car
    accelerate(acc) {
        _speed += acc;
    }
}

```

To drive the car instance, we must accelerate it as follows:

```
library cars;

part 'car.dart';

main() {
    Car car = new Car();
    car.accelerate(10);
    print('Car speed is ${car.speed}');
}
```

We now run our example to get the following result:

```
Car speed is 10
```

In practice, we may have a lot of different car types and would want to test all of them. To help us with this, we created the `CarProxy` class by passing an instance of `Car` in the proxy's constructor. From now on, we can invoke the car's methods through the proxy and save the results in a log as follows:

```
part of cars;

// Proxy to [Car]
class CarProxy {

    final Car _car;
    // Create new proxy to [car]
    CarProxy(this._car);

    @override
    noSuchMethod(Invocation invocation) {
        if (invocation.isMethod &&
            invocation.memberName == const Symbol('accelerate')) {
            // Get acceleration value
            var acc = invocation.positionalArguments[0];
            // Log info
            print("LOG: Accelerate car with ${acc}");
            // Call original method
            _car.accelerate(acc);
        } else if (invocation.isGetter &&
            invocation.memberName == const Symbol('speed')) {
            var speed = _car.speed;
            // Log info
        }
    }
}
```

```

        print("LOG: The car speed ${speed}");
        return speed;
    }
    return super.noSuchMethod(invocation);
}
}

```

As you can see, CarProxy does not implement the Car interface. All the magic happens inside noSuchMethod, which is overridden from the Object class. In this method, we compare the invoked member name with accelerate and speed. If the comparison results match one of our conditions, we log the information and then call the original method on the real object. Now let's make changes to the main method, as shown in the following screenshot:

The screenshot shows a Dart code editor with the following code:

```

1 library cars;
2
3 part 'car.dart';
4 part 'car_proxy.dart';
5
6 main() {
7     Car car = new Car();
8     car.accelerate(10);
9     print('Car speed is ${car.speed}');
10    //
11    CarProxy proxy = new CarProxy(car);
The method 'accelerate' is not defined for the class 'CarProxy'
12    proxy.accelerate(10);
13    // Get car speed through proxy
14    print('Car speed through proxy is ${proxy.speed}');
15
16 }

```

A red box highlights line 11, "CarProxy proxy = new CarProxy(car);", with the error message "The method 'accelerate' is not defined for the class 'CarProxy'" displayed below it.

Here, the static analyzer alerts you with a warning because the CarProxy class doesn't have the accelerate method and the speed getter. You must add the proxy annotation to the definition of the CarProxy class to suppress the static analyzer warning, as shown in the following screenshot:

The screenshot shows the same Dart code as before, but now line 11 includes the @proxy annotation:

```

3 // Proxy to [Car]
4 @proxy
5 class CarProxy {

```

Now with all the warnings gone, we can run our example to get the following successful result:

```

Car speed is 10
LOG: Accelerate car with 10
LOG: The car speed 20
Car speed through proxy is 20

```

Custom annotations

Let's say we want to create a test framework. For this, we will need several custom annotations to mark methods in a testable class to be included in a test case. The following code has two custom annotations. In the case, where we need only marker annotation, we use a constant string test. In the event that we need to pass parameters to an annotation, we will use a Test class with a constant constructor, as shown in the following code:

```
library test;

// Marker annotation test
const String test = "test";

// Test annotation
class Test {
    // Should test be ignored?
    final bool include;
    // Default constant constructor
    const Test({this.include:true});

    String toString() => 'test';
}
```

The Test class has the final `include` variable initialized with a default value of true. To exclude a method from tests, we should pass `false` as a parameter for the annotation, as shown in the following code:

```
library test.case;

import 'test.dart';
import 'engine.dart';

// Test case of Engine
class TestCase {
    Engine engine = new Engine();

    // Start engine
    @test
    testStart() {
        engine.start();
        if (!engine.started) throw new Exception("Engine must start");
    }

    // Stop engine
    @Test()
}
```

```
testStop() {
    engine.stop();
    if (engine.started) throw new Exception("Engine must stop");
}

// Warm up engine
@Test(include:false)
testWarmUp() {
    // ...
}
```

In this scenario, we test the Engine class via the invocation of the `testStart` and `testStop` methods of `TestCase`, while avoiding the invocation of the `testWarmUp` method.

So what's next? How can we really use annotations? Annotations are useful with reflection at runtime, so now it's time to discuss how to make annotations available through reflection.

Reflection

Introspection is the ability of a program to discover and use its own structure. Reflection is the ability of a program to use introspection to examine and modify the structure and behavior of the program at runtime. You can use reflection to dynamically create an instance of a type or get the type from an existing object and invoke its methods or access its fields and properties. This makes your code more dynamic and can be written against known interfaces so that the actual classes can be instantiated using reflection. Another purpose of reflection is to create development and debugging tools, and it is also used for meta-programming.

There are two different approaches to implementing reflection:

- The first approach is that the information about reflection is tightly integrated with the language and exists as part of the program's structure. Access to program-based reflection is available by a property or method.
- The second approach is based on the separation of reflection information and program structure. Reflection information is separated inside a distinct Mirror object that binds to the real program member.

Dart reflection follows the second approach with Mirrors. You can find more information about the concept of **Mirrors** in the original paper written by Gilad Bracha at <http://bracha.org/mirrors.pdf>. Let's discuss the advantages of Mirrors:

- Mirrors are separate from the main code and cannot be exploited for malicious purposes
- As reflection is not part of the code, the resulting code is smaller
- There are no method-naming conflicts between the reflection API and inspected classes
- It is possible to implement many different Mirrors with different levels of reflection privileges
- It is possible to use Mirrors in command-line and web applications

Let's try Mirrors and see what we can do with them. We will continue to create a library to run our tests.

Introspection in action

We will demonstrate the use of Mirrors with something simple such as introspection. We will need a universal code that can retrieve the information about any object or class in our program to discover its structure and possibly manipulate it with properties and call methods. For this, we've prepared the `TypeInspector` class. Let's take a look at the code. We've imported the `dart:mirrors` library here to add the introspection ability to our code:

```
library inspector;

import 'dart:mirrors';
import 'test.dart';

class TypeInspector {
    ClassMirror _classMirror;
    // Create type inspector for [type].
    TypeInspector(Type type) {
        _classMirror = reflectClass(type);
    }
}
```

The `ClassMirror` class contains all the information about the observing type. We perform the actual introspection with the `reflectClass` function of `Mirrors` and return a distinct `Mirror` object as the result. Then, we call the `getAnnotatedMethods` method and specify the name of the annotation that we are interested in. This will return a list of `MethodMirror` that will contain methods annotated with specified parameters. One by one, we step through all the instance members and call the private `_isMethodAnnotated` method. If the result of the execution of the `_isMethodAnnotated` method is successful, then we add the discovering method to the result list of found `MethodMirror`'s, as shown in the following code:

```
// Return list of method mirrors assigned by [annotation].
List<MethodMirror> getAnnotatedMethods(String annotation) {
    List<MethodMirror> result = [];
    // Get all methods
    _classMirror.instanceMembers.forEach(
        (Symbol name, MethodMirror method) {
            if (_isMethodAnnotated(method, annotation)) {
                result.add(method);
            }
        });
    return result;
}
```

The first argument of `_isMethodAnnotated` has the `metadata` property that keeps a list of annotations. The second argument of this method is the annotation name that we would like to find. The `inst` variable holds a reference to the original object in the `reflectee` property. We pass through all the method's metadata to exclude some of them annotated with the `Test` class and marked with `include` equals `false`. All other method's annotations should be compared to the annotation name, as follows:

```
// Check is [method] annotated with [annotation].
bool _isMethodAnnotated(MethodMirror method, String annotation) {
    return method.metadata.any(
        (InstanceMirror inst) {
            // For [Test] class we check include condition
            if (inst.reflectee is Test &&
                !(inst.reflectee as Test).include) {
                // Test must be exclude
                return false;
            }
            // Literal compare of reflectee and annotation
            return inst.reflectee.toString() == annotation;
        });
}
```

Dart Mirrors have the following three main functions for introspection:

- `reflect`: This function is used to introspect an instance that is passed as a parameter and saves the result in `InstanceMirror` or `ClosureMirror`. For the first one, we can call methods, functions, or get and set fields of the `reflectee` property. For the second one, we can execute the closure.
- `reflectClass`: This function reflects the class declaration and returns `ClassMirror`. It holds full information about the type passed as a parameter.
- `reflectType`: This function returns `TypeMirror` and reflects a class, `typedef`, function type, or type variable.

Let's take a look at the main code:

```
library test.framework;

import 'type_inspector.dart';
import 'test_case.dart';

main() {
  TypeInspector inspector = new TypeInspector(TestCase);
  List methods = inspector.getAnnotatedMethods('test');
  print(methods);
}
```

Firstly, we created an instance of our `TypeInspector` class and passed the testable class, in our case, `TestCase`. Then, we called `getAnnotatedMethods` from `inspector` with the name of the annotation, `test`. Here is the result of the execution:

```
[MethodMirror on 'testStart', MethodMirror on 'testStop']
```

The `inspector` method found the methods `testStart` and `testStop` and ignored `testWarmUp` of the `TestCase` class as per our requirements.

Reflection in action

We have seen how introspection helps us find methods marked with annotations. Now we need to call each marked method to run the actual tests. We will do that using reflection. Let's make a `MethodInvoker` class to show reflection in action:

```
library executor;

import 'dart:mirrors';

class MethodInvoker implements Function {
  // Invoke the method
```

```
call(MethodMirror method) {
    ClassMirror classMirror = method.owner as ClassMirror;
    // Create an instance of class
    InstanceMirror inst =
        classMirror.newInstance(new Symbol(''), []);
    // Invoke method of instance
    inst.invoke(method.simpleName, []);
}
```

As the `MethodInvoker` class implements the `Function` interface and has the `call` method, we can call instance it as if it was a function. In order to call the method, we must first instantiate a class. Each `MethodMirror` method has the `owner` property, which points to the `owner` object in the hierarchy. The `owner` of `MethodMirror` in our case is `ClassMirror`. In the preceding code, we created a new instance of the class with an empty constructor and then we invoked the method of `inst` by name. In both cases, the second parameter was an empty list of method parameters.

Now, we introduce `MethodInvoker` to the main code. In addition to `TypeInspector`, we create the instance of `MethodInvoker`. One by one, we step through the methods and send each of them to `invoker`. We print `Success` only if no exceptions occur. To prevent the program from terminating if any of the tests failed, we wrap `invoker` in the try-catch block, as shown in the following code:

```
library test.framework;

import 'type_inspector.dart';
import 'method_invoker.dart';
import 'engine_case.dart';

main() {
    TypeInspector inspector = new TypeInspector(TestCase);
    List methods = inspector.getAnnotatedMethods(test);
    MethodInvoker invoker = new MethodInvoker();
    methods.forEach((method) {
        try {
            invoker(method);
            print('Success ${method.simpleName}');
        } on Exception catch(ex) {
            print(ex);
        } on Error catch(ex) {
            print("$ex : ${ex.stackTrace}");
        }
    });
}
```

As a result, we will get the following code:

```
Success Symbol("testStart")
Success Symbol("testStop")
```

To prove that the program will not terminate in the case of an exception in the tests, we will change the code in `TestCase` to break it, as follows:

```
// Start engine
@test
testStart() {
    engine.start();
    // !!! Broken for reason
    if (engine.started) throw new Exception("Engine must start");
}
```

When we run the program, the code for `testStart` fails, but the program continues executing until all the tests are finished, as shown in the following code:

```
Exception: Engine must start
Success Symbol("testStop")
```

And now our test library is ready for use. It uses introspection and reflection to observe and invoke marked methods of any class.

Summary

This concludes mastering of the advanced techniques in Dart. You now know that generics produce safer and clearer code, annotation with reflection helps execute code dynamically, and errors and exceptions play an important role in finding bugs that are detected at runtime.

In the next chapter, we will talk about the creation of objects and how and when to create them using best practices from the programming world.

3

Object Creation

In this chapter, we will talk about the creation of objects. We will see how and when to create them using best practices from the programming world, and then find a place for them to be accommodated in Dart. The different techniques covered here will help us to make correct choices that will be useful in different business cases. In this chapter, we will cover the following topics:

- A generative constructor
- A constructor with optional parameters
- A named constructor
- A redirecting constructor
- A private constructor
- A factory constructor
- A constant constructor
- Initialization of variables
- Syntactic sugar

Creating an object

A **class** is a blueprint for objects. The process of creating objects from a class is called instantiation. An object can be instantiated with a new statement from a class or through reflection. It must be instantiated before it is used.

A class contains a constructor method that is invoked to create objects from the class. It always has the same name as the class. Dart defines two types of constructors: generative and factory constructors.

A generative constructor

A **generative** constructor consists of a constructor name, a constructor parameter list, either a redirect clause or an initializer list, and an optional body. Dart always calls the generative constructor first when the class is being instantiated, as shown in the following code:

```
class SomeClass {  
    // Default constructor  
    SomeClass();  
}  
  
main() {  
    var some = new SomeClass();  
}
```

If the constructor is not defined, Dart creates an implicit one for us as follows:

```
class AnyClass {  
    // implicit constructor  
}  
  
main() {  
    var some = new AnyClass();  
}
```

The main purpose of a generative constructor is to safely initialize the instance of a class. This initialization takes place inside a class and ensures that the instantiating object is always in a valid state.

A constructor with optional parameters

A constructor is method of a class and has parameters to specify the initial state or other important information about the class. There are required and optional parameters in a constructor. The optional parameters can either be a set of named parameters or a list of positional parameters. Dart doesn't support method overload; hence, the ability to have optional parameters can be very handy.



Dart does not allow you to combine named and positional optional parameters.



Let's take a look at the following `Car` class constructor with optional positional parameters:

```
// Class Car
class Car {
    String color;
    int weight;

    Car([this.color = 'white', this.weight = 1000]);
}
```

We can omit one or all the parameters to create an object of the class as follows:

```
import 'car_optional_parameters.dart';

main() {
    var car = new Car('blue');
    var car2 = new Car();
}
```

Let's take a look at the following `Car` class constructor that uses optional named parameters:

```
// Class Car
class Car {
    String color;
    int weight;

    Car({this.color:'white', this.weight:1000});
}
```

In the following main code, I used the named parameters to create an instance of the object:

```
import 'car_named_parameters.dart';

main() {
    var car = new Car(weight:750, color:'blue');
}
```

So, which kind of optional parameters are better? I recommend the use of named parameters due to the following reasons:

- Here, you need not remember the place of the parameters
- It gives you a better explanation of what the parameters do

A named constructor

Let's say we want to create a `Collection` class to store all our data in one place. We can do it as follows:

```
library collection;

// Collection class
class Collection {
    // We save data here
    List _data;

    // Default constructor
    Collection() {
        _data = new List();
    }

    // Add new item
    add(item) {
        _data.add(item);
    }

    // ...
}
```

Somewhere in the main method, we create the instance of a class and add data to the collection:

```
import 'collection.dart';

var data = [1, 2, 3];

main() {
    var collection = new Collection();
    //
    data.forEach((item) {
        collection.add(item);
    });
}
```

Any chance that my collection will be initialized in this way in the future is high. So, it would be nice to have an initialization method in my Collection class to add a list of data. One of the solutions is to create a constructor with named parameters to manage our initialization. The code creates a new collection from the optional parameter value if specified, as shown in the following code:

```
library collection;

// Collection class
class Collection {
    // We save data here
    List _data;

    // Default constructor with optional [values] or [item].
    Collection({Iterable values:null, String item:null}) {
        if (item != null) {
            _data = new List();
            add(item);
        } else {
            _data = values != null ?
                new List.from(values) :
                new List();
        }
    }

    // Add new item
    add(item) {
        _data.add(item);
    }

    // ...
}
```

This solution has a *right to live*, but only if the number of parameters is small. Otherwise, a simple task initialization of variables results in very complicated code. I have specified two options for the named parameters with a really tangled logic. It is difficult to convey the meaning of what *values* means. A better way is to use named constructors, as shown in the following code:

```
library collection;

// Collection class
class Collection {
    // We save data here
    List _data;
```

```
// Default constructor
Collection() {
    _data = new List();
}

// Create collection from [values]
Collection.fromList(Iterable values) {
    _data = values == null ?
        new List() :
        new List.from(values);
}

// Create collection from [item]
Collection.fromItem(String item) {
    _data = new List();
    if (item != null) {
        add(item);
    }
}
// ...
}
```

The constructor is referred to as *named* because it has a readable and intuitive way of creating objects of a class. There are constructors named `Collection.fromList` and `Collection.fromItem` in our code. Our class may have a number of named constructors to do the simple task of class instantiation, depending on the type of parameters. However, bear in mind that any superclass named constructor is not inherited by a subclass.



The named constructors provide intuitive and safer construction operations.



A redirecting constructor

Constructors with optional parameters and named constructors help us to improve the usability and readability of our code. However, sometimes we need a little bit more. Suppose we want to add values of the map to our collection, we can do this by simply adding the `Collection.fromMap` named constructor as shown in the following code:

```
//...
// Create collection from [values]
Collection.fromList(Iterable values) {
```

```

        _data = values == null ?
            new List() :
            new List.from(values);
    }
    // Create collection from values of [map]
    Collection.fromMap(Map map) {
        _data = map == null ?
            new List() :
            new List.from(map.values());
    }
    // ...
}

```

The preceding method is not suitable because the two named constructors have similar code. We can correct it by using a special form of a generative constructor (redirecting constructor), as shown in the following code:

```

//...
// Create collection from [values]
Collection.fromList(Iterable values) {
    _data = values == null ?
        new List() :
        new List.from(values);
}
// Create collection from values of [map]
Collection.fromMap(Map map) :
    this.fromList(map == null ? [] : map.values);

```

The redirecting constructor calls another generative constructor and passes values of a map or an empty list. I like this approach as it is compact, less error-prone, and does not contain similar code anymore.



The redirecting constructor cannot have a body and initialization list.



A private constructor

I want to mention a couple of things about the private constructor before we continue with our journey. A private constructor is a special generative constructor that prevents a class from being explicitly instantiated by its callers. It is usually used in the following cases:

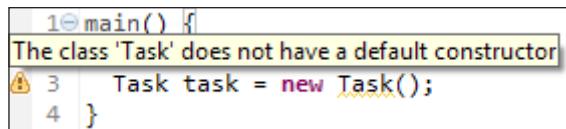
- For a singleton class
- In the factory method

- When the utility class contains static methods only
- For a constant class

Let's define a Task class as follows:

```
class Task {  
    int id;  
  
    Task._();  
  
    Task._internal();  
}
```

In the preceding code, there are private named constructors, `Task._` and `Task._internal`, in the Task class. Also, Dart does not allow you to create an instance of Task as no public constructors are available, as shown in the following screenshot:



The private constructors are used to prevent creating instances of a class.



A factory constructor

A **factory** constructor can only create instances of a class or inherited classes. It is a static method of a class that has the same name as the class and is marked with the factory constructor.



The factory constructor cannot access the class members.



The factory method design pattern

Let's imagine that we are creating a framework and it's time to log the information about different operations of the class methods. The following simple Log class can work for us:

```
library log;

// Log information
abstract class Log {
    debug(String message);
    // ...
}
```

We can print the log information to the console with the `ConsoleLog` class as follows:

```
library log.console;

import 'log.dart';

// Write log to console
class ConsoleLog implements Log {
    debug(String message) {
        // ...
    }
    // ...
}
```

We can also save the log messages in a file with the `FileLog` class as follows:

```
library log.file;

import 'log.dart';

// Write log to file
class FileLog implements Log {
    debug(String message) {
        // ...
    }
    // ...
}
```

Now, we can use the `log` variable to print the debug information to the console in the `Adventure` class, as follows:

```
import 'log.dart';
import 'console_log.dart';

// Adventure class with log
class Adventure {
    static Log log = new ConsoleLog();

    walkMethod() {
        log.debug("entering log");
        // ...
    }
}
```

We created an instance of `ConsoleLog` and used it to print all our messages to the console. There are plenty of classes that will support logging, and we will include code similar to the preceding one in each of them. I can imagine what will happen when we decide to change `ConsoleLog` to `FileLog`. A simple operation can turn into a nightmare, because it might take a lot of time and resources to make the changes. At the same time, we must avoid altering our classes.

The solution is to use the factory constructor to replace the type of code with subclasses, as shown in the following code:

```
library log;

part 'factory_console_log.dart';

// Log information
abstract class Log {

    factory Log() {
        return new ConsoleLog();
    }

    debug(String message);
    // ...
}
```

The factory constructor of the `Log` class is a static method and every time it creates a new instance of `ConsoleLog`. The updated version of the `Adventure` class looks like the following code:

```
import 'factory_log.dart';

// Adventure class with log
class Adventure {
    static Log log = new Log();

    walkMethod() {
        log.debug("entering log");
        // ...
    }
}
```

Now, we are not referring to `ConsoleLog` as an implementation of `Log`, but only using the factory constructor. All the changes from `ConsoleLog` to `FileLog` will happen in one place, that is, inside the factory constructor. However, it would be nice to use different implementations that are appropriate in specific scenarios without altering the `Log` class as well. This can be done by adding a conditional statement in the factory constructor and instantiating different subclasses, as follows:

```
library log;

part 'factory_console_log.dart';
part 'factory_file_log.dart';

// Log information
abstract class Log {

    static bool useConsoleLog = false;

    factory Log() {
        return useConsoleLog ?
            new ConsoleLog() :
            new FileLog();
    }

    debug(String message);
    // ...
}
```

The `useConsoleLog` static variable can be changed programmatically at any point of time to give us a chance to change the logging direction. As a result, we don't change the `Log` class at all.

In our example, the factory constructor is an implementation of the factory method design pattern. It makes a design more customizable and only a little more complicated.

It is always better to use a factory constructor instead of a generative constructor in the following cases:

- It is not required to always return a new instance of a class
- It is required to instantiate any subtype of the return type
- It is essential to reduce the verbosity of creating parameterized type instances of a class

The singleton design pattern

If we want to keep unique information about a user somewhere in our imaginable framework, then we would have to create a `Configuration` class for that purpose, as shown in the following code:

```
library configuration;

// Class configuration
class Configuration {
    // It always keep our [Configuration]
    static final Configuration configuration = new Configuration._();

    // Database name
    String dbName;

    // Private default constructor
    Configuration._();
}
```

The `Configuration` class has a `dbName` variable to keep the database's name and probably a number of other properties and methods as well. It has a private default constructor, so the class cannot be instantiated from other classes. A static variable `configuration` is final and will be initialized only once. All looks good and the standards of implementing the **singleton** design pattern are followed.

We have only one instance of a Configuration class at a point of time and that's the main purpose of the singleton pattern. One disadvantage here is the time of initialization of the configuration variable. This only happens when our program starts. It is better to use the Lazy initialization when it calls the configuration variable for the first time. The following factory constructor comes in handy here:

```
library configuration;

// Class configuration
class Configuration {
    // It always keep our [Configuration]
    static Configuration _configuration;

    // Factory constructor
    factory Configuration() {
        if (_configuration == null) {
            _configuration = new Configuration._();
        }
        return _configuration;
    }

    // Database name
    String dbName;

    // Private default constructor
    Configuration._();
}
```

For now, when we refer to the Configuration class for the first time, Dart will call the factory constructor. It will check whether the private variable configuration was initialized before, create a new instance of the Configuration class if necessary, and only then return an instance of our class. The following are the changes in the framework of the code:

```
import 'factory_configuration.dart';

main() {
    // Access to database name
    new Configuration().dbName = 'Oracle';
    // ...
    print('Database name is ${new Configuration().dbName}');
}
```

We always get the same instance of a Configuration class when we call the factory method in this solution. Factory constructors can be widely used in the implementation of the flyweight pattern and object pooling.

A constant constructor

Let's assume we have the following Request class in our imaginary framework:

```
library request;

// Request class
class Request {
    static const int AWAIT = 0;
    static const int IN_PROGRESS = 1;
    static const int SUCCESS = 2;
    static const int FAULT = 3;

    // Result of request
    int result = AWAIT;

    // Send request with optional [status]
    void send({status:IN_PROGRESS}) {
        // ...
    }
    // ...
}
```

The result variable keeps the status of the last request as an integer value. The Result class has constants to keep all the possible values of the status in one place so that we can always refer to them. This makes the code better in readability and safer too. This technique is called the *enumerated pattern* and is widely used. However, it has a problem when it comes to safety as any other integer value could be assigned to the result variable. This problem makes a class very fragile. Enumerated types would help us to solve this problem, but unfortunately they do not exist in Dart. The solution to this is that we create an enumerated type ourselves with the help of constant constructors as it can be used to create a compile-time **constant**.

We can create an abstract `Enum` class with the entered parameter, as follows:

```
library enumerated_type;

// Enum class
abstract class Enum<T> {
    // The value
    final T value;
```

```
// Create new instance of [T] with [value]
const Enum(this.value);

// Print out enum info
String toString() {
    return "${runtimeType.toString()}" +
        "${value == null ? 'null' : value.toString()}";
}
```

I intentionally used a generic class in the preceding code as I don't know which type of enumeration we will create. For example, to create an enumerated type of `RequestStatus` based on the integer values, we can create a concrete class as follows:

```
import 'enum.dart';

// Enumerated type Status of Request
class RequestStatus<int> extends Enum {

    static const RequestStatus AWAIT = const RequestStatus(0);
    static const RequestStatus IN_PROGRESS = const RequestStatus(1);
    static const RequestStatus SUCCESS = const RequestStatus(2);
    static const RequestStatus FAULT = const RequestStatus(3);

    const RequestStatus(int value) : super(value);
}
```

The `RequestStatus` class extends the `Enum` class and defines the request statuses as static constant members of the class. To instantiate the `RequestStatus` class object, we use `const` instead of the new instantiation expression.



A constant constructor creates compile-time immutable instances of a class.

Let's go back to the `Request` class and modify it with `RequestStatus`, as shown in the following code:

```
library request;

import 'request_status.dart';

// Request class with enum
class Request {
```

```
// Result of request
var result = RequestStatus.AWAIT;

// Send request with optional [status]
void send({status:RequestStatus.IN_PROGRESS}) {
    // ...
}
```

```
}
```

In the preceding code, we used the enumerated type across the whole class. Finally, here is main method in which we use the Request class:

```
import 'request_with_enum.dart';
import 'request_status.dart';

void main() {
    Request request = new Request();
    // ...
    request.send(status:RequestStatus.SUCCESS);
    // ...
    RequestStatus status = request.result;
    //
    switch (status) {
        case RequestStatus.AWAIT:
            print('Result is $status');
            // ...
            break;
    }
}
```

As you can see, the compile-time constants have a wide variety of uses such as default values of variables and constants, default values in method signatures, switch cases, annotations, and enumerators. Moreover, the code uses them to have a better performance and translates them into optimized JavaScript code.

Use cases of the constant constructor have the following restrictions:

- A constant constructor doesn't have a body to prevent any changes of the class state
- All the variables in a class that have a constant constructor must be final as their binding is fixed upon initialization
- The variables from the initialization list of a constant constructor must be initialized with compile-time constants

Initializing variables

Variables reflect the state of a class instance. There are two types of variables, namely, class and instance variables. Class variables are static in Dart. The static variables of a class share information between all instances of the same class. A class provides instance variables when each instance of a class should maintain information separately from others. As we mentioned, the main purpose of a constructor is to initialize the instance of a class in a safe manner. In other words, we initialize instance variables. Let's discuss when and how we should initialize them.

Uninitialized variables in Dart have the value `null` so we should initialize them before using them. The initialization of variables may happen in several places. They are as follows:

- We can assign any value to a variable at the place of declaration
- A variable can be initialized in the body of a constructor
- A variable can be initialized over a constructor parameter
- Initialization can happen in the initialization list of a constructor

Where is the best place to initialize the variables? Noninitialized variables generate null reference runtime exceptions when we try to use them, as shown in the following code:

```
class First {  
    bool isActive;  
  
    doSomething() {  
        if (isActive) {  
            // ...  
        }  
    }  
}  
  
void main() {  
    First first = new First();  
    first.doSomething();  
}
```

The runtime exception will terminate the program execution and display the following error because the `isActive` variable of the `First` class was not initialized:

```
Unhandled exception:  
type 'Null' is not a subtype of type 'bool' of 'boolean expression'.  
#0 First.doSomething (file:///... / no_initialized.dart:5:9)  
#1 main (file:///... / no_initialized.dart:13:19)  
...  
[  The variable must be initialized during the declaration if we are not planning do it in other places. ]
```

Now, let's move ahead. The `First` and `Second` classes are similar to each other with only a few differences, as shown in the following code:

```
class First {  
    bool isActive;  
  
    First(bool isActive) {  
        this.isActive = isActive;  
    }  
}  
  
class Second {  
    bool isActive;  
  
    Second(this.isActive);  
}
```

In the `First` class, we initialize a variable in the body of a constructor; otherwise, the `Second` class initializes a variable via a constructor parameter. Which one is right? The desire to use the `Second` class is obvious because it is compact.



[It is always preferred to use the compact code to initialize the variables via constructor parameters than in a body of the constructor.]

A variable marked `final` is a read-only variable that must be initialized during the instantiation of a class. This means all the final variables must be initialized:

- At the place of declaration
- Over a constructor parameter
- In the initialization list of a constructor

In the following code, we initialize the `isActive` variable at the place of declaration:

```
class First {
    final bool isActive = false;

    First();
}
```

In the `Second` class, we initialize the `isActive` variable via a parameter of the constructor, as follows:

```
class Second {
    final bool isActive;

    Second(this.isActive);
}
```

If the `isActive` final variable is indirectly dependent on the parameter of the constructor, we use the initializer list, as shown in the following code, as Dart does not allow us to initialize the final variable in a body of the constructor:

```
class Third {
    final bool isActive;

    Third(value) :
        this.isActive = value != null;
}
```



The last place to initialize the final variables is in the constructor initializer list.



Syntactic sugar

We talked a lot about the usability of the code, but I could not resist the desire to mention a couple of things about **syntactic sugar**—a syntax that is designed to make a code easier to read or express.

Method call

Dart is an object-oriented programming language by definition. However, sometimes we need a piece of the functional language to be present in our estate. To help us with this, Dart has an interesting feature that may change the behavior of any class instance like a function, as shown in the following code:

```
import 'dart:async';

class Request {
    send() {
        print("Request sent");
    }
}

main() {
    Request request = new Request();
    Duration duration = new Duration(milliseconds: 1000);
    Timer timer = new Timer(duration, (Timer timer) {
        request.send();
    });
}
```

We have a `timer` function that invokes a callback function and sends a request to the server periodically to help organize pull requests, as follows:

```
Request sent
Request sent
...
```

The `call` method added to the class helps Dart to emulate instances of the `Request` class as functions, as shown in the following code:

```
import 'dart:async';

class Request {
    send() {
        print("Request sent");
    }
}

call(Timer timer) {
    send();
}

main() {
    Duration duration = new Duration(milliseconds: 1000);
    Timer timer = new Timer.periodic(duration, new Request());
}
```

So now we invoke the `send` method from the `call` method of the `Request` class. The result of the execution is similar to the result from the preceding example:

```
Request sent
Request sent
...
```



The `call` method allows Dart to execute a class instance as a function.



Cascade method invocation

Dart has quite a large number of innovations to create an application comfortably, but I will mention the one that can help us to write compact code. It is a cascade method invocation. Let's take a look at the ordinary `SomeClass` class in the following code:

```
library some_class;

class SomeClass {
  String name;
  int id;
}
```

Also, we can see the absolutely ordinary object creation:

```
import 'some_class.dart';

void main() {
  SomeClass some = new SomeClass();
  some.name = 'John';
  some.id = 1;
}
```

We created an instance of a class and initialized the instance variables. The preceding code is very simple. A more elegant and compact version of code uses the cascade method invocation, as follows:

```
import 'some_class.dart';

void main() {
  SomeClass some = new SomeClass()
    ..name = 'John'
    ..id = 1;
}
```

In the first line of the `main` method, we create an instance of the `SomeClass` class. At the same time, Dart creates a scope of the `some` variable and invokes all methods located in that scope. The result would be similar to the one that was in the previous code snippet.



Use the cascade method invocation to make the code less verbose to do multiple operations on the members of an object.



Summary

In this chapter, you learned that a constructor is a method that is invoked to create an object from a class. There are generative and factory constructors. The main purpose of a generative constructor is to initialize the instance of a class in a safe manner. The constructor can have required or optional parameters. The optional parameters enable the supply of arguments for only a few parameters from the list of optional parameters.

A constructor can be named. The named constructors provide intuitive and safer construction operations, because named constructors have similar code and some of them can be translated into redirecting constructors. A redirecting constructor calls another constructor that makes the code compact.

Dart supports private constructors. If a class has only private constructors, Dart cannot create an instance of a class. Private constructors are usually used in classes that contain static members, which are only useful in combination with the factory constructor. A factory constructor usually implements the popular factory method and singleton design patterns.

No language can exist without constant variables or constants. Constants play a significant role in programming with Dart and constant constructors can create compile-time constants as instances of a class.

The initialization of variables can happen in several places. Some variables must be initialized during declaration; others can be initialized via constructor parameters. Final variables must be initialized during the instantiation of class. This can happen during a declaration, via constructor parameters or in the constructor initializer list.

This syntax in Dart very often provides a form of syntactic sugar. One of those places is a method call that allows you to execute a class instance as a function. Another one is the cascade method invocation that makes the code less verbose.

In the next chapter, we will discuss advanced technologies to organize asynchronous code execution and learn the best practices to use Futures, Zones, and Isolates in different cases.

4

Asynchronous Programming

In this chapter, we will look at the advanced techniques that help us execute asynchronous code—one of the most important components of Dart. Asynchronous programming is a standard programming paradigm and together with object-oriented principles, it plays an important role in the development of applications. In this chapter, we will cover the following topics:

- Event-driven architecture
- The Dart VM execution model
- Future
- Zone
- Isolates

Call-stack architectures versus event-driven architectures

For a better understanding of asynchronous programming in Dart, we will discuss call-stack and event-driven architectures.

Call-stack architectures

Traditionally, programs are built on the concept of a **call stack**. This concept is pretty straightforward because a program is basically a path of execution and invocation of sequential operations. Every operation can invoke another operation. At the time of invocation, a program creates a context for the callee operation. The caller operation will wait for the callee operation to return and the program will restore the context of it. Finally, the caller continues with their next operation. The callee operation might have executed another operation on its own behalf.

The program creates a call stack to coordinate and manage the context of each call. The basic primitives of this concept are calls. All calls in the program are tightly coupled, because the program knows which operation must be called after the current one and can share the same memory. The call-stack architecture is very popular and pervasive because it is very similar to the architecture of processors.

Event-driven architectures

Event-driven architecture is the exact opposite of the call-stack concept. The basic primitives of this concept are events. The system dispatches events and transmits them among loosely coupled software components and services. The benefits of **event-driven architecture (EDA)** are as follows:

- It helps utilize existing resources efficiently
- It is easy to extend, evolve, and maintain implementation, which reduces the cost of maintenance
- It allows the exchange of events in an asynchronous manner that prevents blocking or waiting in queue
- In event-driven architecture, the producers and consumers are loosely coupled

Interaction between the components is limited to the publisher and one or many consumers. The publisher is free of concurrency issues and synchronization problems. The consumer can be changed at any time as the producers and consumers are loosely coupled.



Event-driven architecture is the right approach to build loosely coupled asynchronous systems.



The Dart VM execution model

Dart relies on event-driven architecture, which is based on a single-threaded execution model with a single **event loop** and two queues. Dart still provides a call stack. However, it uses events to transport context between the producers and consumers. The event loop is backed by a single thread, so no synchronization and locks are required at all.



The event loop blocked with the running operation blocks the entire application.



A combination of the single-threaded execution model and asynchronous operations allows an application to perform more efficiently and is less resource intensive.

The main part of Dart VM is an event loop. Independent pieces of code can register callback functions as event handlers for certain types of events. A callback is the name given to the function that is passed as an argument of another function and is invoked in future after the event occurs in another function. Events from the timer, mouse events, events of input and output, and many others occurring in the system are registered in the event queue. Event loop sequentially processes the queued events by executing the associated callback functions that have been registered.

 Callbacks must be short-running functions to prevent blocking of the event loop.

Dart supports anonymous functions and closures to define callbacks. The closure has a state bind to the callback function. Once the callback is executed, the state is available in the event loop. Callbacks are never executed in parallel because of single-threaded execution, so the occurrence of a deadlock is impossible. Once the callback has been executed, the event-loop fetches the next event from the event queue and applies its callback.

Dart introduced a new term for tasks that must be completed later: **microtasks**. As the name implies, a microtask is a short-running function that does something significantly small, such as updating the state of variables or dispatching a new event. Dart VM provides a special queue for microtasks. The microtasks queue and the events queue process in a single event loop. However, the microtasks queue has higher priority than the events queue. An event loop processes all the microtasks at once, until the queue becomes empty. Then, it moves on to the events queue and processes one event per loop. Using the long-running code in the microtasks queue increases the risk of starving an event queue and can result in the reduction of responsiveness of an application.

 Make sure that the microtasks are extremely small to prevent blocking of the event loop.

Dart VM doesn't expose the event loop and we can't change or manage it. Bear in mind that the sequence of execution of events is predetermined by the events queue. You should also take into account the fact that the time at which the next event will be processed by the event loop is entirely unknown to you.

Synchronous versus Asynchronous code

There is a lot of speculation regarding what is better: synchronous or asynchronous programming. These conversations always end up in the architecture design. So, the important question is what is the difference between synchrony and asynchrony in code designs?

Let's discuss the terms that we will use. Operations are executed serially in the synchronous (**sync**) code; no more, no less. This is very popular because it is simple. The logical flow of the sync code is clear, and we can read and understand it without any significant effort. Let's take a look at the following code snippet:

```
import 'dart:io';

main() {
  try {
    File file = new File("data.txt");
    RandomAccessFile handler = file.openSync();
    List<int> content = handler.readSync(handler.lengthSync());
    String contentAsString = new String.fromCharCodes(content);
    print("Content: $contentAsString");
    handler.closeSync();
  } on FileSystemException catch(e) {
    print(e.message);
  }
}
```

First, we create a `file` reference to `data.txt` on the filesystem. Then, we create a `handler` by opening `file`. Next, the `handler` reads the bytes from the `file` into a `content` variable. Finally, we translate the `content` to a string, print the result, and close the `handler` file. Some operations in this code take more time than others. The file-read operation can be quick because the size of the file is small. If it is bigger, then while reading from the file, our program will wait until it is done. It can take time to translate the content of the file. These operations block the execution of our program; each time-consuming operation has to finish before starting another one. This code is implemented in a sync manner and can be useful while doing simple tasks like this one. However, this approach cannot be applied in complex software. The complex program may have different pieces of code communicating with each other to draw a **User Interface (UI)**, process keyboard input, read information from remote sites, or save information into the files at the same time. So, it's time to discuss the code written in an asynchronous (**async**) fashion. Async code does not wait for each operation to complete; the result of each operation will be handled later when available. Async code uses several important classes from Dart SDK and one of them is `Future`.

Future

Let's change the code from the previous section into `async`, as follows:

```
import 'dart:io';

main() {
    File file = new File("data.txt");
    file.open().then(processFile);
}

processFile(RandomAccessFile file) {
    file.length().then((int length) {
        file.read(length).then(readFile).whenComplete(() {
            file.close();
        });
    });
}

readFile(List<int> content) {
    String contentAsString = new String.fromCharCodes(content);
    print("Content: $contentAsString");
}
```

As you can see, the **Future** class is a proxy for an initially unknown result and returns a value instead of calling a callback function. `Future` can be created by itself or with `Completer`. Different ways of creating `Future` must be used in different cases. The separation of concerns between `Future` and `Completer` can be very useful. On one hand, we can give `Future` to any number of consumers to observe the resolution independently; on the other hand, `Completer` can be given to any number of producers and `Future` will be resolved by the one that resolves it first. `Future` represents the eventual value that is returned from the callback handler, and it can be in one of the following states:

- The incomplete state is an initial state when `Future` is waiting for the result. The `result` field holds a single-linked list of `Future` listeners.
- A completed state with a value as the result.
- A completed state with an error as the result.
- A `Future` class comes in the pending complete or chained state when it is completed or chained to another `Future` class with a success or an error. It will display an error if you try to complete it again.



Future can be completed with a value or an error only once.



A consumer can register callbacks to handle the value or error of the `Future` class. I slightly changed our example to manage exceptions and deliberately used the wrong filename here to throw an exception:

```
//...
main() {
  File file = new File("data1.txt");
  file.open().then(processFile).catchError((error, stackTrace) {
    print("Catched error is $error\n$stackTrace");
  }, test:(error) {
    return error is FileSystemException;
}).whenComplete(() {
  print("File closed");
});
}
//...
```

In the preceding code, we added the `catchError` method to catch errors. Pay attention to the optional `test` parameter of the `catchError` method. This parameter is the function that is called first if `Future` completes with an error, so you have a chance to check if the instance of an error must be handled in the `catchError` method. If optional `test` parameter is omitted, it defaults to a function that always returns true. If the optional `test` parameter returns `true`, the function, specified as the first parameter of `catchError`, is called with the error and possibly stack trace, and the returned `Future` is completed with the result of the call of this function. The resulting exceptions will look like this:

```
Catched error is FileSystemException: Cannot open file, path =
'data1.txt' (OS Error: The system cannot find the file specified.
, errno = 2)
#0      _File.open.<anonymous closure> (dart:io/file_impl.dart:349)
#1      _RootZone.runUnary (dart:async/zone.dart:1082)
//...
File closed
```

If the optional `test` parameter returns `false`, the exception is not handled by the `catchError` method and the returned `Future` class is completed with the same error and stack trace.



The `catchError` method is the asynchronous equivalent of a `catch` block.



Last but not least, the `Future` class has the `whenComplete` method. This method has one parameter that is considered a function, which is always called in the end regardless of the future result (refer to the last statement in the preceding code).



The `whenComplete` method is the asynchronous equivalent of a `finally` block.



Now when we are finished with definitions, let's discuss the different factory constructors of `Future`.

Future and Timer

Let's create a `Future` class containing the result of the calling computation asynchronously with the `run` method of the `Timer` class, as follows:

```
Future calc = new Future(computation);
calc.then((res) => print(res));
```

This `Future` class does not complete immediately. The `Timer` class adds the event to the event queue and executes the computation callback when the event is being processed in the event loop. If the result of the `computation` function throws an error, the returned `Future` is completed with an error. If the `computation` function creates another `Future`, the current one will wait until the new `Future` is completed and will then be completed with the same result.

Future and Microtask

In the following code, the `Future` class is a scheduled task in the microtasks queue, which does not get completed immediately:

```
Future calc = new Future.microtask(computation);
calc.then((res) => print(res));
```

If the result of `computation` throws, the returned `Future` is completed with the error. If `computation` creates another `Future`, the current one will wait until the new `Future` is completed and will then be completed with the same result.

Sync the Future class

It may sound paradoxical, but we can create a sync version of the `Future` class, as follows:

```
Future calc = new Future.sync(computation);
calc.then((res) => print(res));
```

The reason for this is that the `Future` immediately calls the `computation` function. The result of the computation will be returned in the next event-loop iteration.

Future with a value

The Future class can be created with a specified value, as follows:

```
Future valueFuture = new Future.value(true);  
valueFuture.then((res) => print(res));
```

Here, a Future returns specified value in the next event-loop iteration.

Future with an error

The Future class can be created with an error, as follows:

```
try {  
    throw new Error();  
} on Error catch(ex, stackTrace) {  
    Future errorFuture = new Future.error(ex, stackTrace);  
    errorFuture.catchError((err, stack) => print(err));  
}
```

This Future completes with an error in the next event-loop iteration.

Delaying the Future class

Sometimes, it may be necessary to complete Future after a delay. It can be done as follows:

```
Future calc = new Future.delayed(  
    new Duration(seconds:1), computation);  
calc.then((res) => print(res));
```

The Future will be completed after the given duration has passed with the result of the computation function. It always creates an event in the event queue, and the event gets completed no sooner than the next event-loop iteration if the duration is zero or less than zero.



The Future class must not change the completed value or the error to avoid side effects from listeners.



If Future doesn't have a successor, any error could be silently dropped. In preventing these cases, Dart usually forwards the error to the global error handler.

Let's now look at the benefits of the `Future` class:

- It has a consistent pattern to handle callbacks and exceptions
- It is a more convenient way when compared to chain operations
- It is easy to combine `Futures`
- It provides a single control flow to develop web and command-line applications

Now you know why Dart uses `Future` everywhere in its API. The next stop on our journey is **zones**.

Zones

Often, a program generates an uncaught exception and terminates the execution. The commonly occurring exceptions in the program means that the code is broken and must be fixed. However, sometimes exceptions may happen due to errors in communication, hardware faults, and so on. The following is an example of the HTTP server, which is used to demonstrate this problem:

```
import 'dart:io';

main() {
    runServer();
}

runServer() {
    HttpServer
        .bind(InternetAddress.ANY_IP_V4, 8080)
        .then((server) {
            server.listen((HttpRequest request) {
                request.response.write('Hello, world!');
                request.response.close();
            });
        });
}
```

The code in the `main` function can be terminated due to uncaught errors that may happen in the `runServer` function. Termination of the program under those circumstances can be undesirable.

So, how can this problem be solved? We wrap our code within a try/catch block to catch all the uncaught exceptions and it works perfectly, as shown in the following code:

```
main() {
  try {
    runServer();
  } on Error catch(e) {
    // ...
  }
}
```

This solution is universal and can be used in similar situations, so we will generalize it via the creation of a separate wrapper function:

```
wrapper(Function body, {Function onError}) {
  try {
    body();
  } on Error catch(e) {
    if (onError != null) {
      onError(e);
    }
  }
}

main() {
  wrapper(runServer, onError:(e) {
    // ...
  });
}
```

The `body` argument represents any preserved code and is covered within a try/catch block inside `wrapper`. A `wrapper` function uses the `onError` function to handle all the uncaught exceptions. Using a `wrapper` function is a good practice and its use is advised in other such situations. This is the **zone**.



A zone is a configurable execution context that handles uncaught exceptions and asynchronous tasks.



Let's take a look at what zones can do:

- In critical situations, it allows you to handle exceptions properly
- It provides a way to handle multiple async operations in a single group
- It can have an unlimited number of nested zones, which behave like the parent one

Each zone creates a context, some kind of protected area, where the executing code exists. In addition to intercepting uncaught exceptions, zones can have local variables and can schedule microtasks, create one-off or repeating timers, print information, and save a stack trace for debugging purposes.

Simple zone example

Let's transform our code to use a zone instead of the `wrapper` function, as follows:

```
import 'dart:io';
import 'dart:async';

main() {
  runZoned(runServer, onError: (e) {
    // ...
  });
}
```

The `runZoned` function is a code wrapper. By default, the `async` library implicitly creates a root zone and assigns it to a static `current` variable in the `Zone` class. So, we have an active zone that is always available to us inside the `runZoned` function. When the `runZoned` function runs, it forks the new nested zone from root one and executes the `runServer` function inside its context. Use the `fork` method of the current zone to create a new child of this one.

 A zone can be created only through the `fork` method of the current zone.

Zone nesting

Let's say we have to serve static files in our server. So, we will need to read the file and serve it. To do this properly, we fork the nested zone and protect our code with the `runZoned` function, as follows:

```
runServer() {
  HttpServer
    .bind(InternetAddress.ANY_IP_V4, 8080)
    .then((server) {
      server.listen((HttpRequest request) {
        runZoned(() {
          readFile(request.uri.path).then((String context) {
            request.response.write(context);
        });
      });
    });
}
```

```
        request.response.close();
    });
}, onError:(e) {
    request.response.statusCode = HttpStatus.NOT_FOUND;
    request.response.write(e.toString());
    request.response.close();
});
});
}
}

Future<String> readFile(String fileName) {
switch (fileName.trim()) {
case "/":
case "/index.html":
case "/favicon.ico":
    return new Future.sync(() => "Hello, world!");
}
return new Future.sync(() =>
    throw new Exception('Resource is not available'));
}
```

Inside the nested zone, we call the `readFile` function with a resource name and it returns the content. If the resource is not available, `readFile` generates an exception and the program catches it in the `onError` function, which is registered as the zone's error handler. If we don't specify the error handler, the exception will be bubbled up through the zone-nested hierarchy until any parent zone gets caught up in it or reaches a top-level executable and terminates the program.

Zone values

Now, it's time to discuss authentication on our server as some resources may not be available to the general public. We will follow the idea of token-based authentication that relies on a signed token that is sent to the server on each request. We will create a map of `tokens` to remember all the authorized clients, and then fork a new zone for authentication. We will then read the client token from the header that is to be used for authentication. When we get a map of `tokens` from the current zone, we will inject them into the zone via `zoneValues`, as shown in the following code:

```
runServer() {
    HttpServer
    .bind(InternetAddress.ANY_IP_V4, 8080)
```

```
.then((server) {
  Set tokens = new Set.from(['1234567890']);
  server.listen((HttpRequest request) {
    runZoned(() {
      authenticate(request.headers.value('auth-token'));
    }, zoneValues: {'tokens': tokens}, onError: (e) {
      request.response.statusCode = HttpStatus.UNAUTHORIZED;
      request.response.write(e.toString());
      request.response.close();
    });
    runZoned(() {
      readFile(request.uri.path).then((String context) {
        request.response.write(context);
        request.response.close();
      });
    }, onError: (e) {
      request.response.statusCode = HttpStatus.NOT_FOUND;
      request.response.write(e.toString());
      request.response.close();
    });
  });
});
```

The authentication based on the existence of a token within tokens is as follows:

```
authenticate(String token) {  
    Set tokens = Zone.current['tokens'];  
    if (!tokens.contains(token)) {  
        throw new Exception('Access denied');  
    }  
}
```

In the preceding code, we used the zone-local variables to track tokens and authenticate clients. Here, the variables were injected into the zone with the `zoneValues` argument of the `runZoned` function. Our `tokens` variable works like a static variable in the asynchronous context.



The zone-local variables can play the role of static variables that are visible only in the scope of the zone.

Asynchronous Programming

Now check whether our server-side code works as expected. We installed the Postman extension from <http://www.getpostman.com/> to send requests from the Dartium web browser. Our first request to `http://localhost:8080` that we send without auth-token is shown in the following screenshot:

The screenshot shows the Postman interface. At the top, there are tabs for 'Normal', 'Basic Auth', 'Digest Auth', and 'OAuth 1.0'. The URL field contains 'http://localhost:8080/' and the method is set to 'GET'. Below the URL, there are buttons for 'Send', 'Preview', 'Add to collection', and 'Reset'. Under the 'Headers' tab, it shows 'Headers (6)' and 'STATUS 401 Unauthorized TIME 84 ms'. Below the headers, there are buttons for 'Pretty', 'Raw', 'Preview', and 'JSON/XML'. A message box at the bottom displays '1 Exception: Access denied'.

The request was unauthorized because of the absence of auth-token. Let's add it to the HTTP headers and see what happens:

The screenshot shows the Postman interface. The URL field contains 'http://localhost:8080/' and the method is set to 'GET'. In the 'Headers' section, there is a single entry 'auth-token' with the value '1234567890'. Below the headers, there are buttons for 'Send', 'Preview', 'Add to collection', and 'Reset'. Under the 'Headers' tab, it shows 'Headers (6)' and 'STATUS 200 OK TIME 513 ms'. Below the headers, there are buttons for 'Pretty', 'Raw', 'Preview', and 'JSON/XML'. A message box at the bottom displays '1 Hello, world!'

Finally, our request is authorized and returns Hello, world! as a success message.

Zone specifications

Now, we have decided to log information about each server request and authentication. It is not recommended to inject the `log` function in all the possible places. Zones have `print` functions to print messages as a literal string. The `print` function bubbles up the message with the zone-nested hierarchy until a parent zone intercepts it or reaches up to the root zone to print it. So, we only need to override the `print` function in the `ZoneSpecification` class to intercept the message to the logger. We create a new `zoneSpecification` with the `interceptor` function to print and call the `log` function inside, as follows:

```
//...
main() {
    runZoned(runServer(),
        zoneSpecification: new ZoneSpecification(
            print:(self, parent, zone, message) {
                log(message);
            }
        ),
        onError:(e) {
            // ...
        });
}
```

Somewhere down the line, our `log` function logs `message` into a standard `print`, as shown in the following code:

```
log(String message) {
    print(message);
}
```

In the following code, we print the `request path`:

```
runServer() {
    HttpServer
        .bind(InternetAddress.ANY_IP_V4, 8080)
        .then((server) {
            Set tokens = new Set.from(['1234567890']);
            server.listen((HttpRequest request) {
                runZoned(() {
                    Zone.current.print('Resource ${request.uri.path}');
                    authenticate(request.headers.value('auth-token'));
                });
            });
        });
}
```

Asynchronous Programming

Bear in mind that all the interceptor functions expect the following four arguments:

```
print:(Zone self, ZoneDelegate parent, Zone zone, String message)
```

The first three of them are always the same:

- `self`: This argument represents the zone that's handling the callback
- `parent`: This argument furnishes `ZoneDelegate` to the parent zone and we can use it to communicate with the parent zone
- `zone`: This argument is the first to receive the request (before the request is bubbled up)

The fourth argument always depends on the function. In our example, it is the message that will be printed.



Let's request the `index.html` file via the Postman extension to check this code, as shown in the following screenshot:

A screenshot of the Postman extension for a web browser. The interface includes a header bar with tabs for 'Normal', 'Basic Auth', 'Digest Auth', 'OAuth 1.0', and 'No environment'. Below the header, there are fields for the URL ('http://localhost:8080/index.html'), method ('GET'), and various parameters ('URL params' and 'Headers (1)'). A single header entry 'auth-token' with value '1234567890' is listed. At the bottom of the header section are buttons for 'Send', 'Preview', 'Add to collection', and 'Reset'. The main body of the window shows a status bar with 'STATUS 200 OK' and 'TIME 469 ms'. Below this are tabs for 'Body', 'Headers (6)', 'Raw', 'Pretty', 'Preview', 'JSON', and 'XML'. The 'Pretty' tab is selected, displaying the response body: '1 Hello, world!'. The entire screenshot is framed by a thick black border.

The following result will be displayed in the console log:

```
Resource /index.html
```

Finally, all works as expected.

Interaction between zones

Let's see how the communication between the parent and the nested zones can be useful in a server example. Suppose you want to have more control on the print content of static pages, you can use the following code:

```
//...
Set tokens = new Set.from(['1234567890']);
bool allowPrintContent = false;
server.listen((HttpRequest request) {
    runZoned(() {
//...
});
    runZoned(() {
        readFile(request.uri.path).then((String context) {
            Zone.current.print(context);
            request.response.write(context);
            request.response.close();
        });
    }, zoneValues: {'allow-print':allowPrintContent},
    zoneSpecification: new ZoneSpecification(
        print: (Zone self, ZoneDelegate parent, Zone zone,
String message) {
            if (zone['allow-print']) {
                parent.print(zone, message);
            }
        },
        onError:(e) {
//...
```

We add a Boolean variable `allowPrintContent` to manage the print operation.

We call the `print` function of the zone to print the content of the page when processing Future of `readFile`. We inject `allowPrintContent` as a value of the `allow-print` key of `zoneValues`, and finally, inside the overridden `print` function, we add a condition that allows us to print the page content only if `allow-print` is true.

We requested the `index.html` file via the Postman extension again and see the following result in the console:

```
Resource /index.html
Hello, world!
```

As expected, our code prints the information that comes from both the nested zones. Now, we change the value to `false` and restart the server. The following request only prints the message from the first zone:

```
Resource /index.html
```

Interaction between zones can be easily organized via the zone variables.

Tracking the zone execution

The server listener contains two zones. The first one is used to authenticate the files and the second one is used to read the content of the static files and send them back to the client. It is quite interesting to know how long each static page takes to load and process. Zones support several `run` methods to execute a given function in the zone. We can override the `run` method in `ZoneSpecification` to count the time spent by the request processing function. We use `Stopwatcher` as the timer in our example. We are processing each request and print profiling the time just after sending the response back to the client, as shown in the following code:

```
//...
runServer() {
    HttpServer
    .bind(InternetAddress.ANY_IP_V4, 8080)
    .then((server) {
        Set tokens = new Set.from(['1234567890']);
        bool allowPrintContent = true;
        Stopwatch timer = new Stopwatch();
        server.listen((HttpRequest request) {
            runZoned(() {
                //...
            });
            runZoned(() {
                readFile(request.uri.path).then((String context) {
                    Zone.current.print(context);
                    request.response.write(context);
                    request.response.close();
                    Zone.current.print(

```

```
        "Process time ${timer.elapsedMilliseconds} ms");
    });
}, zoneValues: {'allow-print':allowPrintContent},
zoneSpecification: new ZoneSpecification(
    print: (Zone self, ZoneDelegate parent, Zone zone, String
message) {
        if (zone['allow-print']) {
            parent.print(zone, message);
        }
    },
    run: (Zone self, ZoneDelegate parent, Zone zone, f)
        => run(parent, zone, f, timer)
),
onError:(e) {
    request.response.statusCode = HttpStatus.NOT_FOUND;
    request.response.write(e.toString());
    request.response.close();
}),
});
});
```

Now, we override the `run` function in `zoneSpecification` to call a global `run` function with `timer`, as follows:

```
run(ZoneDelegate parent, Zone zone, Function f, Stopwatch timer) {
    try {
        timer.start();
        return parent.run(zone, f);
    } finally {
        timer.stop();
    }
}
```

In the global run function, we perform a trick when we call the original function from the parent zone delegate. We intend to wrap the function with a try/finally block to stop the timer before returning the result to the zone. Let's request the same resource again, as follows:

```
Resource /index.html
Hello, world!
Process time 54 ms
```

Now, we have the profiling information per request processed on the server. In addition to the standard `run` function, the zone has the `runUnary` and `runBinary` functions to pass one or two extra arguments to execute the given function inside a zone.

Isolates

Now, it's time to discuss the performance of our server. We use an HTTP benchmarking tool such as **wrk** (<https://github.com/wg/wrk>) by Will Glozer to help us in our investigation. To avoid confusion, we will take the simplest version of our server, as follows:

```
import 'dart:io';

main() {
  HttpServer
    .bind(InternetAddress.ANY_IP_V4, 8080)
    .then((server) {
      server.listen((HttpRequest request) {
        // Response back to client
        request.response.write('Hello, world!');
        request.response.close();
      });
    });
}
```

We use this code with a benchmarking tool and keep the 512 concurrent connections open for 30 seconds, as shown in the following code:

```
./wrk -t1 -c256 -d30s http://127.0.0.1:8080
```

Here is the result of the preceding code:

```
Running 30s test @ http://127.0.0.1:8080
  1 threads and 256 connections
  Thread Stats      Avg      Stdev      Max  +/- Stdev
    Latency    33.89ms   24.51ms  931.37ms  99.76%
    Req/Sec     7.63k    835.29     9.77k   89.93%
  225053 requests in 30.00s, 15.02MB read
  Requests/sec: 7501.81
  Transfer/sec: 512.82KB
```

The test shows that our server can process close to 7,500 requests per second. Actually, this is not too bad. Can this value be improved? The key issue is that all this work is handled by a single thread:

- A single thread has the code to handle all the clients that appear in one place
- All the work will run sequentially on one thread

If the total work saturates the core, then the additional work will strangle and slow down the responsiveness of the server for all the clients as later requests queue up and wait for the previous work to be completed. **Isolates** can solve this problem and run several instances of the server in different threads. We will continue to improve our server and use the `ServerSockets` feature that came with the Dart 1.4 release. We will use the references of `ServerSocket` to run multiple instances of our server simultaneously. Instead of creating an instance of `HttpServer`, we create `ServerSocket` with the same initial parameters that we used before.

First of all, we need to create `ReceivePort` in the main thread to receive hand-shaking and usual messages from the spawned isolates. We create as many isolates as we can depending on the number of processors we have. The first parameter of the `spawn` static method of the `Isolate` class is a `global` function that helps organize hand-shaking between the main thread and spawned isolate. The second parameter is `port`, which is used as a parameter in the `global` function. The same port is used to send messages from spawned isolates to the main thread. Now, we need to listen to the messages from the spawned isolates. The spawned isolate follows the hand-shaking process and all the sent messages with `SendPort` are listened to in the main thread. On the completion of the hand-shaking procedure, we create and send an instance of `ServerTask`. All other messages will come as a string to be printed out on the console, as shown in the following code:

```
import 'dart:isolate';
import 'dart:io';

main() {
  ServerSocket
    .bind(InternetAddress.ANY_IP_V4, 8080)
    .then((ServerSocket server) {
      // Create main ReceivePort
      ReceivePort receivePort = new ReceivePort();
      // Create as much isolates as possible
      for (int i = 0; i < Platform.numberOfProcessors; i++) {
        // Create isolate and run server task
        Isolate.spawn(runTask, receivePort.sendPort);
      }
      // Start listening messages from spawned isolates
      receivePort.listen((msg) {
        // Check what the kind of message we received
        if (msg is SendPort) {
          // There is hand-shaking message.
          // Let's send ServerSocketReference and port
          msg.send(new ServerTask(
            server.reference, receivePort.sendPort));
        } else {
      
```

Asynchronous Programming

```
// Usual string message from spawned isolates
print(msg);
}
});
}
}

/***
 * Global function helps organize hand-shaking between main
 * and spawned isolate.
 */
void runTask(SendPort sendPort) {
    // Create ReceivePort for spawned isolate
    ReceivePort receivePort = new ReceivePort();
    // Send own sendPort to main isolate as response on hand-shaking
    sendPort.send(receivePort.sendPort);
    // First message comes from main contains a ServerTask instance
    receivePort.listen((ServerTask task) {
        // Just execute our task
        task.execute();
    });
}

/***
 * Task helps create ServerSocket from ServerSocketReference.
 * We use new instance of ServerSocket to create new HttpServer
 * which starts listen HttpRequests and sends requested path into
 * main's ReceivePort.
 */
class ServerTask {
    ServerSocketReference reference;
    SendPort port;

    ServerTask(this.reference, this.port);

    execute() {
        // Create ServerSocket
        reference.create().then((serverSocket) {
            // Create HttpServer and start listening income HttpRequests
            new HttpServer.listenOn(serverSocket)
                .listen((HttpRequest request) {
                    // Send requested path into main's ReceivePort
                    port.send(request.uri.path);
                    // Response back to client
                    request.response.write("Hello, world");
                    request.response.close();
                });
        });
    }
}
```

Our code is clear enough and potentially faster with isolates. The program is clearer because the code to handle each request is nicely wrapped up in its own function and is faster because each `SocketServer` instance keeps different connections asynchronous and independent; the work on one connection doesn't have to wait to be processed sequentially behind work on another connection. In general, this gives a better responsiveness even on a single-core server. In practice, it delivers better scalability under the load on servers that do have parallel hardware. Now, run the tests and we will see a significant improvement in our server:

```
./wrk -t1 -c256 -d30s http://127.0.0.1:8080
```

The following is the result of the preceding code:

```
Running 30s test @ http://127.0.0.1:8080
  1 threads and 256 connections
  Thread Stats      Avg      Stdev     Max   +/- Stdev
    Latency     10.31ms    6.11ms  50.81ms  73.78%
    Req/Sec    24.01k     2.00k   28.32k  67.52%
  709163 requests in 30.00s, 46.67MB read
  Requests/sec: 23638.95
  Transfer/sec:   1.56MB
```

Our server can process close to 24,000 requests per second in a concurrency-enabled environment. Fantastic!

So, after that quick dive into the world of concurrency, let's discuss isolates in general. Just repeat the best practices of the async programming:

- The program is driven by the queued events coming in from different independent sources
- All the pieces of the program must be loosely coupled

Isolate is a process that builds around the model of servicing a simple FIFO messaging queue. It does not share memory with other isolates and all isolates communicate by passing messages, which are copied before they are sent. As you can see, the implementation of isolates follows the same main principles as async programming.



Always set the receiver port to the main isolate if you need to receive messages from other isolates or send them to each other.



Summary

You now have a better understanding of event-driven architecture, which is one of the key concepts of Dart VM. Event-driven architecture is the right approach to build loosely coupled asynchronous systems.

Dart relies on event-driven architecture based on a single-threaded execution model with a single event loop and two queues. The event loop is backed by a single thread, so no synchronization or locks are required at all. When the event loop is blocked with an operation, this blocks the entire application. A combination of single-threaded execution models and asynchronous operations allows an application to be more productive and less resource intensive.

`Future` is a proxy for an initially unknown result that returns as a value instead of calling a callback function. `Future` almost always adds an event or microtask into the queue that is being processed in the event loop. `Future` can be completed with a value or error only once.

`Zones` implement the best practices of a configurable code wrapper to handle the uncaught errors. `Zones` can have local variables and can schedule microtasks, create one-off or repeating timers, print information, and save a stack trace for debugging purposes.

An `isolate` is a process built around the model of servicing simple messaging queue. It does not share the memory with other isolates and all isolates communicate by passing messages, which are copied before they are sent.

In the next chapter, we will see the stream framework and show when and how to properly use it.

5

The Stream Framework

In this chapter, we will talk about streams. Streams have existed since the early days of UNIX. They have proven to be a dependable way to compose large systems out of small components, which does one thing well. Streams restrict the implementation of a surface area into a consistent interface that can be reused. You can plug the output of one stream as the input to another and use libraries that operate abstractly on streams to institute high-level flow control. Streams are an important component of small program design and have important abstractions that are worth considering. In this chapter, we will cover the following topics:

- Single-subscription streams versus broadcast streams
- The stream framework API

Why you should use streams

Just imagine that you need to provide a file on a server in response to a client's request. The following code will do this:

```
import 'dart:io';

main() {
  HttpServer
    .bind(InternetAddress.ANY_IP_V4, 8080)
    .then((server) {
      server.listen((HttpRequest request) {
        new File('data.txt').readAsString()
          .then((String contents) {
            request.response.write(contents);
            request.response.close();
          });
      });
    });
}
```

In the preceding code, we read the entire file and buffered it into the memory of every request before sending the result to the clients. This code works perfectly for small files, but what will happen if the `data.txt` file is very large? The program will consume a lot of memory because it serves a lot of users concurrently, especially on slow connections. One big disadvantage of this code is that we have to wait for an entire file to be buffered in memory before the content can be submitted to the clients.

The `HttpServer` object listens for the HTTP request, which is a Stream. The `HttpServer` object then generates an `HttpRequest` object and adds it to the stream. The body of the request that is delivered by an `HttpRequest` object is a stream of byte lists. An `HttpRequest` object provides you with an access to the `response` property associated with an `HttpResponse` object. We will write the content of a file into the body of the `HttpResponse` object. The fact that the `HttpRequest` and `HttpResponse` classes are streams means that we can write our example in a better way, as shown in the following code:

```
import 'dart:io';

main() {
    HttpServer
        .bind(InternetAddress.ANY_IP_V4, 8080)
        .then((server) {
            server.listen((HttpRequest request) {
                new File('data.txt')
                    .openRead()
                    .pipe(request.response);
            });
        });
}
```

The `openRead` method creates a new independent Stream Instance of byte lists for the content of this file. The `pipe` method binds this stream as the input of the provided stream consumer and listens for the `data`, `error`, or `done` events. The preceding code has the following benefits:

- The code is cleaner, so you don't need to remember how to push data through the nonstreaming API
- The file streams to clients in chunks, one chunk at a time as soon as they are received from the disk
- The server doesn't need buffer chunks of a file in the memory when the remote clients are on a slow or high-latency connection, because the stream handling backs the pressure automatically

Streams make development simple and elegant.

Single-subscription streams versus broadcast streams

The Dart stream framework supports the single-subscription and broadcast streams in order to have different approaches depending on the required solution. Let's see the difference between single-subscription and broadcast streams.

A single-subscription stream

A stream that allows you to have only one listener during the entire lifetime is called a single-subscription stream. You are not allowed to cancel and subscribe to the same stream again. The newly created stream starts generating events only after the subscription starts listening to them. It stops generating events after the subscription is canceled even if the stream can provide more. The single-subscription stream always delivers each event in the correct order to a listener.



Use single-subscription streams when the event delivery and its order are important for your solution.



A good usage example of a single-subscription stream is getting data from the file or server.

A broadcast stream

A stream that allows you to have multiple listeners during the entire lifetime is called the broadcast stream. You are allowed to subscribe or cancel the subscription at any time. A broadcast stream starts to generate events immediately after the creation if it ready, independent of whether any subscription is registered or not. This fact means that some portion of the events can be lost in a moment when no listener is registered. There is no guarantee that multiple listeners will get the event in the same order that they were registered. A broadcast stream only guarantees that each listener will get all the events in the correct order.



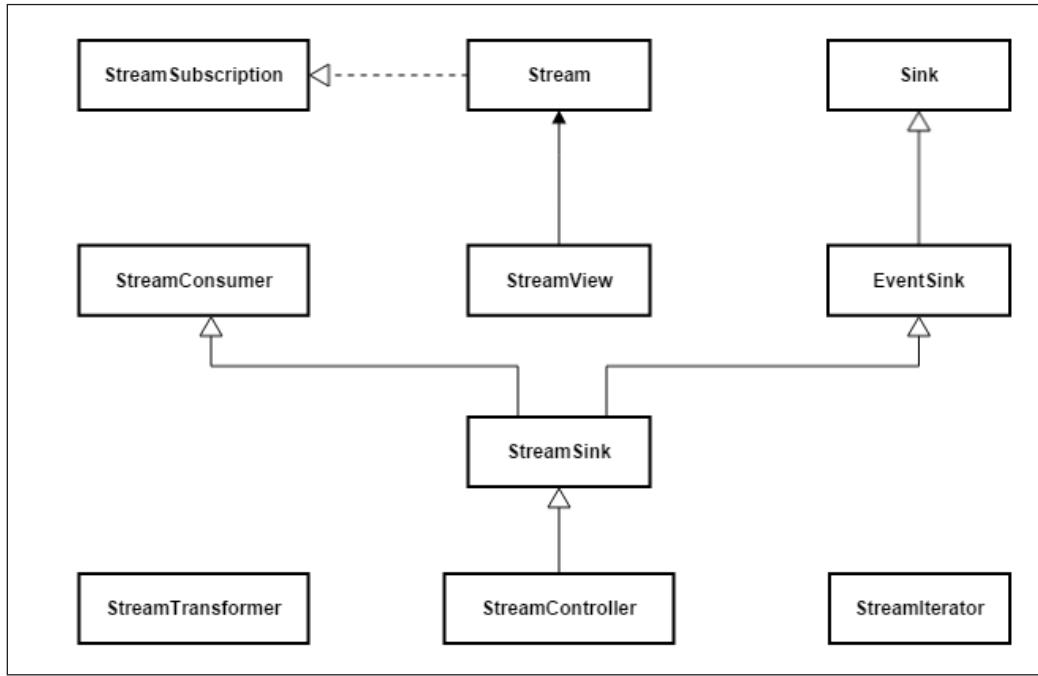
Use broadcast streams when delivering events to multiple listeners is important for your solution.



A good example of using the broadcast stream is the eventbus implementation that supports multiple listeners.

An overview of the stream framework API

You can see the hierarchy of the stream framework classes in the following diagram:



The Stream class

The `Stream` class is abstract and provides a generic view on the sequence of byte lists. It implements numerous methods to help you manage or republish streams in your application. The sequence of events can be seamlessly provided by `Stream`. The events that are generated by `Stream` store the data to be delivered. In case of a failure, the `Stream` class generates an `error` event. When all the events have been sent, the `Stream` class generates `done` event.

The validation methods of the Stream class

The `Stream` class has the following methods that help you validate data returned from a stream:

- `any`: This method checks whether the test callback function accepts any element provided by this stream

- `every`: This method checks whether the test callback function accepts all the elements provided by this stream
- `contains`: This method checks whether the needle object occurs in the elements provided by this stream

The search methods of the Stream class

The following methods of the `Stream` class help you search for specific elements in a stream:

- `firstWhere`: This method finds the first element of a stream that matches the test callback function
- `lastWhere`: This method finds the last element of a stream that matches the test callback function
- `singleWhere`: This method finds the single element in a stream that matches the test callback function

The subset methods of the Stream class

The `Stream` class provides the following methods to create a new stream that contains a subset of data from the original one:

- `where`: This method creates a new stream from the original one with data events that satisfy the test callback function
- `skip`: This method creates a new stream with data events that are left after skipping the first count of the data events from the original stream
- `skipWhere`: This method creates a new stream with the data events from the original stream when they are matched by the test callback function
- `take`: This method creates a new stream with the data events that are provided at most the first count values of this stream
- `takeWhere`: This method creates a new stream with the data events from the original stream when the test callback function is successful

You will find other methods of the `Stream` class on the official help page at <https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart-async.Stream>.

Creating a stream

The `Stream` class has several factory constructors to create single-subscription or broadcast streams. Let's discuss them in detail.

A new stream from the Future class

The first factory constructor could confuse you because it creates a new single-subscription stream from the `Future` instance with the following command:

```
factory Stream.fromFuture(Future<T> future)
```

This confusion will be quickly cleared if I remind you about the translation of the `Future` class into the `Stream` class via the `asStream` method from the same class, as shown in the following code:

```
...
Stream<T> asStream() => new Stream.fromFuture(this);
...
```

The resulting stream will fire an `data` or `error` event when the `Future` class is completed, and closes itself with the `done` event. The following code snippet shows how this factory constructor can be used:

```
import 'dart:async';

main() {
  // single sample data
  var data = new Future<num>.delayed(
    const Duration(milliseconds:500), () {
      // Return single value
      return 2;
  });
  // create the stream
  Stream<num> stream = new Stream<num>.fromFuture(data);
  // Start listening
  var subscriber = stream.listen((data) {
    print(data);
  }, onError:(error) {
    print(error);
  }, onDone:() {
    print('done');
  });
}
```

The execution of the preceding code prints the following result on the console:

```
2
done
```

A new stream from the Iterable class

It's quite obvious that you should have a factory constructor that creates a single-subscription stream from any `Iterable` instance. Iteration over a stream happens only if the stream has a listener and can be interrupted if the listener cancels the subscription. Any error that occurs in the iteration process immediately ends the stream and throws an error. In this case, the `done` event doesn't fire because the iteration was not complete.

In the following code snippet, we create an instance of the `Iterable` class with a factory constructor that generates the sequence of numbers. We intentionally generate an `Exception` at the location of the third element:

```
import 'dart:async';

main() {
    // some sample generated data
    var data = new Iterable<num>.generate(5, (int idx) {
        if (idx < 3) {
            return idx;
        } else {
            throw new Exception('Wrong data');
        }
    });
    // create the stream
    Stream<num> stream = new Stream<num>.fromIterable(data);
    // Start listening
    var subscriber = stream.listen((data) {
        print(data);
    }, onError:(error) {
        print(error);
    }, onDone:() {
        print('done');
    });
}
```

The result of the preceding code is as follows:

```
0
1
2
Exception: Wrong data
```

A new stream with periodically generated events

In the previous topic, we used the `Iterable.generate` factory constructor to emit numeric data into our stream. However, we can do that well via the `Stream.periodic` factory constructor, as follows:

```
factory Stream.periodic(Duration period,
    [T computation(int computationCount)])
```

We can do this in a natural stream and a less verbose way, as follows:

```
import 'dart:async';

main() {
  // some sample generated data
  Stream<num> stream = new Stream
    .periodic(const Duration(milliseconds: 500), (int count) {
      // Return count
      return count;
    });
  // Start listening
  var subscriber = stream.listen((data) {
    print(data);
  }, onError:(error){
    print(error);
  }, onDone:() {
    print('done');
  });
}
```

The first parameter is the duration that gives you the interval between emitting events. The second parameter is the callback function that computes the event values. This function has a single parameter that defines sequential number of iterations.

A new stream from the transformation pipe

The `Stream.eventTransformed` factory constructor is quite interesting because it creates a new `Stream` from the existing one with the help of a sink transformation, as shown in the following code:

```
factory Stream.eventTransformed(Stream source,
    EventSink mapSink(EventSink<T> sink))
```

The first parameter is the source stream that provides events to the new one. The `mapSink` callback function is the one that is called when a new stream is listening. All the events from the existing stream pass through the sink to reach a new stream. This constructor is widely used to create stream transformers. In the following code, we will create a `DoublingSink` class. It accepts the output stream as an argument of the constructor. We will implement the number-doubling algorithm inside the `add` method. The other `addError` and `close` methods are simple and pass the incoming parameter values to the underlying stream object, as shown in the following code:

```
import 'dart:async';

/**
 * An interface that abstracts creation or handling of
 * Stream events.
 */
class DoublingSink implements EventSink<num> {
    final EventSink<num> _output;

    DoublingSink(this._output);

    /** Send a data event to a stream. */
    void add(num event) {
        _output.add(event * 2);
    }

    /** Send an async error to a stream. */
    void addError(errorEvent, [StackTrace stackTrace]) =>
        _output.addError(errorEvent, stackTrace);

    /** Send a done event to a stream.*/
    void close() => _output.close();
}
```

The `DoublingTransformer` class implements `StreamTransformer` for numbers. In the `bind` method, which is compulsory, we will create a new stream via the `eventTransformer` constructor and return the instance of `DoublingSink` as result of the constructor's callback, as shown in the following code:

```
class DoublingTransformer implements StreamTransformer<num, num> {
    Stream<num> bind(Stream<num> stream) {
        return new Stream<num>.eventTransformed(stream,
            (EventSink sink) => new DoublingSink(sink));
    }
}
```

```
}

void main() {
    // some sample data
    var data = [1,2,3,4,5];
    // create the stream
    var stream = new Stream<num>.fromIterable(data);
    // Create DoublingTransformer
    var streamTransformer = new DoublingTransformer();
    // Bound streams
    var boundStream = stream.transform(streamTransformer);
    // Because we start listening the 'bound' stream the 'listen'
    method
    // invokes the 'doublingTransformer' closure
    boundStream.listen((data) {
        print('$data');
    });
}
```

In the `main` method, we created a simple stream via the `Stream.fromIterable` factory constructor and created a stream transformer as an instance of `DoublingTransformer`. So, we can combine them together in a call of the `transform` method. When we start listening to the bounded stream, events from the source stream will be doubled inside `Doublingsink` and accommodated here. The following result is expected:

```
2
4
6
8
10
```

A new stream from StreamController

In the previous topics, we saw how a stream can be easily created from another stream with the help of one of the factory constructors. However, you can create a stream from scratch with help of `StreamController`, which gives you more control over generating events of a stream. With `StreamController`, we can create a stream to send the `data`, `error`, and `done` events to the stream directly. A stream can be created via the `StreamController` class through the different factory constructors. If you plan to create a single-subscription stream, use the following factory constructor:

```
factory StreamController( {void onListen(), void onPause(), void
onResume(), onCancel(), bool sync: false})
```

The controller has a life cycle that presents the following states:

- **Initial state:** This is where the controller has no subscription. The controller buffers all the data events in this state.
- **Subscribed state:** In this state, the controller has a subscription. The `onListen` and `onCancel` callback functions are called when the subscriber registers or ends the subscription accordingly. The callback functions `onPause` and `onResume` are called when the controlling stream via a subscriber changes the state to pause or resume. The controller may not call the `onResume` callback function if the new data from the stream was canceled.
- **Canceled state:** In this state, the controller has no subscription.
- **Closed state:** In this state, adding more events is not allowed.

If the `sync` attribute is equal to `true`, it tells the controller that the events might be directly passed into the listening stream by the subscriber when the `add`, `addError`, or `close` methods are called. In this case, the events will be passed only after the code that creates the events has returned.

A stream instance is available via the `stream` property. Use the `add`, `addError`, and `close` methods of `StreamSink` to manage the underlying stream.

The controller buffers the data until a subscriber starts listening, but bear in mind that the buffering approach is not optimized to keep a high volume of events. The following code snippet shows you how to create a single-subscription stream with `StreamController`:

```
import 'dart:async';

main() {
    // create the stream
    Stream<num> stream = createStream();
    // Start listening
    StreamSubscription<num> sub = createSubscription(stream);
}

StreamSubscription<num> createSubscription(Stream<num> stream) {
    StreamSubscription subscriber;
    subscriber = stream.listen((num data) {
        print('onData: $data');
        // Pause subscription on 3-th element
        if (data == 3) {
            subscriber.pause(new Future.delayed(

```

```
        const Duration(milliseconds: 500), () => 'ok'));  
    }  
},  
onError:(error) => print('onError: $error'),  
onDone:() => print('onDone'));  
return subscriber;  
}  
  
Stream<num> createStream() {  
    StreamController<num> controller = new  
    StreamController<num>(  
        onListen:() => print('Listening'),  
        onPause: () => print('Paused'),  
        onResume: () => print('Resumed'),  
        onCancel: () => print('Canceled'), sync: false);  
    //  
    num i = 0;  
    Future.doWhile((){  
        controller.add(i++);  
        // Throws exception on 5-th element  
        if (i == 5) {  
            controller.addError('on ${i}-th element');  
        }  
        // Stop stream at 7-th event  
        if (i == 7) {  
            controller.close();  
            return false;  
        }  
        return true;  
    });  
    return controller.stream;  
}
```

In the preceding code, we intentionally throw an error at the 5-th element and stop the stream at the 7-th element. The stream subscriber paused listening to the stream at the 3-th element and resumed it after a delay of 500 milliseconds. This will generate the following result:

```
Listening  
onData: 0  
onData: 1  
onData: 2  
onData: 3  
Paused
```

```
onData: 4
onError: on 5-th element
onData: 5
onData: 6
Canceled
onDone
```

The following factory constructor creates a controller for the broadcast stream, which can be listened to more than once:

```
factory StreamController.broadcast({void onListen(),
    void onCancel(), bool sync: false})
```

The controller created by this constructor delivers the `data`, `error`, or `done` events to all listeners when the `add`, `addError`, or `close` methods are called. The invocation method with the same name is called in an order and is always before a previous call is returned. This controller, as opposed to the single-subscribed one, doesn't have the internal queue of events. This means that the `data` or `error` event will be lost if there are no listeners registered at the time, this event is added. Each listener subscription acts independently. If one subscription pauses, then only this one is affected, so all the events buffer internally in the controller until the subscription resumes or cancels. The controller has a life cycle that has the following states:

- **Initial state:** This is where the controller has no subscription. The controller losses all the fired `data` and `error` events in this state.
- **Subscribed state:** This is where the first subscription is added to the controller. The `onListen` and `onCancel` callback functions are called at the moment when the first subscriber is registered or the last one ends its subscription simultaneously.
- **Canceled state:** In this state, the controller has no subscription.
- **Closed state:** In this state, adding more events is not allowed.

If the `sync` attribute is equal to `true`, it tells the controller that events might be passed directly into the listening stream by subscribers when the `add`, `addError`, or `close` methods are called. Hence, the events will be passed after the code that creates the event is returned, but this is not guaranteed when multiple listeners get the events. Independent of the value of the `sync` attribute, each listener gets all the events in the correct order. The following is a slightly changed version of the previous code with two subscriptions:

```
import 'dart:async';

main() {
```

```
// create the stream
Stream<num> stream = createStream();
StreamSubscription<num> sub1 = createSubscription(stream, 1);
StreamSubscription<num> sub2 = createSubscription(stream, 2);
}

StreamSubscription<num> createSubscription(Stream<num> stream, num
number) {
    // Start listening
    StreamSubscription subscriber;
    subscriber = stream.listen((num data) {
        print('onData ${number}: $data');
        // Pause subscription on 3-th element
        if (data == 3) {
            subscriber.pause(new Future.delayed(
                const Duration(milliseconds: 500), () => 'ok'));
        }
    },
    onError: (error) => print('onError: $error'),
    onDone: () => print('onDone'));
    return subscriber;
}

Stream<num> createStream() {
    StreamController<num> controller = new
        StreamController<num>.broadcast(
            onListen: () => print('Listening'),
            onCancel: () => print('Canceled'), sync: false);
    //
    num i = 0;
    Future.doWhile(() {
        controller.add(i++);
        // Throws exception on 5-th element
        if (i == 5) {
            controller.addError('on ${i}-th element');
        }
        // Stop stream at 7-th event
        if (i == 7) {
            controller.close();
            return false;
        }
        return true;
    });
    return controller.stream;
}
```

The preceding code snippet generates the following result:

```
Listening
onData 1: 1
onData 2: 1
onData 1: 2
onData 2: 2
onData 1: 3
onData 2: 3
onData 1: 4
onError: on 5-th element
onData 1: 5
onData 1: 6
onDone
onData 2: 4
onError: on 5-th element
onData 2: 5
onData 2: 6
Canceled
onDone
```

These results reaffirm the fact that the broadcast stream doesn't guarantee the order of the delivery events to different listeners. It only guarantees an order of the delivery events inside each listener.

What does the StreamSubscription class do?

The `listen` method of the `Stream` class adds the following subscription to the stream:

```
StreamSubscription<T> listen(
    void onData(T event),
    { Function onError, void onDone(), bool cancelOnError});
```

A callback function `onData` is called every time when a new data event comes from this stream. Existence of this function is important because without it nothing will happen. The optional `onError` callback is called when an error comes from the stream. This function accepts one or two arguments. The first argument is always an error from the stream. The second argument, if it exists, is a `StackTrace` instance. It can be equal to `null` if the stream received an error without `StackTrace` itself. When the stream closes, it calls the `onDone` callback function. The `cancelOnError` flag informs the subscription to start the cancellation the moment the error occurs.

A result of this method is the instance of the `StreamSubscription` class. It provides events to the listener and holds the callback functions used in the `listen` method of the `Stream` class. You can set or override all the three callback functions via the `onData`, `onError`, and `onDone` methods of the `StreamSubscriber` class. The listening stream can be paused and resumed with the `pause` and `resume` methods. A special flag `isPaused` returns `true` if an instance of the `StreamSubscription` class is paused. The stream subscription can end with the `cancel` method at any time. It returns a `Future` instance, which completes with a `null` value when the stream is done cleaning up. This feature is also useful for tasks such as closing a file after reading it.

Minimizing access to the Stream class members using StreamView

The `StreamView` class is wrapper for the `Stream` class exposes only the `isBroadcast` getter, the `asBroadCastStream` and `listen` methods from original one. So if you need clear `Stream` interface in your code, you can use it like this:

```
import 'dart:async';

main() {
    // some sample data
    var data = [1,2,3,4,5];
    // create the stream
    var stream = new Stream<num>.fromIterable(data);
    // Create a view
    var streamView = new StreamView(stream);
    // Now listen stream view like stream
    var subscriber = streamView.listen((data) {
        print(data);
    }, onError:(error) {
        print(error);
    }, onDone:() {
        print('done');
    });
}
```

You will get the following result:

```
1
2
3
4
5
done
```

The Sink and EventSink interfaces

A `Sink` class represents a generic interface for data receivers. It defines the `add` method that will put the data in the sink and the `close` method, which tells the sink that no data will be added in future. The `EventSink` class uses the `add` method of the `Sink` class to send a `data` event to a stream, as well as the `close` method to send a `done` event. The `addError` method belongs to the `EventSink` class that sends an asynchronous error to a stream.

Importance of the StreamConsumer interface

We can bind one stream to another via the `pipe` method of the `Stream` class. The consumer stream is represented by the `streamConsumer` interface. This interface defines the contract between two streams. The `addStream` method is used to consume the elements of the source stream. The consumer stream will listen on the source stream and do something for each event. It may stop listening after an error or may consume all errors and stop at the `done` event. The `close` method tells the consumer that no future streams will be added. This method returns the `Future` instance that is completed when the source stream has been consumed and the consumer is closed.

What does the StreamSink class do?

The `StreamSink` class combines methods from `StreamConsumer` and `EventSink`. You should know that methods from both the classes will block each other. We cannot send the `data` or `error` events via the methods of the `EventSink` class while we are adding the source stream via the `addStream` method from `StreamConsumer`. We can start using the methods from `EventSink` only after the `Future` instance returned by the `addStream` method is completed with a value. Also, the `addStream` method will be delayed until the underlying system consumes the data added by the `EventSink` method. The `StreamSink` class has a `done` getter that returns the `Future` instance that is completed when the owned `StreamSink` class is finished with one of the following conditions:

- It is completed with an error as a result of adding events in one of the `add`, `addError`, or `close` methods of the `EventSink` class
- It is completed with success when all the events have been processed and the sink has been closed or the sink has been stopped from handling more events

Transforming streams with the StreamTransformer class

The `StreamTransformer` class helps you create a new consumer stream that is bound to the original one via the `bind` method. The `StreamTransformer` class can be instantiated through two factory constructors that define different strategies on how the transformation will happen. In following factory constructor, we need to specify the following special transformer function:

```
const factory StreamTransformer (Function StreamSubscription<T>
    transformer(Stream<S> stream, bool cancelOnError))
```

The transformer function receives a bounded stream as an argument and returns an instance of the `StreamSubscription` class. If you are planning to implement your own stream transformer function, it will look like this:

```
import 'dart:async';

void main() {
    // some sample data
    var data = [1,2,3,4,5];
    // create the stream
    var stream = new Stream<num>.fromIterable(data);
    // Create StreamTransformer with transformer closure
    var streamTransformer =
        new StreamTransformer<num, num>(doublingTransformer);
    // Bound streams
    var boundStream = stream.transform(streamTransformer);
    // Because we start listening the 'bound' stream the
    // 'listen' method invokes the 'doublingTransformer'
    // closure
    boundStream.listen((data) {
        print('$data');
    });
}

StreamSubscription doublingTransformer(Stream<num> input,
    bool cancelOnError) {

    StreamController<num> controller;
    StreamSubscription<num> subscription;
    controller = new StreamController<num>(
        onListen: () {
```

```
subscription = input.listen((data) {
    // Scale the data double.
    controller.add(data * 2);
},
onError: controller.addError,
onDone: controller.close,
cancelOnError: cancelOnError);
});
return controller.stream.listen(null);
}
```

The preceding code generates the following output:

```
2
4
6
8
10
```

The other factory method creates a `StreamTransformer` class that delegates events to the special functions, which handle the `data`, `error`, and `done` events, as shown in the following code:

```
factory StreamTransformer.fromHandlers({
    void handleData(S data, EventSink<T> sink),
    void handleError(Object error, StackTrace stackTrace,
        EventSink<T> sink),
    void handleDone(EventSink<T> sink)})
```

The changed version of the previous example is as follows:

```
import 'dart:async';

void main() {
    // some sample data
    var data = [1,2,3,4,5];
    // create the stream
    var stream = new Stream<num>.fromIterable(data);
    // Create StreamTransformer with transformer closure
    var streamTransformer = new StreamTransformer<num, num>
        .fromHandlers(
            handleData:handleData,
            handleError:handleError,
            handleDone:handleDone);
    // Bound streams
```

```
var boundStream = stream.transform(streamTransformer);
// Because we start listening the 'bound' stream the
// 'listen' method invokes the 'handleData' function
boundStream.listen((data) {
    print('$data');
});
}

handleData(num data, EventSink<num> sink) {
    sink.add(data * 2);
}

handleError(Object error, StackTrace stackTrace, EventSink<num> sink)
{
    sink.addError(error, stackTrace);
}

handleDone(EventSink<num> sink) {
    sink.close();
}
```

The following result of this execution looks similar to previous one:

```
2
4
6
8
10
```

Traverse streams with StreamIterator

The `StreamIterator` class permits a stream to be read using the iterator operations. It has the `moveNext` method that waits for the next stream's value to become available and returns the `Future` value of the `bool` type, as follows:

```
Future<bool> moveNext();
```

If the result of `moveNext` is a success, then the `Future` class completes with the `true` value, else the iteration is done and no new value will be available. The current value of the stream exists in the `current` property of the `StreamIterator` instance, as shown in the following code:

```
T get current;
```

This value is valid when the `Future` class returned by the `moveNext` method completes with the `true` value and only until the next iteration. A `StreamIterator` class is an abstract class and can be instantiated only via the factory constructor, as follows:

```
factory StreamIterator(Stream<T> stream)
```

Let's change the example from the previous topic to use `StreamIterator`. We will create a simple stream from `Iterable` as we did before. Then, we will create an instance of `StreamIterator`. Finally, we will use the `forEach` function to iterate over the stream and call the closure function to print scaled elements of the stream, as shown in the following code:

```
main() {
    // some sample data
    var data = [1,2,3,4,5];
    // create the stream
    var stream = new Stream<num>.fromIterable(data);
    // Create an iterator
    var iterator = new StreamIterator(stream);
    // Iterate over all elements of iterator and print values
    forEach(iterator, (value) {
        // Scale the data double.
        print(value * 2);
    });
}
```

Actually, this code looks similar to the ones where we used iterators. All the magic happens inside the `forEach` method, as shown in the following code:

```
forEach(StreamIterator iterator, f(element)) {
    return Future.doWhile(() {
        Future future = iterator.moveNext();
        future.then((bool hasNext) {
            if (hasNext) {
                f(iterator.current);
            }
        });
        return future;
    });
}
```

As the `moveNext` method returns the `Future` value, we need to use the `doWhile` method of the `Future` class to perform the iteration. The Boolean result of `Future` returns a `hasNext` parameter. We call the closure function until the value of the `hasNext` parameter is `true`.

The code generates the following result:

```
2  
4  
6  
8  
10
```

Summary

Now you have a better understanding of the stream framework, which is one of the key concepts of Dart VM.

Streams have existed since the early days of UNIX. They have proved to be a dependable way to compose large systems out of the small components. The Dart stream framework supports single-subscription and broadcast streams in order to have different approaches depending on the required solution.

In the next chapter, we will see the collection framework and when and how to properly use different types of collection frameworks. You will also learn how to choose the correct data structure based on the usage patterns, concurrency, and performance considerations.

6

The Collection Framework

The collection framework is a set of high-performance classes used to store and manipulate groups of objects. This framework allows different types of collections to work in a similar manner and is designed around a set of standard interfaces. Several standard implementations of interfaces can be extended or adapted very easily.

In this chapter, we will cover the following topics:

- An introduction to the collection framework
- Ordering of elements in collections
- Class hierarchy of the main interfaces
- The Iterable and Iterator interfaces
- The List, Set, Queue, and Map collections implementation
- Immutable collections
- Choosing the right collection

A Dart collection framework

In general, a collection is an object that holds a group of objects. Each item in a collection is called an element. A Dart collection framework has the following benefits:

- It is a set of interfaces that forces developers to adopt some design principles
- It can improve the performance of applications significantly

A framework provides a unified interface to store and manipulate the elements of a collection and hide the actual implementation. The Dart implementation of collections is highly optimized for execution in Dart and JavaScript VMs and is far more efficient than what you could create yourself.

Ordering of elements

Several collections implicitly support ordering of elements and help in sorting them without any effort. We can find any element with or without the filter predicate or perform a binary search within a sorted array to improve the performance of large collections. We can sort collections by providing a collection-compare function via a comparator or an object-compare method via the **Comparable** interface.

The Comparable interface

There are many core classes in Dart that support a native comparison via the implementation of the Comparable interface. You can implement the Comparable interface in classes that you have created to use them in collections to prevent unusual results of the sorting operation. Here, we will modify the `Entity` class to implement the Comparable interface. All we have to do is implement the `compareTo` method as shown in the following code:

```
class Entity implements Comparable {
    final int index;

    Entity(this.index);

    int compareTo(Entity other) {
        return this.index.compareTo(other.index);
    }

    @override
    String toString() => index != null? index.toString() : null;
}
```

The `compareTo` method of the Comparable interface compares this `Entity` class to another one. It returns:

- A negative integer if the class is ordered before another element
- A positive integer if the class is ordered after another element
- A zero if the class and another element are ordered together

Now, we can safely order instances of the `Entity` class in our code with the `sort` method of the `List` class, as shown in the following code:

```
void main() {
    var first = new Entity(1),
        second = new Entity(2);
```

```
var list = [second, first];
print(list);
// => [2, 1]
list.sort();
print(list);
// => [1, 2]
}
```

The Comparator type

So how can you sort a class that doesn't implement a Comparable interface? Here is an Entity2 class where implementing a Comparable interface is either impossible or not desired:

```
class Entity2 {
    final int index;

    Entity2(this.index);

    @override
    String toString() => index != null ? index.toString() : null;
}
```

In order to compare this class, you must use the **Comparator** type definition as follows:

```
typedef int Comparator<T>(T a, T b);
```

The sort method of all the collection classes accepts a function that matches the signature of the Comparator. Here, we pass an anonymous function to sort the comparison of our Entity2 classes as follows:

```
void main() {
    var list = [new Entity2(2), new Entity2(1)];
    print(list);
    // => [2, 1]
    list.sort((Entity2 a, Entity2 b) {
        return a.index.compareTo(b.index);
    });
    print(list);
    // => [1, 2]
}
```

As you can see, the anonymous function takes two arguments of the same type and returns an integer. This exactly matches the signature of the Comparator type definition.

If the arguments of the `sort` method are omitted, it uses the static `compare` method of the `Comparable` interface.

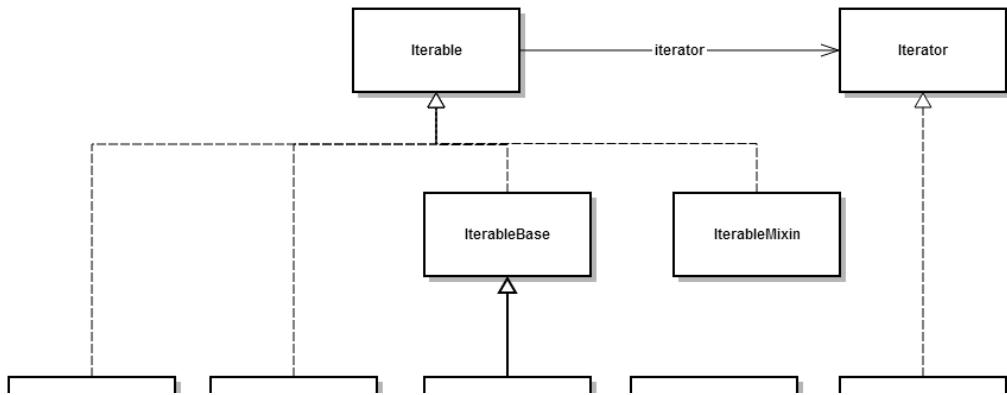
Collections and generics

All the collection classes that are implemented use generics very heavily. As discussed in *Chapter 2, Advanced Techniques and Reflection*, generics provide the type of object that a collection contains. Every attempt to add another type of element generates a static analysis warning. Generics in collections have the following advantages:

- They help avoid class cast errors at runtime since we get the warnings during the static analysis time
- They make the code cleaner, as there is no need to use casting operators and conditions to check types
- They add to the runtime benefit only because the execution is done in the production mode. The code that is compiled to JavaScript does not check types and generics

The collection class hierarchy

The Dart collection framework has a usable set of collection classes that exists in the `dart:core` and `dart:collection` libraries, as shown in the following diagram:



The standard collection interfaces simplify the passing and returning of collections to and from the class methods and allows the methods to work on a wide variety of collections. Using the common collection implementations makes the code shorter and quicker. By adhering to these implementations, you can make your code more standardized and easier to understand for yourself and others.

The Iterable interface

The **Iterable** interface can be defined as the common behavior of all classes in the collection framework that supports a mechanism to iterate through all the elements of a collection. It is an object that uses an Iterator interface to iterate over all the elements in a collection. There are two abstract classes, `IterableBase` and `IterableMixin`, that implement the Iterator interface. The `IterableMixin` class is perfectly suited to be extended in the mixin solutions. If you plan to create your own implementation of the `Iterable` interface, you need to extend one of them. There are many different methods in the `Iterable` interface to help you manipulate the elements in a collection.



The `Iterable` interface doesn't support adding or removing elements from a collection.



Properties of the Iterable collection

Here is a list of the read-only properties that are common for all collections:

- `length`: This property returns the number of elements in a collection
- `isEmpty`: This property returns `true` if there are no elements in a collection
- `isNotEmpty`: This property returns `true` if there is at least one element in a collection
- `first`: This property returns the first element of a collection or throws `StateError` if the collection is empty
- `last`: This property returns the last element of a collection or throws `StateError` if the collection is empty
- `single`: This property returns a single element in a collection. It throws `StateError` if the collection is empty or has more than one element
- `iterator`: This property returns a reference on instance of the `Iterator` class that iterates over the elements of the collection

Checking the items of a collection on a condition

Sometimes, you want to know if all or any of the objects in a collection comply with a specific condition. The following methods check whether the collection conforms to specific conditions:

- `every`: This method returns `true` if every element in the collection satisfies the specified condition. This can be seen in the following code:

```
List colors = ['red', 'green', 'blue'];
print(colors.every((color) => color != null || color != ''));
// => true
```

- `any`: This method returns `true` if one element in the collection satisfies the specified condition. This is illustrated in the following code:

```
List colors = ['red', 'green', 'blue'];
print(colors.any((color) => color == 'red'));
// => true
```

The iterate over collection

The following method helps you iterate through elements of the collection:

- `forEach`: This method applies the specified function to each element of the collection. This can be seen in the following code:

```
List colors = ['red', 'green', 'blue'];
colors.forEach((color) => print(color));
// => red
// => green
// => blue
```

The search over collection

The search over collection list includes the following methods that are used to search for an element in a collection:

- `contains`: This method returns `true` if the collection contains an element that is equal to the requested one. This is shown in the following code:

```
List colors = ['red', 'green', 'blue'];
print(colors.contains('red'));
// => true
```

- `elementAt`: This method returns the `indexth` element. Exactly which object is returned depends on the sorting algorithm implemented by the specific collection class that you're using, as shown in the following code:

```
List colors = ['red', 'green', 'blue'];
print(colors.elementAt(0));
// => red
```



If the collection does not support ordering, then the result of calling `elementAt` may be any element.

- `firstWhere`: This method returns the first element in the collection that satisfies the given predicate test or the result of the `orElse` function. It throws `StateError` if `orElse` was not specified, as illustrated in the following code:

```
List colors = ['red', 'green', 'blue'];
print(colors.firstWhere((color) => color == 'orange',
  orElse:() => 'orange'));
// => orange
```

- `lastWhere`: This method returns the last element in the collection that satisfies the given predicate test or the result of the `orElse` function. It throws `StateError` if `orElse` was not specified, as shown in the following code:

```
List colors = ['red', 'green', 'blue'];
print(colors.lastWhere((color) => color != 'orange',
  orElse:() => ''));
// => blue
```

- `singleWhere`: This method returns a single element of the collection that satisfies the test. If the collection is empty or more than one element matches, then it throws `StateError`. The code is as follows:

```
List colors = ['red', 'green', 'blue'];
print(colors.singleWhere((color) => color == 'red'));
// => red
```

Creating a new collection

The following list includes methods to create a new collection from the original one; all of them return Lazy Iterable results:

- `expand`: This method returns new collections by expanding each element of the original one to zero or more elements. This can be seen in the following code:

```
List colors = ['red', 'green', 'blue'];
print(colors.expand((color) {
    return color == 'red'
        ? ['orange', 'red', 'yellow']
        : [color];
}));
// => [orange, red, yellow, green, blue]
```

- `map`: This method creates a new collection of elements based on the elements from the original collection that are transformed with specified function. The code is as follows:

```
List colors = ['red', 'green', 'blue'];
print(colors.map((color) {
    if (color == 'green') return 'orange';
    if (color == 'blue') return 'yellow';
    return color;
}));
// => ['red', 'orange', 'yellow']
```

- `take`: This method returns an Iterable collection with a specified number of elements from the original collection. The value of these elements must not be negative. If the number of requested elements is more than the actual number of elements, then it returns all the elements from the collection:

```
List nums = [1, 2, 3, 4, 5, 6];
print(nums.take(7));
// => [1, 2, 3, 4, 5, 6]
```

- `takeWhile`: This method returns an Iterable collection that stops once the test is not satisfied anymore. This is illustrated in the following code:

```
List nums = [1, 2, 3, 4, 5, 6];
print(nums.takeWhile((element) => element < 5));
// => [1, 2, 3, 4]
```

- **skip**: This method returns an Iterable collection that skips the specified number of initial elements. If it has fewer elements than the specified number, then the resulting Iterable collection is empty. Also, the specified number must not be negative. The code is as follows:

```
List nums = [1, 2, 3, 4, 5, 6];
print(nums.skip(4));
// => [5, 6]
```

- **skipWhile**: This method returns an Iterable collection that skips the elements while the test is satisfied. This is shown in the following code:

```
List nums = [1, 2, 3, 4, 5, 6];
print(nums.skipWhile((element) => element <= 4));
// => [5, 6]
```

- **where**: This method returns a Lazy Iterable collection with all the elements that satisfy the predicate test. This is illustrated in the following code:

```
List nums = [1, 2, 3, 4, 5, 6];
print(nums.where((element) => element > 1 && element < 5));
// => [2, 3, 4]
```

- **toList**: This method creates a list that contains the elements of the original collection. It creates a fixed length List if the growable attribute is false:

```
List nums = [1, 2, 3];
print(nums.toList(growable:false));
// => [1, 2, 3]
```

- **toSet**: This method creates a set that contains the elements of the original collection. It ignores the duplicate elements. The code is as follows:

```
List nums = [1, 2, 1];
print(nums.toSet());
// => {1, 2}
```

Reducing a collection

The following list includes methods to reduce the number of elements in a collection:

- **reduce**: This method reduces the collection to a single value by iteratively combining the elements of the collection using the provided function. If the collection is empty, this results in `StateError`. In the following example, we will calculate the sum of all elements in the collection:

```
List nums = [1, 2, 3];
print(nums.reduce((sum, element) => sum + element));
// => 6
```

- `fold`: This method reduces the collection to a single value by iteratively combining each element of the collection with an existing value using the specified function. Here, we have to specify the initial value and aggregation function:

```
List nums = [1, 2, 3];
print(nums.fold(0, (acc, element) => acc + element));
// => 6
```

Converting a collection

The following method is used to convert all the elements of a collection:

- `join`: This method converts each element of the collection into a string and returns the concatenated result separated with an optional separator. If the collection is empty, it doesn't actually modify the type of the elements in the collection, but just returns an empty string:

```
List nums = [1, 2, 3];
print(nums.join(' - '));
// => 1 - 2 - 3
```

Generating a collection

The Iterable interface has a factory method that helps to create a new Iterable interface and is filled with a specified number of values generated by a generator function, as shown in the following code:

```
Iterable generated = new Iterable.generate(4,
    (count) => "Is $count");
print(generated);
// => [Is 0, Is 1, Is 2, Is 3]
```

If the generator function is absent, this method generates a collection with only the integer values:

```
Iterable generated = new Iterable.generate(4);
print(generated);
// => [0, 1, 2, 3]
```

The Lazy Iterable

The Lazy Iterable term is used plenty of times in Iterable interfaces. It is an iteration strategy that delays the iteration of a collection until its value is needed and avoids repeated iterations. In the following example, our code iterates over the list of numbers.

The `where` method prints the information about the current fetched element. This function calls the object only when we actually fetch the element in the `forEach` method of the `Iterable` interface, as shown in the following code:

```
lazyIterable() {
    List nums = [1, 2, 3];
    print('Get Iterable for $nums');
    Iterable iterable = nums.where((int i) {
        print('Fetched $i');
        return i.isOdd;
    });
    print('Start fetching');
    iterable.forEach((int i) {
        print("Received $i");
    });
}
```

Here is the output of the preceding function:

```
Get Iterable for [1, 2, 3]
Start fetching
Fetched 1
Received 1
Fetched 2
Fetched 3
Received 3
```

The following are the benefits of the Lazy Iterable:

- The performance increases because unnecessary iterations are avoided
- The memory usage footprint decreases because the values are iterated when needed
- It helps to create infinite data structures

Bear in mind that iteration over the Lazy Iterable could be much slower than a normal iteration because the code incurs the cost of an invocation to fetch the next item from the Iterable source.

The Iterable interface

The Iterable interface has a strong relation to the Iterator. The Iterator is an interface used to get items from a collection, one at a time. It follows the fail-fast principles to immediately report whether the iterating collection was modified. The Iterator has a property called `current`, which is used to return a currently pointed element. The Iterator is initially positioned before the first element in a collection. The `moveNext` method returns `true` if there is a next element in the collection and `false` if not. Before using the Iterator, it must be initialized with the `moveNext` method to point it to the first element. In the following code, we don't initialize the Iterator with the `moveNext` method:

```
void main() {
    List<String> colors = ['red', 'green', 'blue'];

    Iterator<String> iter = colors.iterator;
    do {
        print(iter.current);
    } while (iter.moveNext());
}
```

The result of this code is unspecified, but it can return `null` or generate an exception, as shown in the following code:

```
null
red
green
blue
```



Always initialize the Iterator with the `moveNext` method to prevent unpredictable results.



Here is an example that shows you how to use the Iterator properly:

```
void main() {
    List<String> colors = ['red', 'green', 'blue'];

    Iterator<String> iter = colors.iterator;
    while (iter.moveNext()) {
        print(iter.current);
    }
}
```

The result is as expected:

```
red  
green  
blue
```

Invocation of the `moveNext` method returns `false` after the collection ends, and the current pointer always returns the last element.

The `for` loop statement uses the Iterator transparently to iterate through the collection:

```
void main() {  
    List<String> colors = ['red', 'green', 'blue'];  
  
    for (String color in colors) {  
        print(color);  
    }  
}
```

The result is similar to that of the preceding example.

BidirectionalIterator

Sometimes, we need to iterate over a collection of elements in both directions. To help in such cases, Dart provides `BidirectionalIterator`. In the following code, `BiListIterator` is the implementation of `BidirectionalIterator`:

```
class BiListIterator<E> implements BidirectionalIterator<E> {  
    final Iterable<E> _iterable;  
    final int _length;  
    int _index;  
    E _current;
```

The constructor has an extra optional `back` parameter that defines the direction of the iteration:

```
BiListIterator(Iterable<E> iterable, {bool back:false}) :  
    _iterable = iterable, _length = iterable.length,  
    _index = back ? iterable.length - 1 : 0;  
  
    E get current => _current;
```

The following code shows the `moveNext` method of the Iterator to move forward. This and the next method compare the length of the Iterable and the actual length of the collection to check concurrent modifications. The code is as follows:

```
bool moveNext() {
    int length = _iterable.length;
    if (_length != length) {
        throw new ConcurrentModificationError(_iterable);
    }
    if (_index >= length) {
        _current = null;
        return false;
    }
    _current = _iterable.elementAt(_index);
    _index++;
    return true;
}
```

The following `movePrevious` method of `BidirectionalIterator` is used to move backwards:

```
bool movePrevious() {
    int length = _iterable.length;
    if (_length != length) {
        throw new ConcurrentModificationError(_iterable);
    }
    if (_index < 0) {
        _current = null;
        return false;
    }
    _current = _iterable.elementAt(_index);
    _index--;
    return true;
}
```

I have created a small example to prove that we can move in both directions:

```
main() {
    var list = new List.from([1, 2, 3, 4]);
    // Forward Iteration
    BiListIterator iter = new BiListIterator(list);
    while(iter.moveNext()) {
        print(iter.current);
    }
    // => 1, 2, 3, 4
```

```
// Backward Iteration
iter = new BiListIterator(list, back:true);
while(iter.movePrevious()) {
    print(iter.current);
}
// => 4, 3, 2, 1
```

First, I created an instance of `List`, but it might be `Set` or `Queue` or any other collection that implements the `Iterable` interface. Then, I instantiated `BiListIterator` and pointed it to my `Iterable` collection so that we are ready to traverse via elements of the collection in the forward direction. Later, I created another instance of `BiListIterator` but specified the backward direction of the iteration. Finally, I could call `movePrevious` to move in the backward direction.

The collection classes

The collection framework has the following classes for all occasions:

- `List`: This is an ordered collection that supports indexed access to elements and allows duplicate elements
- `Set`: This is a collection of elements in which each element can occur only once
- `Queue`: This is a collection that can be manipulated at both ends
- `Map`: This is a collection of key-value pairs where each element is accessible by a unique key

All of them define their own specific way to add or remove elements from collections. Let's discuss each of them.

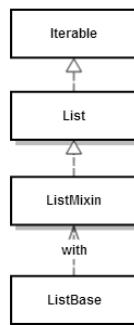
List

The `List` class implements the `Iterable` interface and intensively uses the indexed order to iterate over elements in the collection. The `List` class can be of a fixed or variable length. A fixed-length list can only be created by a constructor with a specific number of elements: `new List(5)`. The fixed-length type has restrictions on all operations changing the length of the list and finishing with `UnsupportedError`. The features of the fixed-length type are as follows:

- The length cannot be changed
- Any value can be assigned to the list but by the index operator only

- The elements cannot be removed from the list
- The list cannot be cleaned

The variable list returns as a result of the new `List()` or `[]` operations. It has an internal buffer and dynamically changes its size when needed. Any attempt to change the length of the `List` class during iteration will result in `ConcurrentModificationError`. The following diagram shows the hierarchy of the list-based classes:



Dart always creates an instance of the `List` class as a result of instantiation. If the standard implementation of the `List` class is not enough, you can create your own implementation; just extend the `ListBase` class to align with your needs, as shown in the following code:

```
import "dart:collection";

class NewList<E> extends ListBase {
    final List<E> _elements;

    NewList() :_elements = new List<E>();

    @override
    operator [] (int index) {
        return _elements[index];
    }

    @override
    void operator []=(int index, value) {
        _elements[index] = value;
    }
}
```

```

@Override
int get length => _elements.length;

@Override
void set length(int newLength) {
    _elements.length = newLength;
}
}

```

You might be surprised to know that you need to implement only four methods to have a fully functional list-based class. This is because the other methods of the `ListBase` class use those four main methods to manage the internal buffer and iterate over elements. If you do not desire to extend the `ListBase` class, you can use `ListMixin` as follows:

```

class OtherList<E> extends MainList with ListMixin<E> {
    // ...
}

```

The `List` class interface supports ordering via a `sort` method.

 The `asMap` method of the `List` class returns a `Map` view that cannot be modified.

Sometimes, we need to randomly rearrange the elements in the `List` class. The `shuffle` method of the `List` class can come in handy while doing that:

```

import 'dart:math';

main() {
    var list = new List.from([1, 2, 3, 4, 5]);
    print(list);
    // => [1, 2, 3, 4, 5]
    // Create seed to initialize internal state of
    // random-number generator
    var seed = new DateTime.now().millisecondsSinceEpoch;
    // Create instance of generator
    var random = new Random(seed);
    // Re-arrange elements in list
    list.shuffle(random);
    print(list);
    // => [4, 5, 1, 3, 2]
}

```

Run the preceding code snippet a couple of times and see the different results of the shuffle operation.

LinkedList

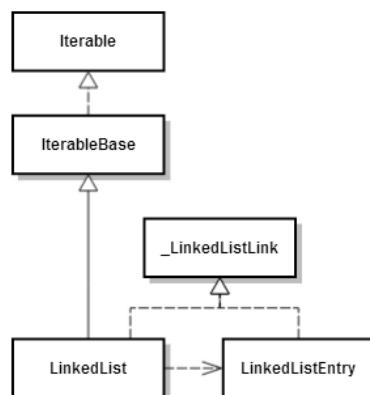
The `LinkedList` class is a double-linked list. A `LinkList` class is a collection of elements of the same data type, and it is efficient when it comes to the insertion and deletion of elements of a complex structure. Despite the name, it has nothing in common with the `List` class.



The `LinkedList` class does not extend or implement the `List` class.



Let's take a look at the class hierarchy of the `LinkedList` class:



All the elements in `LinkedList` are based on `LinkedListEntry` and connected through pointers. Each `LinkedListEntry` class has a pointer field pointing to the next and previous elements in the list. It contains the link of the `LinkedList` instance it belongs to. Before adding the element to another `LinkedList` class, it must be removed from the current one. If it is not, `StateError` is thrown. Each element of the `LinkedList` class knows its own position, so we can use methods such as `addBefore`, `addAfter`, or `unlink` of `LinkedListEntry` to manipulate them:

```
import "dart:collection";
```

We must create a wrapper class `Element` based on `LinkedListEntry` to keep our elements, as shown in the following code:

```
class Element<E> extends LinkedListEntry {  
    final E value;  
    Element(this.value);  
    @override  
    String toString() => value != null ? value.toString() : null;  
}
```

Here, we create the `LinkedList` instance and use the `Element` wrapper:

```
main() {  
    LinkedList<Element> list = new LinkedList<Element>();  
    Element b = new Element("B");  
    list.add(b);  
    //  
    b.insertAfter(new Element("A"));  
    b.insertBefore(new Element("C"));  
    print(list);  
    // => (C, B, A)  
    b.unlink();  
    print(list);  
    // => (C, A)  
}
```

Finally, we use `insertAfter`, `insertBefore`, and `unlink` of the `Element` methods to manipulate these elements. The advantages of `LinkedList` are as follows:

- It is not necessary to know the number of elements in advance, and it does not allocate more memory than necessary
- Operations such as insertion and deletion have a constant time and handle memory efficiently, especially when the element is inserted in the middle of a list
- It uses the exact amount of memory needed for an underlying element and wrapper

The disadvantages of `LinkedList` are as follows:

- It doesn't support random access to any element
- The element search can be done only via iteration
- It uses more memory to store pointers on linked elements than the list uses

Set

The `Set` class is a collection that cannot contain identical elements. It does not allow indexed access to an element in the collection, so only the `iterator` and `for-each` loop methods can traverse elements of a `Set` class:

```
void main() {  
    var asset = new Set.from([3, 2, 3, 1]);  
    print(asset);  
    // => {3, 2, 1}  
}
```

A `Set` class can contain at most one null element.



The `Set` factory creates the instance of `LinkedHashSet`.



The `Set` class can return a new set as a result of the execution of the `intersection` method between its internal collection and the other one:

```
main() {  
    var asset = new Set.from([3, 2, 3, 1]);  
    print(asset);  
    // => {3, 2, 1}  
    var other = new Set.from([2, 1, 5, 6]);  
    print(other);  
    // => {2, 1, 5, 6}  
    var intersect = asset.intersection(other);  
    print(intersect);  
    // => {2, 1}  
}
```

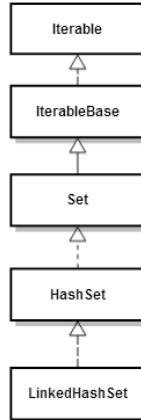
The `union` method returns a new `Set` class that contains all the elements in its internal collection and the other one:

```
main() {  
    var asset = new Set.from([3, 2, 3, 1]);  
    print(asset);  
    // => {3, 2, 1}  
    var other = new Set.from([2, 1, 5, 6]);  
    print(other);  
    // => {2, 1, 5, 6}  
    var union = asset.union(other);  
    print(union);  
    // => {3, 2, 1, 5, 6}  
}
```

If you need to find the difference between the elements of a certain collection and other collections, use the `difference` method of the `Set` class as follows:

```
main() {
  var asset = new Set.from([3, 2, 3, 1]);
  print(asset);
  // => {3, 2, 1}
  var other = new Set.from([2, 1, 5, 6]);
  print(other);
  // => {2, 1, 5, 6}
  var difference = asset.difference(other);
  print(difference);
  // => {3}
}
```

Here is the class hierarchy of the set-based classes:



HashSet

The `HashSet` class is a hash-table implementation of `Set`, providing fast lookup and updates. There are operations such as `add`, `contains`, `remove`, and `length` that have a constant time of execution:

```
import 'dart:collection';

void main() {
  var hset = new HashSet.from([3, 2, 3, 1]);
  print(hset);
  // => {1, 2, 3}
}
```

With `HashSet`, we can control the consistent equality via the constructor arguments:

```
import 'dart:collection';

void main() {
  var hset = new HashSet(equals: (e1, e2) {
    return e1 == e2;
  }, hashCode: (e) {
    return e.hashCode;
  });
  hset.addAll([3, 2, 3, 1]);
  print(hset);
  // => {1, 2, 3}
  hset.add(1);
  print(hset);
  // => {1, 2, 3}
}
```

The constructor's named argument, `equals`, must be a function to compare the equality of two elements in the collection. Another constructor's named argument, `hashCode`, must be a function that calculates the hash code of the specified element. If both the elements are deemed equal, then they should return the same hash code. If both named arguments are omitted, the `Set` class uses the internal `equals` and `hashCode` methods of the element.

LinkedHashSet

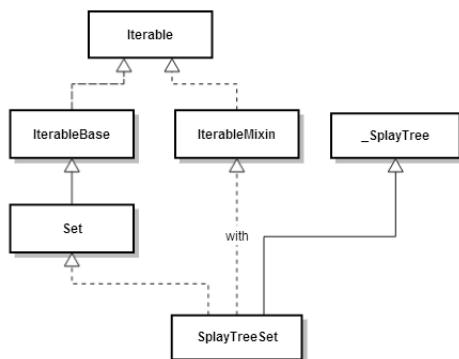
The `LinkedHashSet` class is an ordered hash-table-based set implementation that maintains the insertion order of elements for iteration and runs nearly as fast as `HashSet`. The order of adding items to a collection determines the order of iteration via elements of the collection. The consistent equality in `LinkedHasList` is defined by the `equals` operator and mostly based on the value of the `hashCode` method. Adding an element that is already in `Set` does not change its position in the iteration order, but removing an element and adding it again will make it the last element of iteration:

```
import 'dart:collection';

void main() {
  var hset = new LinkedHashSet();
  hset.addAll([3, 2, 3, 1]);
  print(hset);
  // => {3, 2, 1}
  hset.add(1);
  print(hset);
  // => {3, 2, 1}
}
```

SplayTreeSet

Last but not least, `SplayTreeSet` is a class that maintains the collection in a sorted order, but is slow when it comes to lookups and updates. It extends a `_SplayTree` class. Class `_SplayTree` is a self-balancing **binary search tree (BST)**. The time taken by most operations in BST is proportional to the height of the tree, and it is better to keep it small. Self-balancing BST reduces the height by performing tree transformation at logarithmic time, $O(\log(n))$, where n is the height of the tree. The following diagram shows the hierarchy of classes:



By default, a `SplayTreeSet` class assumes that all elements are comparable and uses an object-compare method to order them. In my example, I have added an array of strings to `SplayTreeSet`:

```

import 'dart:collection';

main() {
  var sset = new SplayTreeSet();
  sset.addAll(['33', '2', '33', '10']);
  print(sset);
  // => (10, 2, 33)
}
  
```

The order of the result is correct from the perspective of comparing the strings, but we have to order them with respect to the integer values to represent them as strings. To fix this problem, we can pass the compare function as an argument of the constructor:

```

import 'dart:collection';

main() {
  var sset = new SplayTreeSet((e1, e2) {
  
```

```
        return int.parse(e1).compareTo(int.parse(e2));
    });
    sset.addAll(['33', '2', '33', '10']);
    print(sset);
    // => (2, 10, 33)
}
```

Now the result looks exactly the way we want it to be.

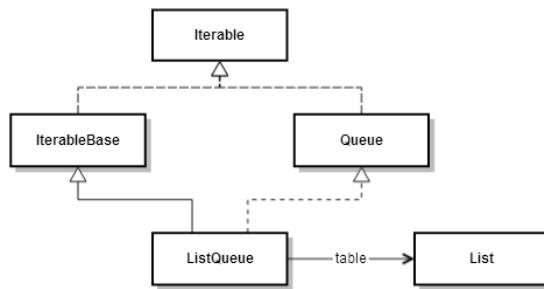
Queue

A Queue class is a collection of elements that are added and removed in a specific order at both ends. It generally accepts null elements. A ListQueue class is the implementation of the general purpose Queue class. It uses a List instance to keep the collection elements and uses head and tail pointers to manipulate them. The ListQueue class implementation is a very efficient solution for any queue or stack usage with a small memory footprint, as shown in the following code:

```
import 'dart:collection';

void main() {
    var queue = new Queue();
    queue.add(2);
    queue.add(3);
    queue.addFirst(1);
    print(queue);
    // => {1, 2, 3}
    queue.removeLast();
    print(queue);
    // => {1, 2}
}
```

Here is the class hierarchy of Queue:



The initial capacity of `ListQueue` is eight elements, and can be changed by passing another number as an argument of the constructor. Actually, this value will be rounded up to the nearest power of two. It always checks the number of elements in queue and grows an internal collection automatically via the creation of a new `List` instance and copying elements from the old one to the new one. The removal of elements is performed by moving elements one by one to fill the hole. The `Queue` class does not reduce the size of an internal collection when it removes elements. Elements in a `Queue` class can be traversed via a `for-each` loop or within an `Iterator`.

Map

The `Map` class is a collection of key-value pairs. Each key has a reference to only one value. It does not allow duplicate keys, but allows duplicate values. The `Map` class is not a subtype of the `Iterator`, `IteratorBase`, or even `IteratorMixin`. The `Map` class provides separate iterators for keys and values:

```
void main() {
    var map = new Map.fromIterables([3, 2, 1], ['3', '2', '1']);
    print(map);
    // => {3: 3, 2: 2, 1: 1}
}
```

The `Map` class allows you to use the `null` value as a key. Each of the `Map` class implementations behave a little differently with respect to the order of elements when iterating the map. A key of the `Map` class must implement the `equals` operator and the `hashCode` method.



The `Map` factory creates the instance of `LinkedHashMap`.

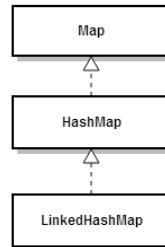
The `Map` class doesn't support duplicate keys. It has a `putIfAbsent` method to look up the value of the key or add a new value if it is not present, as shown in the following code:

```
void main() {
    var map = new Map.fromIterables([3, 2, 1], ['3', '2', '1']);
    print(map);
    // => {3: 3, 2: 2, 1: 1}
    map.putIfAbsent(3, () => '33');
    map.putIfAbsent(4, () => '4');
    print(map);
    // => {3: 3, 2: 2, 1: 1, 4: 4}
}
```

This method adds key-value pairs only if the key is absent. The `containsKey` and `containsValue` methods return the search result:

```
void main() {
    var map = new Map.fromIterables([3, 2, 1], ['3', '2', '1']);
    print(map);
    // => {3: 3, 2: 2, 1: 1}
    print(map.containsKey(1));
    // => true
    print(map.containsKey(5));
    // => false
    print(map.containsValue('2'));
    // => true
    print(map.containsValue('55'));
    // => false
}
```

Here is the hierarchy of the `Map` class:



HashMap

The `HashMap` class is a hash-table-based implementation of `Map`, providing fast lookup and updates. It maps the keys and values without guaranteeing the order of elements. In the following code, the `print` function result might be in a different order:

```
import 'dart:collection';

void main() {
    var map = new HashMap.fromIterables([2, 3, 1], ['2', '3', '1']);
    print(map);
    // => {2: 2, 1: 1, 3: 3}
}
```

Iteration of keys and values happens in parallel to reduce the time to search elements:

LinkedHashMap

The `LinkedHashMap` class is a hash-table-based implementation of `Map` with the link list of keys to facilitate `insert` and `delete` operations, and it runs nearly as fast as `HashMap`. It remembers the key insertion order and uses it when it iterates via keys. The change in the values doesn't affect the keys' order:

```
import 'dart:collection';

void main() {
    var map = new LinkedHashMap(
        fromIterables([3, 2, 1], ['3', '2', '1']));
    print(map);
    // => {3: 3, 2: 2, 1: 1}
    map.remove(3);
    map[3] = '3';
    print(map);
    // => {2: 2, 1: 1, 3: 3}
}
```

You can provide the custom `equals` and `hashCode` functions as arguments of the constructor. The `equals` function is used to compare the keys in the table with the new keys. The following `hashCode` function is used to provide a hash value of the key:

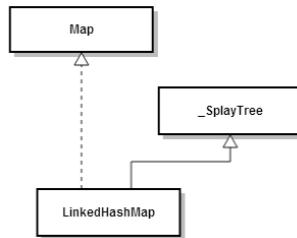
```
import 'dart:collection';

void main() {
    var map = new LinkedHashMap(equals: (e1, e2) {
        return e1 == e2;
    }, hashCode: (e) {
        return e.hashCode;
    });
    map.addAll({3: '3', 2: '2', 1: '1'});
    print(map);
    // => {3: 3, 2: 2, 1: 1}
    map.remove(3);
    map[3] = '3';
    print(map);
    // => {2: 2, 1: 1, 3: 3}
}
```

If the `equals` attribute in a constructor is provided, it is used to compare the keys in the hash table with the new keys, else the comparison of the keys will happen with the `==` operator. Similarly, if the `hashCode` attribute of a constructor is provided, it is used to produce a hash value for the keys in order to place them in the hash table or else use the keys' own `hashCode` method.

SplayTreeMap

The `SplayTreeMap` class is an ordered map that maintains a collection in a sorted order, but is slower when it comes to lookups and updates. This class is based on the `_SplayTree` class and also on `SplayTreeSet`. The `SplayTreeMap` class is a self-balancing binary search tree, as shown in the following diagram:



In the following code, the keys of the map are compared with the `compare` function passed in the constructor:

```
import 'dart:collection';

void main() {
    var map = new SplayTreeMap((e1, e2) {
        return e1 > e2 ? 1 : e1 < e2 ? -1 : 0;
    });
    map.addAll({3: '3', 2: '2', 1: '1'});
    print(map);
    // => {1: 1, 2: 2, 3: 3}
    map.remove(3);
    map[3] = '3';
    print(map);
    // => {1: 1, 2: 2, 3: 3}
}
```

By default, a `SplayTreeMap` function assumes that all the keys are comparable and uses an object-compare method to compare the keys of elements.

Unmodifiable collections

The collection framework has unmodifiable versions of the existing collections with the following advantages:

- To make a collection immutable once it has been built and not modify the original collection to guarantee absolute immutability, although the elements in that collection are still mutable
- To allow read-only access to your data structure from the client code and the client code can look into it without modifying it while you have full access to the original collection

The unmodifiable list

The unmodifiable collection based on the `List` class is `UnmodifiableListView`. It creates an unmodifiable list backed by the source provided via the argument of the constructor:

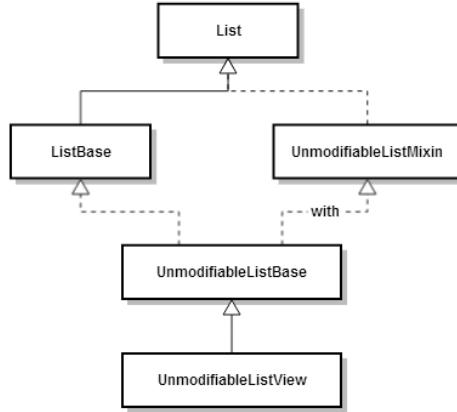
```
import 'dart:collection';

void main() {
  var list = new List.from([1, 2, 3, 4]);
  list.add(5);
  var unmodifiable = new UnmodifiableListView(list);
  unmodifiable.add(6);
}
```

The execution fails when we try adding a new element to an unmodifiable collection, as shown in the following code:

```
Unsupported operation: Cannot add to an unmodifiable list
#0 ListBase<&&UnmodifiableListMixin>.add (...)
#1 main (file:///.../bin/unmodifiable.dart:7:19)
...
...
```

The following diagram shows the class hierarchy of `UnmodifiableListView`:



The unmodifiable map

Another unmodifiable collection is `UnmodifiableMapView`, which is based on the `Map` class. It disallows modifying the original map via the view wrapper:

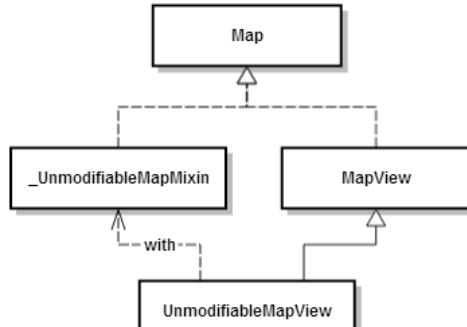
```
import 'dart:collection';

void main() {
  var map = new Map.fromIterables(
    [1, 2, 3, 4], ['1', '2', '3', '4']);
  map[5] = '5';
  var unmodifiable = new UnmodifiableMapView(map);
  unmodifiable[6] = '6';
}
```

Any attempt to modify the collection throws a runtime exception as follows:

```
Unsupported operation: Cannot modify unmodifiable map
#0      MapView&&_UnmodifiableMapViewMixin.[]=(...)
#1      main (file:///.../bin/unmodifiable_map.dart:7:15)
...
```

The following diagram shows the class hierarchy of `UnmodifiableMapView`:



Choosing the right collection

How do you choose the right collection for specific cases? I'm sure many of us have asked that question at least once. Let me try to help you to make the right choice:

- The `List`, `Set` (such as `LinkedHashSet`), and `Map` (such as `LinkedHashMap`) classes are perfect choices for general purposes. They have enough functionality to cover all of your needs.
- Choosing a class implements the minimum functionality that you require. Don't choose a class that supports sorting if you don't actually need it.

Here is a table that combines all the classes with the supported features:

Class	Order	Sort	Random access	Key-values	Duplicates	Null
<code>List</code>	Yes	Yes	Yes	No	Yes	Yes
<code>LinkedList</code>	Yes	No	No	No	Yes	Yes
<code>Set</code> or <code>LinkedHashSet</code>	Yes	No	No	No	No	No
<code>HashSet</code>	No	No	No	No	No	No
<code>SplayTreeSet</code>	Yes	Yes	No	No	No	No
<code>Queue</code> or <code>ListQueue</code>	Yes	Yes	No	No	Yes	Yes
<code>Map</code> or <code>LinkedHashMap</code>	Yes	No	Yes	Yes	No	Yes
<code>HashMap</code>	No	No	Yes	Yes	No	Yes
<code>SplayTreeMap</code>	Yes	Yes	Yes	Yes	No	Yes

The order is supported via iteration. Sort is supported via a collection-compare function via a `Comparator` or an object-compare method via the `Comparable` interface.

Summary

We have now discovered the collection framework, and it's time to have a look at what we have mastered.

The collection framework is set for high-performance classes to store and manipulate groups of objects. The framework provides a unified architecture to store and manipulate the elements of a collection and hide the actual implementation.

Several collections implicitly support ordering of elements and help us to sort elements without effort. We can also sort collections by providing a collection-compare function via a `Comparator` or an object-compare method via the `Comparable` interface.

The `Iterable` interface defines the common behavior of all the classes in a collection framework that supports a mechanism to iterate through all the elements of a collection. The `Iterator` follows the fail-fast principles to immediately report whether the iterating collection was modified. If you plan to create your own implementation of the `Iterable` interface, you need to extend `IterableBase` or `IterableMixin`. `BidirectionalIterator` helps to iterate over collections of elements in both directions.

The collection framework also has the `List`, `Map`, `Queue`, and `Set` classes for all occasions. `LinkedList` does not extend the `List` class. `HashSet`, `LinkedHashSet`, and `SplayTreeSet` are implementations of the `Set` interface. `ListQueue` is the implementation of the `Queue` interface. `HashMap`, `LinkedHashMap`, and `SplayTreeMap` are implementations of the `Map` interface.

The collection framework has a couple of unmodifiable implementations of known interfaces, such as `UnmodifiableListView` and `UnmodifiableMapView`.

Many different programs can be written in JavaScript. In the next chapter, you will learn how to communicate with them from Dart. We will shed light on how to use Dart and JavaScript together to build web applications.

7

Dart and JavaScript Interoperation

A lot of web applications are written in JavaScript nowadays and in this chapter, we will focus on how to communicate with them using Dart. We will also cover the following topics:

- The `dart:js` library
- Type conversion
- `JsObject` and instantiation
- `JsFunction` and the `this` keyword
- Dart with jQuery

Interoperation at a glance

Web applications written in Dart can be executed only in a Dart VM that is embedded in a browser called Dartium, a special build of Chromium. To be executed in other web browsers, the Dart code must be compiled to JavaScript—the language supported by all web browsers. Latest tendencies indicate that JavaScript will exist as the language for web development for a long time to come, as it is popular among people and has a lot of tools and frameworks. This means that Dart must have the ability to communicate with the JavaScript code in any web browser.

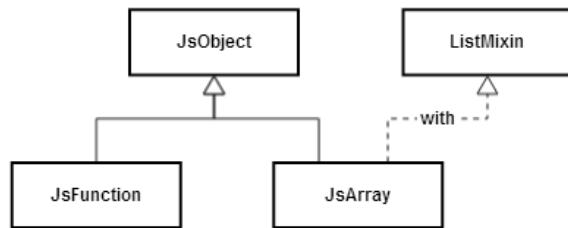


The JavaScript libraries need to be included in HTML before the Dart code.



The dart:js library

The core set of Dart libraries include `dart:js` to facilitate interoperation between the Dart and JavaScript code. The Dart code can create new instances, invoke methods, and read and write properties of the code written in JavaScript. While communicating, the `dart:js` library translates the JavaScript objects to Dart objects and vice versa, or uses proxy classes. Let's take a look at the class hierarchy of the `dart:js` library:



JsObject

Similar to an `Object` class in JavaScript, `JsObject` is a main class in the `dart:js` library. It represents a proxy of a JavaScript object and provides the following advantages:

- Access to all the properties of the underlying JavaScript object by indexing the `[]` and `[] =` operators
- Access to invoke any methods of the underlying JavaScript object through `callMethod`
- Access to the global JavaScript object (usually `window`) in the web browser through the `context` property
- Usage of the `instanceOf` method to check if the underlying JavaScript object has the specified type in its prototype chain
- Checks the existence of the property of the underlying JavaScript object via the `hasProperty` method; this method is equivalent to the `in` operator in JavaScript
- Any property of the underline JavaScript object can be removed with the `deleteProperty` method; this method is equivalent to the `delete` operator in JavaScript

`JsObject` can be acquired from the JavaScript object or can be created using the following factory constructors:

- `factory JsObject(JsFunction constructor, [List arguments]):`
This constructor creates a new JavaScript object from the JavaScript constructor function and returns a proxy on it.
- `factory JsObject.fromBrowserObject(object):` This constructor creates a new JavaScript object from a native Dart object and returns a proxy on it. Use this factory constructor only if you wish to access the properties of the browser-hosted objects such as `Node` or `Blob`. This constructor throws an exception if the object is `null` or has the type `bool`, `num`, or `string`.
- `factory JsObject.jsify(object):` This constructor creates a new JavaScript object or an array from the Dart Map or Iterable and returns a proxy on it. This constructor recursively converts each Dart object from the collection into a JavaScript object.

A library has a top-level context getter that provides the `JsObject` class instance, which represents the JavaScript global object in a web browser, as shown in the following code:

```
import 'dart:js' as js;

void main() {
    print('Context is ${js.context}');
}
```

The print result of the preceding code confirms that the context points to the `Window` object:

```
Context is [object Window]
```

Let's assume we need to log some information on the console from Dart. This can be done as follows:

```
import 'dart:js' as js;

void main() {
    js.JsObject console = js.context['console'];
    console.callMethod('log', ['Hello World!']);
}
```

Firstly, we used `js.context` to receive the proxy of the `JsObject` `console` object. Then, we used `callMethod` of `JsObject` to invoke the `log` function of the `console`. The second optional argument of `callMethod` delivers arguments in the underline JavaScript function. Finally, we get the **Hello World!** message on the web browser's console.

[ The context returns `null` if the requested object does not exist.]

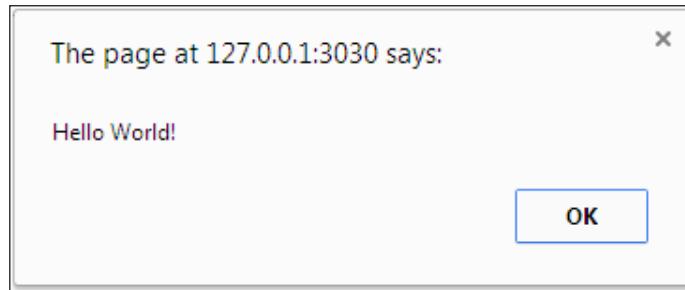
JsFunction

The next important piece of JavaScript is the `Function` type. The `dart:js` library has the `JsFunction` class that extends `JsObject` to represent a proxy to the JavaScript function. To call the `alert` JavaScript function from the Dart code, you can use the following code:

```
import 'dart:js' as js;

void main() {
  js.JsFunction alert = js.context['alert'];
  alert.apply(['Hello World!']);
}
```

When we get a proxy of the `alert` function, we invoke the `apply` method with a list of parameters and get the following result:



JsArray

An array in JavaScript is the object used to store multiple values in a single variable. JsArray extends the JsObject class by representing a JavaScript array and proxy underlying instance to be available in the Dart code. I created a JavaScript code with the colors array, as shown in the following code:

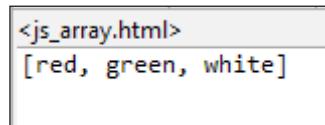
```
var colors = ['red', 'green', 'white'];
```

To get all the colors and print them in Dart, we use the following code:

```
import 'dart:js' as js;

void main() {
  js.JsArray colorsArray = js.context['colors'];
  print(colorsArray);
}
```

Now, we assign a proxy to the array from the colors in JavaScript and finally print them all, as shown in the following screenshot:



If you select the colorsArray property in the variable inspector of Dart Editor when it reaches the breakpoint, you might see a set of items from the JavaScript array:

Name	Value
▲ ● colorsArray	[red, green, white] [id=5]
● class	JsArray [id=6]
▲ ● [[JavaScript View]]	Array[3] [id=1]
● 0	"red"
● 1	"green"
● 2	"white"

Type conversion

All the code we've seen so far was based on the automatic type conversion that happens inside the `dart:js` library. This conversion always happens in both directions. Once you know how this happens, you will have a better understanding of the limits of that solution and it will help you avoid mistakes.

Direct type conversion

Dart supports the following small subset of types by directly converting them from the JavaScript types:

- `null`, `bool`, `num`, `String`, and `DateTime` (basic types)
- `Blob`
- `Event`
- `HtmlCollection`
- `ImageData`
- `KeyRange`
- `Node`
- `NodeList`
- `TypedData` (including its subclasses such as `Int32List`, but not `ByteBuffer`)
- `Window`

Here is set of different types of JavaScript variables that we prepared in the JavaScript file:

```
var v_null = null;
var v_bool = true;
var v_num = 1.2;
var v_str = "Hello";
var v_date = new Date();
var v_blob = new Blob(
  ['<a id="a"><b id="b">hey!</b></a>'],
  {type : 'text/html'});
var v_evt = new Event('click');
var v_nodes = document.createElement("form").children;
var v_img_data = document.createElement("canvas").
  getContext("2d").createImageData(10, 10);
var v_key_range = IDBKeyRange.only(100);
var v_node = document.createElement('div');
var v_node_list = document.createElement('div').childNodes;
var v_typed = new Int32Array(new ArrayBuffer(8));
var v_global = window;
```

Now, we'll use reflection to investigate how conversion happens in Dart in the following code:

```
import 'dart:js' as js;
import 'dart:mirrors';

void main() {
    print(getFromJSContext('v_null'));
    print(getFromJSContext('v_bool'));
    print(getFromJSContext('v_num'));
    print(getFromJSContext('v_str'));
    print(getFromJSContext('v_date'));
    print(getFromJSContext('v_blob'));
    print(getFromJSContext('v_evt'));
    print(getFromJSContext('v_nodes'));
    print(getFromJSContext('v_img_data'));
    print(getFromJSContext('v_key_range'));
    print(getFromJSContext('v_node'));
    print(getFromJSContext('v_node_list'));
    print(getFromJSContext('v_typed'));
    print(getFromJSContext('v_byte_data'));
    print(getFromJSContext('v_global'));
}

getFromJSContext(name) {
    var obj = js.context[name];
    if (obj == null) {
        return name + ' = null';
    } else {
        return name + ' = ' + obj.toString() + ' is ' + getType(obj);
    }
}

getType(obj) {
    Symbol symbol = reflect(obj).type.qualifiedName;
    return MirrorSystem.getName(symbol);
}
```

In the preceding code, the `getFromJSContext` method returned the name, value, and type of the converted JavaScript object. Here is the result printed on the web browser's console:

```
v_null = null
v_bool = true is dart.core.bool
v_num = 1.2 is dart.core._Double
v_str = Hello is dart.core._OneByteString
v_date = 2014-06-21 18:09:03.145 is dart.core.DateTime
v_blob = Instance of 'Blob' is dart.dom.html.Blob
v_evt = Instance of 'Event' is dart.dom.html.Event
v_nodes = [object HTMLCollection] is dart.js.JsObject
v_img_data = Instance of 'ImageData' is dart.dom.html.ImageData
v_key_range = Instance of 'KeyRange' is dart.dom.indexed_db.KeyRange
v_node = div is dart.dom.html.DivElement
v_node_list = [object NodeList] is dart.js.JsObject
v_typed = [0, 0] is dart.typed_data._ExternalInt32Array
v_byte_data = null
v_global = <window> is dart.dom.html.Window
```

You can check which Dart type will be converted from the JavaScript object with the preceding technique.

Proxy type conversion

All the other JavaScript types are converted to Dart types with the help of a proxy. Let's take a look at the following JavaScript code:

```
var Engine = function(type) {
  this.type = type;
  this.start = function() {
    alert('Started ' + type + ' engine');
  };
};

var v_engine = new Engine('test');
```

In the preceding code, we instantiated the `Engine` class and assigned it to the `v_engine` variable. The following code helps to investigate the conversion of the `Engine` class to Dart:

```
import 'dart:js' as js;
import 'dart:mirrors';

void main() {
```

```
    print(getFromJSContext('v_engine'));  
}  
  
getFromJSContext(name) {  
  var obj = js.context[name];  
  if (obj == null) {  
    return name + ' = null';  
  } else {  
    return name + ' = ' + obj.toString() + ' is ' + getType(obj);  
  }  
}  
  
getType(obj) {  
  Symbol symbol = reflect(obj).type.qualifiedName;  
  return MirrorSystem.getName(symbol);  
}
```

In the preceding code, we copied `getFromJSContext` and `getType` from the previous code. Here is the result that is displayed in the web browser console:

```
v_engine = [object Object] is dart.js.JsObject
```

This result confirms that any ordinary object can be converted to Dart with the `JsObject` proxy.

Collection conversion

Dart collections can be converted into JavaScript collections with the `jsify` constructor of `JsObject`. This constructor converts Dart Maps and Iterables into JavaScript objects and arrays recursively, and returns a `JsObject` proxy to it. It supports internal collections as well. The following JavaScript code has the variable `data` and the `toType` and `log` methods:

```
function toType(obj) {  
  return ({}) .toString .call (obj) .  
    match (/ \s ([a-zA-Z]+) / ) [1] .toLowerCase ()  
};  
  
var data;  
  
function log() {  
  console.log(toType(data));  
  for (i in data) {  
    console.log (' - ' + i .toString () + ': ' +  
      data[i] .toString () + ' (' + toType(data[i]) + ')');  
  }  
};
```

Let's take a look at the Dart code that has references to the `log` function of JavaScript:

```
import 'dart:js' as js;

void main() {
  js.JsFunction log = js.context['log'];

  js.JsArray array = new js.JsObject.jsify([1, 'a', true]);
  js.context['data'] = array;
  log.apply([]);

  js.JsObject map = new js.JsObject.jsify(
    {'n':1, 't':'a', 'b':true, 'array':array}
  );
  js.context['data'] = map;
  log.apply[];
}
```

In the preceding code, we created an `array` and assigned it to a JavaScript `data` variable. We logged all the items of the JavaScript `array` via the `apply` method of `log`. We send empty array as argument of the `apply` function because the corresponding JavaScript function doesn't have parameters. Later, we created a `map` object, filled it, and then call a `log` JavaScript function again. To check the support of the internal collections, we inserted the `array` into the `map` object as the last item. Here is the result:

```
array
- 0: 1 (number)
- 1: a (string)
- 2: true (boolean)
object
- n: 1 (number)
- t: a (string)
- b: true (boolean)
- array: 1,a,true (array)
```

All our objects and internal collections were converted into the correct JavaScript objects.

JsObject and instantiation

The object constructor function is a standard way to create an instance of an Object class in JavaScript. An object constructor is just a regular JavaScript function and it is robust enough to define properties, invoke other functions, and do much more. To create an instance of the object, we need to call the object constructor function via a new operator. Let's have a look at the next JavaScript code:

```
function Engine(type) {
  this.type = type;
  this.start = function() {
    console.log ('Started ' + type + ' engine');
  };
}
```

Engine is an object constructor function. It has a `type` property and a `start` method. Here is how we can create an instance of the JavaScript object from Dart:

```
import 'dart:js' as js;

void main() {
  js.JsFunction JsEngine = js.context['Engine'];
  js.JsObject engineObj = new js.JsObject(JsEngine, ['diesel']);
  assert(engineObj.instanceof(JsEngine));
  engineObj.callMethod('start');
}
```

We created a `JsFunction` variable, `JsEngine`, as a reference to the JavaScript object constructor function `Engine`. To create an object type `engineObj`, we used an object constructor created via the `JsObject` proxy. The function arguments must be sent via a second parameter, so we specified `diesel` as the engine type. Later, we check whether `engineObj` is an instance of the `JsEngine` type. Finally, we call the `start` method from `engineObj` and it prints the following message on the console:

```
Started diesel engine
```

If you select the `JsEngine` property in the variable inspector of the Dart Editor when the breakpoint is reached, you will see the source code of the JavaScript function, as shown in the following screenshot:

Name	Value
↳ <code>JsEngine</code>	<pre>function Engine(type) { this.type ... function Engine(type) {} this.type = type; this.start = function() { console.log('Started ' + type + ' engine'); }; }</pre>

JsFunction and the this keyword

The `this` keyword refers to the current instance of a class in Dart and never changes once the `class` object is instantiated. Generally, we should omit the `this` keyword and use it only if we have name conflicts between the class members and function arguments or variables. In JavaScript, the `this` keyword refers to the object that owns the function and behaves differently compared to Dart. It mostly depends on how a function is called. We can't change the value of `this` during function execution and it can be different every time the function is called. The `call` and `apply` methods of `Function.prototype` were introduced in ECMAScript 3 to bind any particular object on the value of `this` in the call of these methods:

```
fun.call(thisArg[, arg1[, arg2[, ...]])  
fun.apply(thisArgs[, argsArray])
```

While the syntax of both these functions looks similar, the fundamental difference is that the `call` method accepts an argument list while the `apply` method accepts a single array of arguments.

All the functions in JavaScript inherit the `call` and `apply` methods from `Function.prototype`, so both the methods invoke the original function and assign the first argument to the value of the `this` keyword permanently so it cannot be overridden.

The following `bind` method of `Function.prototype` was introduced in ECMAScript 5:

```
fun.bind(thisArg[, arg1[, arg2[, ...]])
```

This method creates a new function with the same body and scope as the original function, but the `this` keyword is permanently bound to the first argument of the `bind` function, regardless how the function is being used. The `call` and `bind` methods can solve the issues we face while changing the `this` keyword. Let's take a look at how we can do this with the following JavaScript code:

```
this.name = 'Unknown';

function sendMessage(message) {
    console.log('Send message: ' + this.name + ' ' + message);
}

function DieselEngine() {
    this.name = 'Diesel';
}

sendMessage('engine started');

var engine = new DieselEngine()

var dieselLog = sendMessage.bind(engine);
dieselLog('engine started');
```

The invocation of the `sendMessage` function results in the following console log:

```
Send message: Unknown engine started
```

The `sendMessage` function prints a message on the console with the `Unknown` name because `this` references the global object in the web browser. Then, we create an instance of `Engine` and bind the `sendMessage` function to `engine`. The `bind` method creates a new function `dieselLog` with the same body, but the `this` keyword is permanently bound to `engine`. So, when you call the `dieselLog` function, it uses the name from `DieselEngine` and prints the following message:

```
Send message: Diesel engine started
```

If we need to use the Dart version of `sendMessage` instead of the original one, we can use `JsFunction` instantiated with the `withThis` constructor of the `JsObject` class to call the function with the value of `this` passed as the first argument:

```
import 'dart:js' as js;

void main() {
    js.context['sendMessage'] = new
        js.JsFunction.withThis(otherSendMessage);
```

```
js.JsFunction DieselEngine = js.context['DieselEngine'];
js.JsObject engine = new js.JsObject(DieselEngine);

js.JsFunction sendMessage = js.context['sendMessage'];
sendMessage.apply(['engine started'], thisArg: engine);
}

otherSendMessage(self, String message) {
  print('Message sent: ' + self['name'] + ' ' + message);
}
```

First, we assigned Dart's `otherSendMessage` function to JavaScript's `sendMessage` function. The named constructor `withThis` creates a JavaScript function pattern and uses the reference on `otherSendMessage` instead of `func` in all the future calls:

```
function () {
  return func(this, Array.prototype.slice.apply(arguments));
}
```

So, when we call the `apply` method of `sendMessage`, the JavaScript function calls the original `otherSendMessage` function. It passes the `engine` object to the `self` parameter of the `otherSendMessage` function and passes the `engine started` string in the `message` parameter. The result is printed to the web console:

```
Send message: Unknown engine started
Send message: Diesel engine started
Message sent: Diesel engine started
```

Bear in mind that the type of parameter `self` of the `otherSendMessage` function depends on the value passed as the second argument of the `apply` method of `sendMessage` instance.

Dart with jQuery

There is no doubt that jQuery has become very popular among developers because of its simplicity. Let's try to combine the simplicity of jQuery and the power of Dart in a real example. For demonstration purposes, we created the `js_proxy` package to help the Dart code to communicate with jQuery. It is available on the pub manager at https://pub.dartlang.org/packages/js_proxy. This package is layered on `dart:js` and has a library of the same name and sole class `JProxy`. An instance of the `JProxy` class can be created via the generative constructor where we can specify the optional reference on the proxied `JsObject`:

```
JProxy([this._object]);
```

We can create an instance of `JProxy` with a named constructor and provide the name of the JavaScript object accessible through the `dart:js` context as follows:

```
JProxy.fromContext(String name) {  
  _object = js.context[name];  
}
```

The `JProxy` instance keeps the reference on the proxied `JsObject` class and makes all the manipulation on it, as shown in the following code:

```
js.JsObject _object;  
js.JsObject get object => _object;
```

How to create a shortcut to jQuery

We can use `JProxy` to create a reference to `jQuery` via the context from the `dart:js` library as follows:

```
var jquery = new JProxy.fromContext('jQuery');
```

Another very popular way is to use the dollar sign as a shortcut to the `jQuery` variable as shown in the following code:

```
var $ = new JProxy.fromContext('jQuery');
```

Bear in mind that the original `jQuery` and `$` variables from JavaScript are functions, so our variables reference to the `JsFunction` class. From now, `jQuery` lovers who moved to Dart have a chance to use both the syntax to work with selectors via parentheses.

Why does `JProxy` need a method call?

Usually, `jQuery` sends a request to select HTML elements based on IDs, classes, types, attributes, and values of their attributes or their combination, and then performs some action on the results. We can use the basic syntax to pass the search criteria in the `jQuery` or `$` function to select the HTML elements:

```
$(selector)
```

As mentioned in *Chapter 3, Object Creation*, Dart has a syntactic sugar method, `call`, that helps us to emulate a function and we can use the `call` method in the jQuery syntax. Dart knows nothing about the number of arguments passing through the function, so we use the fixed number of optional arguments in the `call` method. Through this method, we invoke the proxied function (because `jquery` and `$` are functions) and returns results within `JProxy`:

```
dynamic call([arg0 = null, arg1 = null, arg2 = null,
    arg3 = null, arg4 = null, arg5 = null, arg6 = null,
    arg7 = null, arg8 = null, arg9 = null]) {
    var args = [];
    if (arg0 != null) args.add(arg0);
    if (arg1 != null) args.add(arg1);
    if (arg2 != null) args.add(arg2);
    if (arg3 != null) args.add(arg3);
    if (arg4 != null) args.add(arg4);
    if (arg5 != null) args.add(arg5);
    if (arg6 != null) args.add(arg6);
    if (arg7 != null) args.add(arg7);
    if (arg8 != null) args.add(arg8);
    if (arg9 != null) args.add(arg9);
    return _proxify(_object as js.JsFunction).apply(args));
}
```

How does `JProxy` invoke jQuery?

The `JProxy` class is a proxy to other classes, so it marks with the `@proxy` annotation. We override `noSuchMethod` intentionally to call the proxied methods and properties of jQuery when the methods or properties of the proxy are invoked. The logic flow in `noSuchMethod` is pretty straightforward. It invokes `callMethod` of the proxied `JsObject` when we invoke the method on proxy, or returns a value of property of the proxied object if we call the corresponding operation on proxy. The code is as follows:

```
@override
dynamic noSuchMethod(Invocation invocation) {
    if (invocation.isMethod) {
        return _proxify(_object.callMethod(
            symbolAsString(invocation.memberName),
            _jsify(invocation.positionalArguments)));
    } else if (invocation.isGetter) {
        return
            _proxify(_object[symbolAsString(invocation.memberName)]);
    } else if (invocation.isSetter) {
```

```
        throw new Exception('The setter feature was not implemented  
        yet.');
```

```
    }  
    return super.noSuchMethod(invocation);  
}
```

As you might remember, all Map or Iterable arguments must be converted to JsObject with the help of the `jsify` method. In our case, we call the `_jsify` method to check and convert passed arguments aligned with a called function, as shown in the following code:

```
List _jsify(List params) {  
    List res = [];  
    params.forEach((item) {  
        if (item is Map || item is List) {  
            res.add(new js.JsObject.jsify(item));  
        } else {  
            res.add(item);  
        }  
    });  
    return res;  
}
```

Before return, the result must be passed through the `_proxyify` function as follows:

```
dynamic _proxyify(value) {  
    return value is js.JsObject ? new JProxy(value) : value;  
}
```

This function wraps all JsObject classes within a `JProxy` class and passes other values as it is.

An example project

Now create the `jquery` project, open the `pubspec.yaml` file, and add `js_proxy` to the dependencies. Open the `jquery.html` file and make the following changes:

```
<!DOCTYPE html>  
  
<html>  
  <head>  
    <meta charset="utf-8">  
    <meta name="viewport"  
      content="width=device-width, initial-scale=1">  
    <title>jQuery</title>
```

```
<link rel="stylesheet" href="jquery.css">
</head>
<body>
  <h1>Jquery</h1>

  <p>I'm a paragraph</p>
  <p>Click on me to hide</p>
  <button>Click me</button>
  <div class="container">
    <div class="box"></div>
  </div>

</body>

<script src="//code.jquery.com/jquery-1.11.0.min.js"></script>

<script type="application/dart" src="jquery.dart"></script>
<script src="packages/browser/dart.js"></script>
</html>
```

This project aims to demonstrate that:

- Communication is easy between Dart and JavaScript
- The syntax of the Dart code could be similar to the jQuery code

In general, you can copy the JavaScript code, paste it in the Dart code, and probably make slightly small changes.

How to get the jQuery version

It's time to add `js_proxy` in our code. Open `jquery.dart` and make the following changes:

```
import 'dart:html';
import 'package:js_proxy/js_proxy.dart';

/**
 * Shortcut for jQuery.
 */
var $ = new JProxy.fromContext('jQuery');

/**
 * Shortcut for browser console object.
 */
```

```
var console = window.console;

main() {
    printVersion();
}

/***
 * jQuery code:
 *
 * var ver = $().jquery;
 * console.log("jQuery version is " + ver);
 *
 * JS_Proxy based analog:
 */
printVersion() {
    var ver = $().jquery;
    console.log("jQuery version is " + ver);
}
```

You should be familiar with jQuery and console shortcuts by now. The call to `jQuery` with empty parentheses returns `JProxy` and contains `JsObject` with reference to `jQuery` from JavaScript. The `jQuery` object has a `jquery` property that contains the current version number, so we reach this one via `noSuchMethod` of `JProxy`. Run the application, and you will see the following result in the console:

```
jQuery version is 1.11.1
```

Let's move on and perform some actions on the selected HTML elements.

How to perform actions in jQuery

The syntax of jQuery is based on selecting the HTML elements and it also performs some actions on them:

```
$(selector).action();
```

Let's select a button on the HTML page and fire the `click` event as shown in the following code:

```
/***
 * jQuery code:
 *
 * $("button").click(function(){
 *     alert('You click on button');
 * });
 *
```

```
* JS_Proxy based analog:  
*/  
events() {  
    // We remove 'function' and add 'event' here  
    $("button").click((event) {  
        // Call method 'alert' of 'window'  
        window.alert('You click on button');  
    });  
}
```

All we need to do here is just remove the `function` keyword, because anonymous functions on Dart do not use it, and then add the `event` parameter. This is because this argument is required in the Dart version of the event listener. The code calls `jQuery` to find all the HTML button elements to add the `click` event listener to each of them. So when we click on any button, a specified alert message will be displayed. On running the application, you will see the following message:



How to use effects in jQuery

The `jQuery` supports animation out of the box, so it sounds very tempting to use it in Dart. Let's take a look at the following code snippet:

```
/**  
 * jQuery code:  
 *  
 *  $("p").click(function() {  
 *      this.hide("slow",function(){  
 *          alert("The paragraph is now hidden");  
 *      });  
 *  });  
 *  $(".box").click(function(){  
 *      var box = this;  
 *  
 *      $(box).animate({  
 *          width: 0,  
 *          height: 0  
 *      }, 1000);  
 *  });  
 */
```

```
*      startAnimation();
*      function startAnimation(){
*          box.animate({height:300}, "slow");
*          box.animate({width:300}, "slow");
*          box.css("background-color", "blue");
*          box.animate({height:100}, "slow");
*          box.animate({width:100}, "slow", startAnimation);
*      }
*  );
*
* JS_Proxy based analog:
*/
effects() {
    $("p").click((event) {
        $(event['target']).hide("slow", () {
            window.alert("The paragraph is now hidden");
        });
    });
    $(".box").click((event) {
        var box = $(event['target']);
        startAnimation() {
            box.animate({'height':300}, "slow");
            box.animate({'width':300}, "slow");
            box.css("background-color", "blue");
            box.animate({'height':100}, "slow");
            box.animate({'width':100}, "slow", startAnimation);
        };
        startAnimation();
    });
}
}
```

This code finds all the paragraphs on the web page to add a `click` event listener to each one. The JavaScript code uses the `this` keyword as a reference to the selected paragraph to start the hiding animation. The `this` keyword has a different notion on JavaScript and Dart, so we cannot use it directly in anonymous functions on Dart. The `target` property of `event` keeps the reference to the clicked element and presents `JavascriptObject` in Dart. We wrap the clicked element to return a `JProxy` instance and use it to call the `hide` method.

The jQuery is big enough and we have no space in this book to discover all its features, but you can find more examples at https://github.com/akserg/js_proxy.

What is the impact on performance?

Now we should talk about the performance impact of using different approaches across several modern web browsers. The algorithm must perform all the following actions:

- It should create 10000 DIV elements
- Each element should be added into the same DIV container
- Each element should be updated with one style
- All elements must be removed one by one

This algorithm must be implemented in the following solutions:

- The clear jQuery solution on JavaScript
- The jQuery solution calling via JProxy and dart:js from Dart
- The clear Dart solution based on dart:html

We implemented this algorithm on all of them, so we have a chance to compare the results and choose the champion. The following HTML code has three buttons to run independent tests, three paragraph elements to show the results of the tests, and one DIV element used as a container. The code is as follows:

```
<div>
    <button id="run_js" onclick="run_js_test()">Run JS</button>
    <button id="run_jproxy">Run JProxy</button>
    <button id="run_dart">Run Dart</button>
</div>

<p id="result_js"></p>
<p id="result_jproxy"></p>
<p id="result_dart"></p>

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

The JavaScript code based on jQuery is as follows:

```
function run_js_test() {
    var startTime = new Date();
    process_js();
    var diff = new Date(new Date().getTime() -
        startTime.getTime()).getTime();
    $('#result_js').text('jQuery tooks ' + diff +
        ' ms to process 10000 HTML elements.');
}
```

```
function process_js() {
    var container = $('#container');
    // Create 10000 DIV elements
    for (var i = 0; i < 10000; i++) {
        $('<div>Test</div>').appendTo(container);
    }
    // Find and update classes of all DIV elements
    $('#container > div').css("color", "red");
    // Remove all DIV elements
    $('#container > div').remove();
}
```

The main code registers the `click` event listeners and the call function `run_dart_js_test`. The first parameter of the `run_dart_js_test` function must be a function which we will investigate. The second and third parameters are used to pass the selector of the result element and test the title:

```
void main() {
    querySelector('#run_jproxy').onClick.listen((event) {
        run_dart_js_test(process_jproxy, '#result_jproxy', 'JProxy');
    });
    querySelector('#run_dart').onClick.listen((event) {
        run_dart_js_test(process_dart, '#result_dart', 'Dart');
    });
}

run_dart_js_test(Function fun, String el, String title) {
    var startTime = new DateTime.now();
    fun();
    var diff = new DateTime.now().difference(startTime);
    querySelector(el).text = '$title tooks ${diff.inMilliseconds} ms
        to process 10000 HTML elements.';
}
```

Here is the Dart solution based on `JProxy` and `dart:js`:

```
process_jproxy() {
    var container = $('#container');
    // Create 10000 DIV elements
    for (var i = 0; i < 10000; i++) {
        $('<div>Test</div>').appendTo(container.object);
    }
    // Find and update classes of all DIV elements
    $('#container > div').css("color", "red");
    // Remove all DIV elements
    $('#container > div').remove();
}
```

Finally, a clear Dart solution based on `dart:html` is as follows:

```
process_dart() {
    // Create 10000 DIV elements
    var container = querySelector('#container');
    for (var i = 0; i < 10000; i++) {
        container.appendHtml('<div>Test</div>');
    }
    // Find and update classes of all DIV elements
    querySelectorAll('#container > div').forEach((Element el) {
        el.style.color = 'red';
    });
    // Remove all DIV elements
    querySelectorAll('#container > div').forEach((Element el) {
        el.remove();
    });
}
```

All the results are in milliseconds. Run the application and wait until the web page is fully loaded. Run each test by clicking on the appropriate button. My result of the tests on Dartium, Chrome, Firefox, and Internet Explorer are shown in the following table:

Web browser	jQuery framework	jQuery via JProxy	Library <code>dart:html</code>
Dartium	2173	3156	714
Chrome	2935	6512	795
Firefox	2485	5787	582
Internet Explorer	12262	17748	2956

Now, we have the absolute champion—the Dart-based solution. Even the Dart code compiled in the JavaScript code to be executed in Chrome, Firefox, and Internet Explorer works quicker than jQuery (four to five times) and much quicker than `dart:js` and `JProxy` class-based solutions (four to ten times).

Summary

Let me finish the story of Dart to JavaScript interoperation to highlight our expertise.

The core set of Dart libraries include `dart:js` to help you interoperate between the Dart and JavaScript code. The `dart:js` library converts the original JavaScript objects, functions, and collections with the help of `JsObject`, `JsFunction`, and `JsArray`. It supports automatic type of conversion in both directions. Dart supports a small subset of types, transferring directly from JavaScript types. All other JavaScript types are converted to Dart types with the help of a proxy. Dart Map and Iterable collections could be translated into JavaScript collections with the `jsify` constructor of `JsObject`.

We compared `jQuery`, `JProxy`, and `dart:js` and cleared the Dart code based on the `dart:html` solutions to identify who is quicker than the others. The `dart:html` library-based solution is the unbeatable champion and hero of this chapter.

In the next chapter, we will talk about how i18n and l10n accesses can be embedded into our code to help design and develop web applications that enable easy localization for different cultures, regions, and languages.

8

Internalization and Localization

If you are planning to work with multiple languages, you need to add internalization support to your web applications. We will see how the `i18n` and `l10n` access can be embedded in our code to help design and develop web applications that enable easy localization for different cultures, regions, and languages. The topics that will be covered in this chapter are as follows:

- The key principles
- The `intl` library
- How to internationalize a web application
- How to extract messages
- How to use Google Translator Toolkit
- How to use translated messages

The key principles

The development of globalized software is not a simple task. In general, the standard development process to create globalized software includes the following steps:

- Internalization that covers designing and developing web applications
- Localization that covers translating and customizing web applications for a specific locale

We will start designing and developing globalized software with the `Intl` library from the `intl` package available on the pub manager and follow some rules that will help us to easily translate and customize our application.

Executable code versus User Interface

All executable code must be separated from the programming code that implements the **User Interface (UI)**. Also, code that describes the UI elements and the layout of the UI elements must be kept separated from the code that implements and manages them.

Numbers and dates

Various cultures have different ways to represent numbers and dates. You must avoid converting numbers and dates into strings directly.



Converting numbers and dates must be done with special formatters.



Messages

Often, messages that contain individual pieces of text are used together to create complete sentences. In the process of localization, these pieces of text might go together in a different order. Using the `message` method of the `Intl` class allows you to display messages as simple expressions.

Measuring units and currencies

Measuring units such as meters and miles and currencies such as USD and Euro are ubiquitous and depend on the locale. The `NumberFormat` class contains special constructors for the quick creation of frequently used patterns that can be very handy to measure units and currencies.

Text input and layout

The size of the text on the screen is one of the biggest problems that affect programmers who develop globalized software. The main reason for this is that any assumption about the width of the text, the direction of its flow, and its position on the screen, if incorrect, can hamper well-structured layouts in a flash. Using the `BidiFormatter` class helps manage messages that contain text in both the text directionalities.

Formatting date and time

The locale determines how the date and time must be displayed. The `DateFormat` class has a big set of naming patterns that is useful to avoid mistakes and display data in the correct format.

The Intl library

The `Intl` library can help you design and develop globalized server and web client applications. It supports all major locales and sublanguage pairs. As all information on the screen represents a set of strings, we must translate the other types of objects into the string format with special formatters provided by the `Intl` library. Translated versions of displayed strings are bundled in separate text files. The `Intl` library contains a class of the same name that is used to work with messages. The `Intl` library considers numbers, dates, and messages in different internalization areas. Each area is intentionally initialized separately to reduce the application size and avoid loading unnecessary resources. The internalization of numbers and dates is implemented via formatters. Messages are internalized through special functions and can be externalized into files in the **Application Resource Bundle (ARB)** format.

Changing a locale

By default, the current locale is set to the English language of USA (`en_US`). A new value assigned to `defaultLocale` can affect all the methods of the `Intl` library using the following code:

```
Intl.defaultLocale = "fr_FR";
```

There are several ways to use a different locale on a temporary basis than using the current one. They are as follows:

- Specify the locale directly when you call the methods of the `Intl` class
- Provide the locale when you create an instance of the formatter class
- Use the following special `withLocale` method of the `Intl` class:

```
static withLocale(String locale, Function message_function)
```

The main purpose of the `withLocale` method is to delay calling the `message_function` function until the proper locale has been set. The `message_function` function can be a simple message function, a wrapper around the message function, or a complex wrapper that manipulates multiple message functions. As the locale string is not known at the static analysis time, this method silently performs the following steps:

1. It swaps the specified locale string with the current one.
2. Then, it executes the `message_function` function and saves the result.
3. It swaps the locales back.
4. Finally, it returns the result of the `message_function` function.



International Components for Unicode (ICU) is an open source project created for the Unicode support via the implementation of the Unicode standard. The `Intl` library uses the ICU patterns for internalization and localization.

Formatting numbers

The `NumberFormat` class provides the ability to format a number in a locale-specific manner. To create `NumberFormat`, we must specify the pattern in the ICU format, as shown in the following code:

```
var f = new NumberFormat("###.0#");
print(f.format(12.345));
// Result: 12.35
```

The second optional parameter of the `NumberFormat` factory constructor is the locale. If the locale parameter is not specified, the constructor uses a default value in the current locale. The `NumberFormat` class contains the following named constructors for the quick creation of frequently used patterns in a specific locale:

- The decimal format uses the decimal pattern, that is, `, ##0.###`:

```
var d = new NumberFormat.decimalPattern("de_DE");
print(d.format(12.345));
// Result: 12,345
```

- The percent format uses the percent pattern, that is, `,##0%`:

```
var p = new NumberFormat.percentPattern("de_DE");
print(p.format(12.345));
// Result: 1.235%
```

- The scientific format prints only the terms equivalent to `#E0` and does not take into account the significant digits:

```
var s = new NumberFormat.scientificPattern("de_DE");
print(s.format(12.345));
// ==> 1E1
```

- The currency format always uses the name of the currency passed as the second parameter:

```
var c = new NumberFormat.currencyPattern("de_DE", 'EUR');
print(c.format(12.345));
// ==> 12,35EUR
```

Formatting dates

The `DateFormat` class can format and parse the date in a locale-sensitive manner. We can choose the format-parse pattern from a set of standard date and time formats, or we can create a customized one under certain locales. The `DateFormat` class formats the date in the default `en_US` locale without any initialization. For other locales, the formatting data must be obtained and the global `initializeDateFormatting` function must be called to return `Future` that is complete once the locale data is available. Depending on the type of the application you develop, you can choose one of the following libraries that provide this function implementation and enables you to access to the formatting data:

- `date_symbol_data_local`: For a small application, the data to be formatted can be embedded in the code that is available locally so that you can choose the `date_symbol_data_local` library. In the following code, we initialize the date formatting for all the locales at once. Both the parameters of the `initializeDateFormatting` method are ignored because the data for all the locales is directly available, as shown in the following code:

```
import 'package:intl/date_symbol_data_local.dart';
import 'package:intl/intl.dart';

void main() {
    initializeDateFormatting(null, null)
        .then((_) {
        Intl.defaultLocale = "de_DE";
        DateFormat df = new DateFormat("EEE, MMM d, yyyy");
        print(df.format(new DateTime.now()));
    }).catchError((err) {
        print(err);
    });
    // Result: Sa., Sep. 20, 2014
}
```

- `date_symbol_data_http_request`: For the client side, you need an application that runs inside the web browser and possibly compiles into the JavaScript code. You need to read the data from the server using the `XmlHttpRequest` mechanism so that you can choose the `date_symbol_data_http_request` library. We need set up the lookup for the date symbols using URL as a second parameter of the `initializeDateFormatting` method. We use the `path` package that provides common operations to manipulate paths in our example:

```
import 'package:intl/date_symbol_data_http_request.dart';
import 'package:intl/intl.dart';
import 'package:path/path.dart' as path;
```

```
void main() {
    String datesPath = path.join(path.current,
        path.fromUri("packages/intl/src/data/dates/"));
    initializeDateFormatting("pt_BR", datesPath)
    .then((_) {
        Intl.defaultLocale = "pt_BR";
        DateFormat df = new DateFormat("EEE, MMM d, yyyy");
        print(df.format(new DateTime.now()));
    }).catchError((err) {
        print(err);
    });
}
// Result: sáb, set 20, 2014
```

In the preceding code, we requested the date formats for Portuguese – BRAZIL locale and then set them as a default locale. After that, we used `DateFormat` to format current date and time.

- `date_symbol_data_file`: For the server side, you need an application that executes inside the Dart VM so that you can choose the `date_symbol_data_file` library that helps you to read the data from the files in the filesystem. We use the second parameter of the `initializeDateFormatting` method to pass the path to those files. The `path` parameter will end with a directory separator that is appropriate for the platform. We use the `path` package for the following example again:

```
import 'package:intl/date_symbol_data_local.dart';
import 'package:intl/Intl.dart';
import 'package:path/path.dart' as path;

void main() {
    String datesPath = path.join(path.current,
        path.fromUri("packages/intl/src/data/dates/"));
    initializeDateFormatting("fr", datesPath)
    .then((_) {
        DateFormat df = new DateFormat("EEE, MMM d, yyyy",
            "fr_FR");
        print(df.format(new DateTime.now()));
    }).catchError((err) {
        print(err);
    });
}
// Result: sam., sept. 20, 2014
```

When the locale data is ready to use, we need to specify the ICU date/time patterns, which should be used either in full names or preferably their compact skeleton forms as shown in the following code:

```
new DateFormat.yMd(); // Skeleton form
new DateFormat(DateFormat.YEAR_NUM_MONTH_DAY); // ICU full name
// Result: 7/10/2005
```

We can create compound formats with a set of the `add_*` methods as follows:

```
new DateFormat.yMd().add_Hm();
// Result: 7/10/2005 09:10 PM
```

The `DateFormat` class accepts custom formats that follow the explicit pattern syntax. The constructor resolves a custom pattern and adapts it in different locales, as shown in the following code:

```
new DateFormat("EEE, MMM d, yyyy");
// Result: Fri, October 7, 2005
```

The locale is the second optional parameter of the `DateFormat` constructor that helps to create the formatter in a specific locale. The constructor generates `ArgumentError` if a specified locale does not exist in the set of supported locales.

Internalizing messages

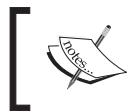
The internalization of messages is based on a lookup via a named localized version of messages and returning the translated message, possibly interpolated with a list of specified arguments. So, to localize any message, such as `Hello $name from Dart!`, we will create a lookup message function that returns the result of the `Intl.message` method call, as shown in the following code:

```
String hello(name) => Intl.message(
    "Hello $name from Dart!",
    name: "hello",
    args: [name],
    examples: { "name": "World" },
    desc: "Greet the user with specified name");
```

We will use the `hello` message function as a wrapper due to the following reasons:

- The function scope can encapsulate an implementation
- The function parameters can be passed as parameters in the `Intl.message` method

The message string that is passed as the first parameter must be a simple expression where only function parameters and curly brackets are allowed. Other parameters must be literal and should not contain any interpolation expressions.



The name and args arguments of the message function are required and the name argument must match the name of the caller function.

Now, instead of assigning the message to our code, we will call the `hello` function to return the translated message for us, as shown in the following code:

```
querySelector("#dart_greeting")
  ..text = hello('John');
```

In other cases that are similar to our example, the `hello` function can interpolate the results of the translated message with a list of arguments that is passed in. The `examples` and `desc` parameters are not used at runtime and are only made available to the translators. Now, we ready to start using the message functions without any localization and will finally have the correct translation in the current locale.

Adding parentheses

We can use parentheses to combine the singular and the plural forms into one string with the `plural` method of the `Intl` class, as shown in the following code:

```
String replace(int num, String str) => Intl.plural(
  num,
  zero: "No one occurrence replaced for $str",
  one: "$num occurrence replaced for $str",
  other: "$num occurrences replaced for $str",
  name: "replace",
  args: [num, str],
  desc: "How many occurrences replaced for string",
  examples: {'num':2, 'str':'hello'});
```

The `plural` method translates and interpolates the contents of the `zero`, `one`, and `other` parameters into a localized message. We missed the `two`, `few`, and `many` parameters because a method can only combine the `one` and `other` methods but you can specify them if necessary. The `plural` method when represented as a `String` expression can be used as part of `Intl.message`, which specifies only the plural attributes as shown in the following code:

```
String replace(int num, String str) => Intl.message(
  """${Intl.plural(
    num,
```

```

        zero: "No one occurrence replaced for $str",
        one: "$num occurrence replaced for $str",
        two: "$num occurrence replaced for $str",
        few: "$num occurrences replaced for $str",
        other: "$num occurrences replaced for $str") }""|,
      name: "replace",
      args: [num, str],
      desc:"How many occurrences replaced for string",
      examples: {'num':2, 'str':'hello'});
  );

```

Adding gender

The `gender` method of the `Intl` class provides out-of-the-box support for gender-based selection, as shown in the following code:

```

String usage(String name, String gender, String car) =>
  Intl.gender(
    gender,
    male: "$name uses his $car",
    female: "$name uses her $car",
    other: "$name uses its car",
    name: "usage",
    args: [name, gender, car],
    desc: "A person uses the car.");

```

The `gender` parameter must equal to one of the literal values: `male`, `female`, or `other`. This method can be used as a part of the `Intl.message` method:

```

String usage(String name, String gender, String car) =>
  Intl.message(
    """${Intl.gender(
      gender,
      male: "$name uses his $car",
      female: "$name uses her $car",
      other: "$name uses its car")}""",
    name: "usage",
    args: [name, gender, car],
    desc: "A person uses the car.");

```

Adding select

Last but not least, a `select` method from the `Intl` class is used to format messages differently, depending on the available choice:

```
String currencySelector(currency, amount) => Intl.select(currency,
{
    "USD": "$amount United States dollars",
    "CDN" : "$amount Canadian dollars",
    "other" : "$amount some currency or other."
},
name: "currencySelector",
args: [currency, amount],
examples: {'currency': 'USD', 'amount':'20'},
desc: "Translate abbreviation into full name of currency");
```

The `select` method looks up the value of the currency in a map of cases and returns the results that are found or returns an empty string. It can be a part of the `Intl.message` method, as shown in the following code:

```
String currencySelector(currency, amount) => Intl.message(
    """${Intl.select(currency,
    {
        "USD": "$amount United States dollars",
        "CDN" : "$amount Canadian dollars",
        "other" : "$amount some currency or other."
    })}""",
    name: "currencySelector",
    args: [currency, amount],
    examples: {'currency': 'USD', 'amount':'20'},
    desc: "Translate abbreviation into full name of currency");
```

Creating complex message translations

The `message`, `plural`, `gender`, and `select` methods can be combined with each other to create complex message translations as shown in the following code:

```
String currencySelector(currency, amount) => Intl.select(currency,
{
    "USD": """${Intl.plural(amount,
        one: '$amount United States dollar',
        other: '$amount United States dollars')}"",
    "CDN": """${Intl.plural(amount,
        one: '$amount Canadian dollar',
        other: '$amount Canadian dollars')}"",
}
```

```
        "other": "$amount some currency or other.",  
    },  
    name: "currencySelector",  
    args: [currency, amount],  
    examples: {'currency': 'USD', 'amount':'20'},  
    desc: "Translate abbreviation into full name of currency");
```

In the preceding code, we translated the abbreviation into a full currency name depending on the amount of money and use them to create the available choice.

Bidirectional formatting

The `Intl` library supports the development of web applications localized for both **right-to-left (RTL)** and **left-to-right (LTR)** locales.

We can combine languages with locales that have different directions in only one text by easily using the HTML markup wrappers. In the following code, we use the `` HTML tags to embed the company name in Hebrew and have the surrounding text in English:

Copyright 2014 דיתע יצו

In cases where the information is entered by the user or if it comes from the backend or third-party web resources, the `BidiFormatter` class can insert it automatically at runtime as shown in the following code:

```
    ב. פותישו דיתע ,copyrightLbl() => Intl.message("Copyright 2014",  
        name: "copyrightLbl",  
        desc: "Copyright label");  
...  
BidiFormatter bidiFormatter = new BidiFormatter.UNKNOWN();  
querySelector("#copyrightLbl").text =  
    bidiFormatter.wrapWithUnicode(copyrightLbl());
```

Internationalizing your web application

Let's see an example of how we can internationalize a standard web application. To do so, we will create a simple web application in Dart Editor, designed the registration form, and embedded it inside the body of an `index.html` file.

The code is as follows:

```
<h1>Registration Form</h1>
<form>
  <table>
    <tr>
      <td><label for="firstName">First Name:</label></td>
```

```
<td><input type="text" id="firstName" name="firstName"></td>
</tr>
<tr>
    <td><label for="lastName">Last Name:</label></td>
    <td><input type="text" id="lastName" name="lastName"></td>
</tr>
<tr>
    <td><label>Gender:</label></td>
    <td>
        <input type="radio" name="sex" value="male">
        <span>Male</span>
        <input type="radio" name="sex" value="female">
        <span>Female</span>
    </td>
</tr>
<tr>
    <td colspan="2"><input type="submit" value="Register"></td>
</tr>
<tr>
    <td colspan="2">
        <span dir="RTL">Copyright 2014 דיבת שיטות</span>
    </td>
</tr>
</table>
</form>
```

The following screenshot shows the result of the preceding code in a web browser:

A screenshot of a web browser displaying a registration form. The form has a title "Registration Form" at the top. It contains fields for "First name" and "Last name" with input boxes. A "Gender" section with radio buttons for "Male" and "Female" is shown. A "Register" button is at the bottom left. At the bottom right, there is a copyright notice in Hebrew: "Copyright 2014 שיטות ועניד". The entire page is displayed in a right-to-left direction.

First of all, we add the `intl` package in the dependency section of our `pubspec.yaml` file. Then, we decide to try out a combination of naming conventions and internationalization methods to offer the utmost flexibility in setting up the display names, because meaningful display names are very important here. To create a display name of a component, we will perform the following actions:

- Make all the levels of the header tag end with the `Head` suffix
- End all the labels with the `Lbl` suffix
- End the input elements to be submitted with the `Btn` suffix

Each referencing element must use the identifier name of the references element in combination with the previously mentioned suffix. So, now we are ready to internationalize the registration form.

All elements in the form that contains the string messages must have unique identifiers. We will remove all the text messages from the registration form, using the following code:

```
<h1 id="formHead"></h1>
<form>
  <table>
    <tr>
      <td><label for="firstName" id="firstNameLbl"></label></td>
      <td><input type="text" id="firstName" name="firstName"></td>
    </tr>
    <tr>
      <td><label for="lastName" id="lastNameLbl"></label></td>
      <td><input type="text" id="lastName" name="lastName"></td>
    </tr>
    <tr>
      <td><label id="genderLbl"></label></td>
      <td>
        <input type="radio" name="sex" value="male">
        <span id="maleLbl"></span>
        <input type="radio" name="sex" value="female">
        <span id="femaleLbl"></span>
      </td>
    </tr>
    <tr>
      <td colspan="2"><input type="submit" id="registerBtn"></td>
    </tr>
    <tr>
      <td colspan="2"><span id="copyrightLbl"></span></td>
    </tr>
  </table>
</form>
```

We need to create one function per message that is getting translated in the Dart code. In such cases, you must choose the naming convention for the following code:

```
formHead() => "Registration Form";
firstNameLbl() => "First name:";
lastNameLbl() => "Last name:";
genderLbl() => "Gender:";
maleLbl() => "Male";
femaleLbl() => "Female";
registerBtn() => "Register";
copyrightLbl() => "Copyright 2014 ©";
```

We will use the `querySelector` function to find all the elements that are translated by the unique identifier and assign the translated messages to their appropriate property, as shown in the following code:

```
querySelector("#formHead").text = formHead();
querySelector("#firstNameLbl").text = firstNameLbl();
querySelector("#lastNameLbl").text = lastNameLbl();
querySelector("#genderLbl").text = genderLbl();
querySelector("#maleLbl").text = maleLbl();
querySelector("#femaleLbl").text = femaleLbl();
(querySelector("#registerBtn") as InputElement)
    .value = registerBtn();
BidiFormatter bidiFormatter = new BidiFormatter.UNKNOWN();
querySelector("#copyrightLbl").text =
    bidiFormatter.wrapWithUnicode(copyrightLbl());
```

Now, apply the `Intl.message` method to every function that's created, as shown in the following code:

```
formHead() => Intl.message("Registration Form",
    name: "formHead",
    desc: "Registration Form title");
firstNameLbl() => Intl.message("First name:",
    name: "firstNameLbl",
    desc: "First Name label");
lastNameLbl() => Intl.message("Last name:",
    name: "lastNameLbl",
    desc: "Last Name label");
genderLbl() => Intl.message("Gender:",
    name: "genderLbl",
    desc: "Gender label");
maleLbl() => Intl.message("Male",
    name: "maleLbl",
    desc: "Male label");
```

```
femaleLbl() => Intl.message("Female",
    name: "femaleLbl",
    desc: "Female label");
registerBtn() => Intl.message("Register",
    name: "registerBtn",
    desc: "Registration Button name");
copyrightLbl() => Intl.message("Copyright 2014 נורית דיתען",
    name: "copyrightLbl",
    desc: "Copyright label");
```

If you open your page in a web browser, it will now look like the original one. Now that you know how messages can be translated with the `Intl` class, it's time to discuss localization.

Extracting messages

Now, we have all the messages separated from the UI and we are ready for translation. We need to extract them from the source code into the external file with a special program called `extract_to_arb.dart` from the `intl` package, as shown in the following code:

```
pub run intl:extract_to_arb --output-dir=web web/registration_form.dart
```

The preceding program generates the `intl_messages.arb` file inside the specified web directory. This file contains all the messages in the ARB format. ARB is a localization resource format based on JSON. This format provides the following benefits:

- **Simplicity:** This format is simple and human-readable because it is based on JSON
- **Extensibility:** In this format, vocabulary can be added without affecting the existing tools and usage
- **Direct usability:** Applications can access the resource directly from this format without converting them to another form

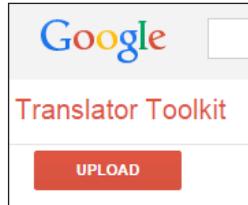
You can find more about the ARB format in the specification available at <https://code.google.com/p/arb/wiki/ApplicationResourceBundleSpecification>.

After this, you can send the files in the ARB format to the human translator. This is done as the form in our case is quite simple and we can translate the files into other languages ourselves by using Google Translator Toolkit.

Using Google Translator Toolkit

The following steps will show you how to use Google Translator Toolkit:

1. Open your web browser and navigate to <https://translate.google.com/toolkit>. Click on the **UPLOAD** button to upload a file as shown in the following screenshot:



2. You will be taken to another web page where you can add the content that you want to translate. You can choose the `intl_messages.arb` file to be translated:

A screenshot of a web browser showing the Google Translator Toolkit interface. The title bar says "Google" and "Translator Toolkit". Below the title is a section titled "What would you like to get translated?". It shows a "Choose File" button with the path "intl_messages.arb" and a dropdown menu set to "English". Below these are two input fields: one containing "intl_messages" and another below it.

3. Select the appropriate translating language that you want the message to be translated into, and then click on the **Next** button:

A screenshot of a web browser showing the Google Translator Toolkit interface. The title bar says "Google" and "Translator Toolkit". Below the title is a section titled "What languages would you like to translate into?". It lists several language options with checkboxes: Arabic, German (which is checked), Portuguese (Portugal), Chinese (Simplified), Italian, and Russian. Below this is a text input field with placeholder text "Type language name to pick more languages". At the bottom left is a "Tools" link, and at the bottom right is a blue "Next" button.

4. On the next page, you can choose any one of the vendors that are ready for translation or click on the **No, thanks** button to translate the messages yourselves. The resulting new file will appear in the list of files ready for translation, as shown in the following screenshot:

NAME	WORDS	LANGUAGE	LAST MODIF	SHARING
intl_messages	14	German	12:04 PM	me

5. Now click on the filename in the list of files and Google Translator Toolkit will open it in the editor:

Original text:

- Message Name: formHead
- Message 1
- Registration Form**
- Description
- Registration Form title
- Message Name: firstNameLbl
- Message 2
- First name:

Translation: English » German 63% complete, 14 words

Machine translation

Anmeldeformular

Characters: 15

Automatic Translation Search Custom Translation Search

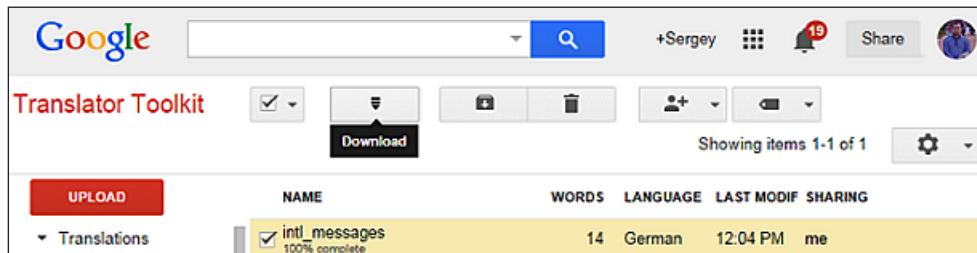
Translation Search Results
No previous translations available.

Computer Translation
Anmeldeformular

Use suggestion

Glossary (0)
No glossary matches available.

6. Google Translator Toolkit helps you translate your messages easily, giving you suggestions. You can just choose all of them and click on the **Complete** button, but you can play with the translation options to properly translate your messages if you want. Finally, choose **Save** and close the menu item in the **File** menu to return to the main page of Google Translator Toolkit:



7. We selected our file and clicked on the **Download** button to save it with the name `translate_de.arb`. We also changed the name of the file from `intl_messages.arb` to `translate_en.arb` to keep all the filenames similar.

Using translated messages

Now, it's time to generate a set of Dart libraries that contain translated versions of our messages—one per locale from the ARB files prepared before. We use the `generate_from_arb` program from the `intl` package:

```
pub run intl:generate_from_arb --output-dir=web web/registration_form.  
dart web/translate_en.arb web/translate_de.arb
```

The program generates the `message_de.dart`, `message_en.dart`, and `messages_all.dart` files in the specified `web` directory. Each `message_<locale_tag>.dart` file contains the `MessageLookup` class that implements `MessageLookupByLibrary`. The `MessageLookup` class has a getter method `localeName`, a set of static functions that are returned translated on the specific locale text messages, and final constant messages that contain the name of all the static methods. The `messages_all.dart` file combines all the lookups in one place to make them available for the localization code from the `Intl` library. The single available public method of the `message_all` library is `initializeMessages`, as shown in the following code:

```
Future initializeMessages(String localeName)
```

This method should be called first before using the specified `localeName` method. Let's change our code to make the German locale available by default. All we need to do is import the `messages_all.dart` file to our project and add `initializeMessages` in the `main` method, as shown in the following code:

```
import 'messages_all.dart';
...
void main() {
  initializeMessages('de').then((_) {
    Intl.defaultLocale = 'de';
    querySelector("#formHead").text = formHead();
    querySelector("#firstNameLbl").text = firstNameLbl();
    querySelector("#lastNameLbl").text = lastNameLbl();
    querySelector("#genderLbl").text = genderLbl();
    querySelector("#maleLbl").text = maleLbl();
    querySelector("#femaleLbl").text = femaleLbl();
    (querySelector("#registerBtn") as InputElement)
      .value = registerBtn();
    BidiFormatter bidiFormatter = new BidiFormatter.UNKNOWN();
    querySelector("#copyrightLbl").text =
      bidiFormatter.wrapWithUnicode(copyrightLbl());
  });
}
```

In the preceding code, we specified the German locale when we initialized the messages as default. Now, open the `index.html` file in the browser to see the correct translation of our form in the German locale:

The screenshot shows a registration form with the following elements:

- Title:** Registrierungsformular
- Fields:**
 - Vorname: [Input field]
 - Nachnamen: [Input field]
- Gender Selection:** Geschlecht: Männchen Weibchen
- Submit Button:** Registrieren
- Copyright Notice:** © Urheberrecht 2014

Summary

In this chapter, we discussed some important aspects of internalization and localization of projects based on the Dart language. The development of globalized software includes internationalization and covers designing and developing web applications, and localization that includes translating and customizing web applications for a specific locale.

The `intl` package from the pub manager helps you design and develop applications for the server side and client side in a pretty straightforward manner. All the executable code must be separated from the programming code that implements the UI. You must avoid converting numbers and dates into strings directly because various cultures have different ways of presenting numbers and dates.

The `NumberFormat` class contains special constructors to quickly create the frequently used patterns, which can be very handy to measure units and currencies. The `BidiFormatter` class helps manage messages that contain text in both directions with the `BIDI` wrapping, automatic directionality estimation, and character escaping. The `DateFormat` class has a huge set of naming patterns that are useful to avoid mistakes and display data in the correct format. The internalization of messages is based on the lookup via the named localized version of messages and returns the translated messages, possibly those interpolated with a list of specified arguments.

The `plural` method of the `Intl` class can help use parentheses to combine the singular and the plural forms to one string. The `gender` method of the `Intl` class provides out-of-the-box support for gender-based selection. The `select` method from the `Intl` class helps format the message differently based on the available choice. The `message` method of the `Intl` class can represent messages as simple expressions with possibilities of interpolating them with attributes.

In the next chapter, we will show you how to properly organize client to server communication. You will find answers on presumably important questions about the right choice of Dart classes using the client-to-server communication.

9

Client-to-server Communication

In this chapter, we will discuss how to organize client-to-server communication. We will find answers to presumably the important questions, such as the right choice of Dart classes using the client-to-server communication. In this chapter, we will cover the following topics:

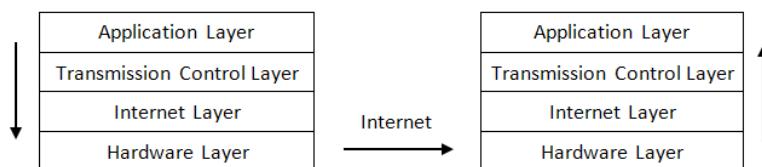
- Communication at a glance
- Hypertext Transfer Protocol
- AJAX polling request
- AJAX long polling request
- Server-Sent Events
- WebSocket

Communication at a glance

Some of us can't imagine the modern world without the Internet, cell phones, or computers. Each device connected to the Internet can either be a client, a server, or both of these simultaneously. Communication between a client and a server is the basis of modern digital world. Communication based on a system of special rules and format of messages is known as a communication protocol that enables data exchange between clients and servers.

The Internet protocol stack

Any device connected to the Internet has an **Internet Protocol (IP)** address. An IP address can be permanent or temporary when it is obtained from the **Dynamic Host Configuration Protocol (DHCP)** server. In any case, any device that is connected to the Internet has a unique IP address. Many programs working in scope of a single unique IP address use different port numbers to have access to the Internet simultaneously. A message transmitted from one device to another over the Internet is delivered through a long route via the protocol stack, which is represented as a set of layers that lie on top of one another as shown in the following diagram:



Let's see what actually happens with a message:

1. A message arrives at the application protocol layer present on the top of the protocol stack, and then moves down. Usually, the application layer formats the message in one of the standard ways that is applicable to applications such as HTTP, SMTP, FTP, or others.
2. The formatted message is then forwarded to the **transmission control protocol (TCP)** layer. It splits the message into small, manageable chunks of data known as packets and assigns a number to each of them. This number specifies the order of the packets and allows the recipient's TCP layer to reconstruct the original message from the packets. It assigns a port number to each packet, depending on the protocol being used at the application level.
3. Then, these packets proceed to the IP layer. It attaches the IP address of the sender and recipient to each packet. A combination of the IP address and port number is called a socket address.
4. Finally, a hardware layer attaches the **Media Access Control (MAC)** address of the sender and recipient to the packets. It allows the packets to be directed to a specific network interface on the IP address of the destination device. On the hardware layer, all packets are converted to electronic signals one by one, transmitted over the wire, and connected to the **Internet Service Provider (ISP)** modem.
5. From here, it is the ISP's task to deliver packets of message to the specified IP address on the Internet via routers and ISP backbones.

6. Packets delivered by the ISP start at the bottom of the protocol stack of the destination device. Any extra information stripped from the packets goes upwards.
7. Eventually, data reaches the top of the stack where it is decoded into the original message.

Hypertext Transfer Protocol

The **World Wide Web (WWW)** is one of the most commonly used services on the Internet. The **Hypertext Transfer Protocol (HTTP)** is a text-based application layer protocol that makes it work. All web browsers and web service applications use HTTP to communicate with each other over the Internet. A web browser or standalone application opens a connection and sends a request to the web server. The web server services the request and closes the connection to the web client.



You cannot find any information about persistent connections in the HTTP 1.0 specification. However, in essence, it was unofficially added to an existing protocol via the following additional header to the request:

```
Connection: Keep-Alive
```

So, if a client supports `Keep-Alive`, it adds the preceding header to his request. A server receives this request and generates a response includes this header.

Starting from HTTP 1.1, all the connections are considered persistent unless declared otherwise. A HTTP persistent connection does not use external `Keep-Alive` messages. Multiple requests could be sent to use a single opened connection.

We can use the HTTP protocol for communication purposes via the following different libraries:

- The `dart:io` library from the Dart SDK contains the `HttpClient` class, which communicates with the server over the HTTP protocol.
- The `dart:html` library from the Dart SDK has the `HttpRequest` class uses a client-side `XMLHttpRequest` to obtaining data from the URL. It also helps in obtaining data from HTTP or FTP, or updating page content via AJAX.

- The http package from the pub server written by the Google development team contains a future-based library to create HTTP requests. It is platform independent, so we can use the http.dart library to generate HTTP requests from standalone applications or the browser_client.dart library for web browser-based applications.

Let's see how we can organize communication between the web browser or standalone application on one side and the web server on the other side.

Web server

In this chapter, we will create a simple web server that can be used for all examples as follows:

```
import 'dart:io';
import 'package:route/server.dart';

import 'urls.dart' as urls;
import 'files.dart';
main() {
  final allUrls = new RegExp('/(.*)');

  HttpServer.bind(urls.serverAddress, urls.serverPort)
    .then((server) {
      print("Server runs on ${server.address.host}:${server.port}");
      new Router(server)
        ..serve(urls.dataUrl, method:'GET').listen(processParams)
        ..serve(urls.dataUrl, method:'DELETE').listen(processParams)
        ..serve(urls.dataUrl, method:'POST').listen(processBody)
        ..serve(urls.dataUrl, method:'PUT').listen(processBody)
        ..serve(allUrls).listen(serveDirectory('', as: '/'))
        ..defaultStream.listen(send404);
    });
}
```

In the preceding code, we intentionally used the route package from the pub server to reduce the number of code wraps around the main functionality. Our code serves the client's request to match dataUrl. Depending on the method of request, the code invokes the processParams or processBody functions. We keep serverAddress, serverPort, and dataUrl inside the urls.dart file. We especially move them away to the external file to share this data with client code. We set the shared headers in the setHeaders method as shown in following code:

```
setHeaders(HttpRequest request) {
  request.response.headers.contentType =
    new ContentType("text", "plain", charset: "utf-8");
}
```

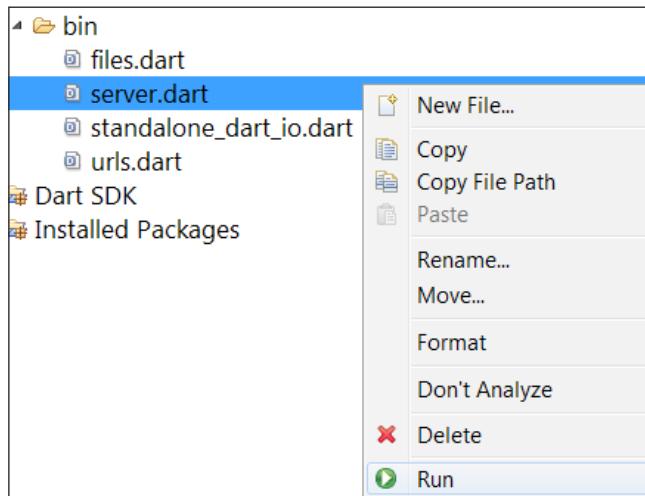
The following method processes the query parameters:

```
processParams(HttpServletRequest request) {  
    setHeaders(request);  
    request.response.write(  
        "${request.method}: ${request.uri.path}");  
    if (request.uri.queryParameters.length > 0) {  
        request.response.write(", Params:" +  
            request.uri.queryParameters.toString());  
    }  
    request.response.close();  
}
```

The following method processes the requested message body. Depending on the amount of the content, we write the content in the output stream or just return the message with the method name and path, as follows:

```
processBody(HttpServletRequest request) {  
    setHeaders(request);  
    if (request.contentLength > 0) {  
        request.listen((List<int> buffer) {  
            request.response.write(  
                "${request.method}: ${request.uri.path}");  
            request.response.write(", Data:" +  
                new String.fromCharCodes(buffer));  
            request.response.close();  
        });  
    } else {  
        request.response.write(  
            "${request.method}: ${request.uri.path}");  
        request.response.close();  
    }  
}
```

Our server doesn't do anything special. It responds with the method's name, query parameters, or the message body, which is represented as a string per request. The server must always be online to handle our requests. Dart Editor can run different Dart programs simultaneously. To do this, just right-click on the `server.dart` file and run it as shown in the following screenshot:



Standalone HTTP communication via the `dart:io` library

Let's start from the standard `dart:io` library and the `HttpClient` class to organize communication from the standalone client to the web server. Any method of the `HttpClient` class that is used for communication is a two-step process. The first time we call the original method, it returns `Future<HttpClientRequest>`. From now, the underlying network communication is opened but no data is sent yet. You can set the HTTP headers or body on the request and finally return the results of the request's `close` method. Take a look at the following code based on the `HttpClient` class from the `dart:io` library:

```
import 'dart:io';
import 'dart:convert';
import 'urls.dart' as urls;

var url = "http://${urls.serverAddress}:${urls.serverPort}${urls.
    dataUrl}";
var name = "HttpClient Standalone";
// The following method processes all responses:
responseHandler(response) {
```

```

if (response is HttpClientResponse) {
    response.transform(UTF8.decoder).listen((contents) {
        print("${response.statusCode}: ${contents}");
    });
} else {
    print("Readed: " + response.toString());
}
}

// We send data to a server via this method:
sendData(HttpClientRequest request, data) {
    request.headers.contentType =
        new ContentType("application", "json", charset: "utf-8");
    List<int> buffer = JSON.encode(data).codeUnits;
    request.contentLength = buffer.length;
    request.add(buffer);
}

main() {
    // We need to encode name before sending it:
    String query = "name=" + Uri.encodeQueryComponent(name);
    // We create the client instance to send multiple requests:
    HttpClient client = new HttpClient();
    // Make a GET request with query:
    client.getUrl(Uri.parse("$url?$query"))
        .then((HttpClientRequest request) {
            return request.close();
        }).then(responseHandler);
    // Here we send a map with data via the POST request:
    client.postUrl(Uri.parse(url))
        .then((HttpClientRequest request) {
            sendData(request, {'post name': name});
            return request.close();
        }).then(responseHandler);
    // The PUT request is very similar to the POST one:
    client.putUrl(Uri.parse(url))
        .then((HttpClientRequest request) {
            sendData(request, {'put name': name});
            return request.close();
        }).then(responseHandler);
    // Here is the DELETE request:
    client.deleteUrl(Uri.parse("$url?$query"))
        .then((HttpClientRequest request) {
            return request.close();
        }).then(responseHandler);
}

```

Run the server code as explained in the *Web server* section and then run the code in `standalone_dart_io.dart` via the context-sensitive menu. Refer to the following client output:

```
200: GET: /data, Params:{name: HttpClient Standalone}
200: DELETE: /data, Params:{name: HttpClient Standalone}
200: POST: /data, Data:{"post name":"HttpClient Standalone"}
200: PUT: /data, Data:{"put name":"HttpClient Standalone"}
```

The `HttpClient` class provides a set of methods to create HTTP requests but a two-step process is a real disadvantage.

Standalone HTTP communication via the http package

Let's see how the `http` library from the `http` package can improve the client-side development experience. Before using the `http` library, we should add the `http` package in a group of dependencies in the `pubspec.yaml` file of our project. We create a `standalone_http.dart` file with the help of the following code:

```
import 'package:http/http.dart' as http;
import 'dart:async';
import 'urls.dart' as urls

var url = "http://${urls.serverAddress}:${urls.serverPort}${urls.
    dataUrl}";
var name = "Http Standalone";
// We process all responses in this method:
responseHandler(response) {
    if (response is http.Response) {
        print("${response.statusCode}: ${response.body}");
    } else {
        print("Readed: " + response.toString());
    }
}

main() {
    // We need to encode name before sending it:
    String query = "name=" + Uri.encodeQueryComponent(name);
    // Static functions such as GET, POST, and so on create new
    instances of the Client interface per request:
    // All static functions such as get from the http library always
    create
    // new instance of the Client class
    http.get("$url?$query").then(responseHandler);
```

```
var client = new http.Client();
Future.wait([
  client.get("$url?$query").then(responseHandler),
  client.post(url, body: {"name": name}).then(responseHandler),
  client.put(url, body: {"name": name}).then(responseHandler),
  client.delete("$url?$query").then(responseHandler)])
.then((list) {
  client.close();
});
}
```

A huge advantage of using the `Client` class from the `http` library over `HttpClient` from the `dart:io` library is less verbose code with a similar result:

```
200: GET: /data, Params:{name: Http Standalone}
200: GET: /data, Params:{name: Http Standalone}
200: POST: /data, Data:name=Http+Standalone
200: PUT: /data, Data:name=Http+Standalone
200: DELETE: /data, Params:{name: Http Standalone}
```

Web browser HTTP communication via the `dart:html` library

You cannot use the `dart:io` library to write a web browser-based application because this library was written especially for standalone and server applications. Instead, we will use the `HttpRequest` class from the `dart:html` library to achieve the same result. This class can be used to obtain data from the HTTP or FTP application protocols as well as AJAX polling requests, as shown in the following code:

```
import 'dart:html';
import 'dart:convert';
import 'urls.dart' as urls;

var url = "http://${urls.serverAddress}:${urls.serverPort}${urls.dataUrl}";
var name = "HttpClient Browser";
// We save the response text in a DIV element and append it to the
// DIV container:
responseHandler(DivElement log, responseText) {
  DivElement item = new DivElement()
    ..text = responseText.toString();
  log.append(item);
}

void main() {
  DivElement log = querySelector("#log");
  // Here we prepare query to send:
  String query = "name=" + Uri.encodeQueryComponent(name);
  String data = JSON.encode({"name": name});
```

```
HttpRequest request = new HttpRequest();
// We open a connection to the server via HttpRequest:
request.open("GET", "$url?$query");

request.onLoad.listen((ProgressEvent event) {
    responseHandler(log, request.response);
});
// Now send data via call the send method:
request.send();

request.open("POST", url);
request.onLoad.listen((ProgressEvent event) {
    responseHandler(log, request.response);
});
request.send(data);

request.open("PUT", url);
request.onLoad.listen((ProgressEvent event) {
    responseHandler(log, request.response);
});
request.send(data);

request.open("DELETE", "$url?$query");
request.onLoad.listen((ProgressEvent event) {
    responseHandler(log, request.response);
});
request.send();
// HttpRequest supports less verbose code:
HttpRequest.request("$url?$query")
.then((HttpRequest request)
=> responseHandler(log, request.response));

HttpRequest.request(url, method: "POST", sendData: data)
.then((HttpRequest request)
=> responseHandler(log, request.response));

HttpRequest.request(url, method: "PUT", sendData: data)
.then((HttpRequest request)
=> responseHandler(log, request.response));

HttpRequest.request("$url?$query", method: "DELETE")
.then((HttpRequest request)
```

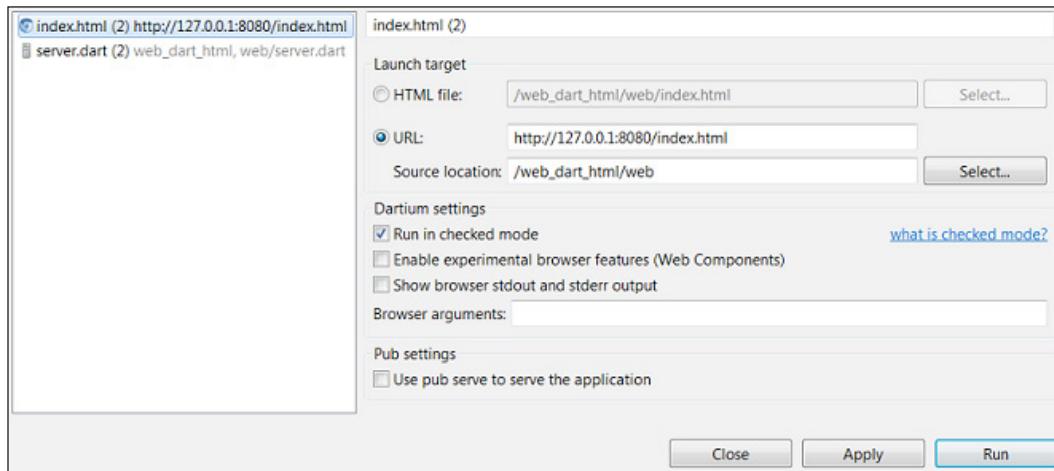
```

=> responseHandler(log, request.response)) ;

// The getString method is the absolute champion of size
// if you need a simple GET request:
HttpRequest.getString("$url?$query")
.then((response) => responseHandler(log, response));

```

We now create a launch target to run the `index.html` file via the **Manage Launches** item from the **Run** menu, as shown in the following screenshot:



This Dartium launch configuration opens the `index.html` file in the default web browser. Take into account the fact that the **Use pub service to serve the application** option is unchecked because we are using our own server to serve all the browser requests. You could set breakpoints and debug code if necessary. Run the server code as mentioned in the *Web server* section and `index.html` through the launcher. The following is the result of our requests when printed on the web page:

```

DELETE: /data, Params:{name: HttpClient Browser}
GET: /data, Params:{name: HttpClient Browser}
POST: /data, Data:{ "name": "HttpClient Browser" }
DELETE: /data, Params:{name: HttpClient Browser}
PUT: /data, Data:{ "name": "HttpClient Browser" }
GET: /data, Params:{name: HttpClient Browser}

```

Web browser HTTP communication via the http package

As mentioned earlier, the `http` package combines two sorts of libraries to help in client-to-server communication. Let's see how the `BrowserClient` class from the `http` package can help us achieve the same result with less effort:

```
import 'dart:html' as dom;
import 'package:http/browser_client.dart';
import 'package:http/src/request.dart';
import 'package:http/src/streamed_response.dart';
import 'dart:convert';
import 'urls.dart' as urls;

var url = "http://${urls.serverAddress}:${urls.serverPort}${urls.
  dataUrl}";
var name = "HttpClient Browser";
// The response handler is as follows:
responseHandler(domDivElement log, StreamedResponse response) {
  domDivElement item = new domDivElement();
  response.stream.transform(UTF8.decoder).listen((contents) {
    item.text = contents;
  });
  log.append(item);
}

void main() {
  domDivElement log = dom.querySelector("#log");
  String query = "name=" + Uri.encodeQueryComponent(name);
  String data = JSON.encode({"name": name});

  BrowserClient client = new BrowserClient();
  Request request = new Request("GET", Uri.parse("$url?$query"));
  // We organize request via call the send method of BrowserClient
  // class:
  client.send(request).then((StreamedResponse response)
    => responseHandler(log, response));

  request = new Request("POST", Uri.parse(url));
  request.body = data;
  client.send(request).then((StreamedResponse response)
    => responseHandler(log, response));
}
```

```

request = new Request("PUT", Uri.parse(url));
request.body = data;
client.send(request).then((StreamedResponse response)
    => responseHandler(log, response));

request = new Request("DELETE", Uri.parse("$url?$query"));
client.send(request).then((StreamedResponse response)
    => responseHandler(log, response));
}

```

Create a Dartium launch configuration and open the `index.html` file in a web browser. Run the server and launch the new configuration to see the following expected result:

```

GET: /data, Params:{name: HttpClient Browser}
POST: /data, Data:{ "name":"HttpClient Browser"}
DELETE: /data, Params:{name: HttpClient Browser}
PUT: /data, Data:{ "name":"HttpClient Browser"}

```

You now know how to easily create client-to-server communication via the `HttpClient` and `Request` classes from the `http` and `html` packages.

AJAX polling request

Usually, a web browser sends requests and immediately receives responses. Every new portion of data requested from the server starts to reload a whole web page. **Asynchronous JavaScript and XML (AJAX)** is a technology that allows the client-side JavaScript code to request data from a server without the need to reload the current page in the web browser. An important advantage of using AJAX compared to traditional flow of web pages is its ability to bring usability and behavior of desktop applications into a web application. `HttpRequest` is a client-side XHR request to get data from a URL, formally known as `XMLHttpRequest`, and has an important role in the AJAX web development technique.



All the requests created within `HttpRequest` must be in the server from the same origin as the requested resource.

The state of many web applications is inherently volatile. Changes can come from numerous sources, such as other users, external news and data, results of complex calculations, and triggers based on the current time and date. The solution is to periodically issue a request to gain new information. For this purpose, we can use the AJAX polling request technique. Let's take a look at the client side of our web application:

```
import 'dart:html';
import 'dart:async';
import 'urls.dart' as urls;

var url = "http://${urls.serverAddress}:${urls.serverPort}${urls.
dataUrl}";
```

To print the timestamp, use the following code:

```
timeStamp() {
    DateTime dt = new DateTime.now();
    return " (${dt.minute}:${dt.second}) ";
}
```

The response keeps a request number as follows:

```
responseHandler(DivElement log, responseText) {
    DivElement item = new DivElement()
    ..text = responseText.toString() + timeStamp;
    // We insert a new element at the top of the log:
    log.insertAdjacentElement('beforebegin', item);
}

void main() {
    DivElement log = querySelector("#log");
    int num = 0;
    // A timer in 2 seconds interval calls the callback function:
    new Timer.periodic(new Duration(seconds: 2), (Timer timer) {
        // We generate the query parameter with number:
        String query = "number=${num++}";
        HttpRequest.getString("$url?$query")
            .then((response) => responseHandler(log, response));
    });
}
```

This is a simple implementation that creates new requests in every 2 seconds. Each request has a unique number; when the server returns it, this helps us check the order of responses. We'll make small changes to the server code to add some random generated data to the response. The `readyChecker` variable is a random generator whose values determine whether the data is ready to be fetched. The `dataGenerator` variable is a random generator of integer numbers in a range between 0 and 100, as shown in the following code:

```
import 'dart:math';
Random readyChecker = new Random(
    new DateTime.now().millisecondsSinceEpoch);
Random dataGenerator = new Random(
    new DateTime.now().millisecondsSinceEpoch);
//...
In the fetchData function, we check whether the data is ready to be
fetched or returns an empty string:
String fetchData() {
    if (ready.nextBool()) {
        return data.nextInt(100).toString();
    } else {
        return "";
    }
}
```

In the following code, we add the fetched data to the response:

```
processQueryParams(HttpRequest request) {
    setHeaders(request);
    request.response.write(
        "${request.method}: ${request.uri.path}");
    request.response.write(", Data:" + fetchData());
    if (request.uri.queryParameters.length > 0) {
        request.response.write(", Params:" +
            request.uri.queryParameters.toString());
    }
    request.response.close();
}
```

Create a Dartium launch configuration and open the `index.html` file in the web browser. Run the server and launch a new configuration to see the following result:

```
GET: /data, Data:4, Params:{number: 0} (31:28)
GET: /data, Data:26, Params:{number: 1} (31:30)
GET: /data, Data: , Params:{number: 2} (31:32)
GET: /data, Data: , Params:{number: 3} (31:34)
GET: /data, Data:47, Params:{number: 4} (31:36)
GET: /data, Data:69, Params:{number: 5} (31:38)
GET: /data, Data: , Params:{number: 6} (31:40)
GET: /data, Data: , Params:{number: 7} (31:42)
GET: /data, Data:98, Params:{number: 8} (31:44)
```

The polling requests have the following disadvantages:

- Periodical requests consume bandwidth and server resources
- The server must always respond with or without data; as a result, the client receives a lot empty strings

Let's try to use the AJAX long polling request to fix them.

AJAX long polling request

From a client perspective, the AJAX long polling request looks similar to normal one. An important difference between them on the server side is that if the server doesn't have any information available or the client, it holds the request and waits for information to become available or for a suitable timeout, after which a completed response is sent to the client. The long polling requests reduces the amount of data that needs to be sent because the server only sends data when it is really available.

The long pooling request is useful in the following cases:

- When your solution must work in old web browsers
- When the data traffic between the client and server is low
- When the implementation is very simple

The advantages of using the long pooling request are as follows:

- It works across all browsers
- It is easy to develop and perfectly fits in the legacy code without significant changes and effort
- It can detect a connection failure quickly and resume a session to avoid data loss

- It has a strong immunity against IP address changes for free, because requests are short-lived and their state is stored independently
- It performs well for most real-time applications and reduces bandwidth consumption and server resources utilization

The disadvantages of using this request are as follows:

- The client constantly has to establish connections; if the server sends the information back to the client and the connection is closed, it must wait until the client sends the next request to open a new connection
- The client needs to create an extra connection to send data to the server
- There are problems with parallel requests
- The volume of data can make the next update from the server quite excessive if the data comes to the server until then it waits for the new connection from the client

The following browsers are supported by the long pooling request:

- Chrome 1.0+
- Firefox 0.6+
- Opera 8.0+
- Safari 1.2+
- Internet Explorer 5.0+

To implement a long-lasting HTTP connection, you need to make changes on the client side and the server side. We introduce the `longLasting` method to manage requests and register it instead of using `processParams`:

```
//...
main() {
    final allUrls = new RegExp('/(.*)');

    HttpServer.bind(urls.serverAddress, urls.serverPort)
        .then((server) {
            print("Server runs on ${server.address.host}:${server.port}");
            new Router(server)
                ..serve(urls.dataUrl, method: 'GET').listen(longLasting)
                ..serve(allUrls).listen(serveDirectory('', as: '/'))
                ..defaultStream.listen(send404);
        });
}
```

```
// Next function was changed to return data only:  
String fetchData() {  
    return dataGenerator.nextInt(100).toString();  
}
```

The new `longLasting` function is shown in the following code. It runs two periodic timers. The first one emulates the process based on the request timeout. If no data was collected every 30 seconds, it resets the second timer and response with an empty string. The second one is managing the data availability. If the data is available in a second's interval, it resets the first timer and responds with the collected data:

```
longLasting(HttpServletRequest request) {  
    Timer reqTimer, dataReadyTimer;  
  
    reqTimer = new Timer.periodic(  
        new Duration(seconds:30), (Timer t) {  
            dataReadyTimer.cancel();  
            processParams(request);  
        });  
  
    dataReadyTimer = new Timer.periodic(  
        new Duration(seconds:1), (Timer t) {  
            if (ready.nextBool()) {  
                reqTimer.cancel();  
                processQueryParams(request, ready:true);  
            }  
        });  
}
```

The updated version of the `processQueryParams` function is illustrated in the following code. This function returns data from `fetchData` or returns an empty string depending on the value of the `ready` attribute. The code is as follows:

```
processQueryParams(HttpServletRequest request, {bool ready:false}) {  
    request.response.write(  
        "${request.method}: ${request.uri.path}");  
    if (ready) {  
        request.response.write(", Data:" + fetchData());  
    } else {  
        request.response.write(", Data:");  
    }  
    if (request.uri.queryParameters.length > 0) {
```

```
    request.response.write("", Params:" +
        request.uri.queryParameters.toString());
    }
    request.response.close();
}
```

Let's take a look at how we change the AJAX polling request example to organize the long polling request at the client side. You can initiate the long polling request by calling the `longPolling` function as follows:

```
void main() {
    DivElement log = querySelector("#log");
    int num = 0;
    longPolling(log, num);
}
```

In this function, we send the AJAX request and wait for the response from the server. When the response is available, we call the `responseHandler` function and start the new request immediately:

```
longPolling(DivElement log, int num) {
    String query = "number=${num++}";
    HttpRequest.getString("$url?$query")
        .then((response) {
            responseHandler(log, response);
            longPolling(log, num);
        });
}
```

Create a Dartium launch configuration, run the server, and open the `index.html` file in the web browser to see the following result:

```
GET: /data, Data:12, Params:{number: 0} (51:0)
GET: /data, Data:9, Params:{number: 1} (51:2)
GET: /data, Data:60, Params:{number: 2} (51:3)
GET: /data, Data:41, Params:{number: 3} (51:5)
GET: /data, Data:15, Params:{number: 4} (51:7)
GET: /data, Data:0, Params:{number: 5} (51:8)
```

For now, we reduced the consumption of bandwidth and server utilization; hence, the clients will mostly receive only valid data.

Server-Sent Events

Another AJAX-based technique is **Server-Sent Events (SSE)**, which is also known as **Server Push** or **HTTP Streaming**. In this, the client opens a connection to the server via an initial HTTP request, and the server sends events to the client when there is new information available. So, if the usual functions of your clients are similar to stock tickers or news feeds and they need updates from the server with time, then the SSE technique is the ideal solution for you. By the way, if a client has new information to send to the server, it can send it through a new HTTP request.

SSE is useful in the following cases:

- The client is oriented towards receiving large volume of data
- The solution must work in old web browsers

The advantages of SSE are as follows:

- The server implementation is simple enough
- A web browser can automatically reconnect to the server
- The format of exchanging messages is flexible enough
- The solution is based on one permanent connection to the server
- The solution well enough for a real-time application
- Clients don't need to establish a new connection after every response
- Server solutions can be based on the event loop

The disadvantages of SSE are as follows:

- It works only from a server to client
- Internet Explorer doesn't support it
- It can be blocked by proxy servers
- It is impossible to connect to the server from another domain

The following browsers are supported:

- Chrome 6.0+
- Firefox 6.0+
- Opera 9.0+
- Safari 5.0+

The `EventSource` class is used to receive SSE. This class connects via a specified URL to a server over HTTP and receives server events without closing the connection. The events come in a `text/event-stream` format. To open a connection to a server and start receiving events, we create a new instance of the `EventSource` class via a factory constructor and pass the URL of the resource that generates the events through it. Once the connection is established, we wait for the messages.

 The URL of the resource that generates the events must match the origin of the calling page.

On the server side, each message is sent as a block of text terminated by a pair of new lines. The text data is encoded with UTF-8. Each message consists of one or more lines of text listing the fields for that message. Each field is represented by the field name, followed by a colon and the text data for that field's value. Here is an example of data-only messages:

```
; this is a comment
data: some text

data: another text
```

In the preceding code, the first line is just a comment. All the text messages starting with a colon character are comments. A comment could be useful as `Keep-Alive` if the messages are not sent regularly. The second and the third line contains just a data field with a text value. The third line contains an extra new line character terminating a message. Several events could be sent via a message. Each event has a name specified in the `event` field and `data` field whose values are any string. Data could also be in a JSON format as shown in the following code:

```
event: userLogon
data: {"username": "John", "time": "01:22:45", "text": "Hello World"}

event: userMessage
data: "Any data"
```

You can use unnamed events in messages without the `event` field. The SSE implementation uses `message` as a name for unnamed events. Each event might have an id. In this scenario, data can be combined as follows:

```
id: 123
data: some text
data: {"text": "Another text"}
```

Let's take a look at the server side of our example:

```
//...
main() {
    final allUrls = new RegExp('/(.*)');

    HttpServer.bind(urls.serverAddress, urls.serverPort)
    .then((server) {
        print("Server runs on ${server.address.host}:${server.port}");
        new Router(server)
            ..serve(urls.dataUrl, method: 'GET').listen(processSSE)
            ..serve(allUrls).listen(serveDirectory('', as: '/'))
            ..defaultStream.listen(send404);
    });
}

String fetchData() {
    return dataGenerator.nextInt(100).toString();
}
```

The `EventSource` class instance, which is created on the client side, opens a connection that submits the HTTP request with the `text/event-stream` value in the `accept` header. This is a signal for the server to start the SSE communication. In our example, we send the `logon` event first. The server connection remains open to the client so we can send message events periodically, as shown in the following code:

```
processSSE(HttpRequest request) {
    if (request.headers.value(HttpHeaders.ACCEPT) == EVENT_STREAM) {
        writeHead(request.response);
        int num = 0;

        new Timer.periodic(new Duration(seconds:5), (Timer t) {
            sendMessageEvent(request.response, num++);
        });

        sendLogonEvent(request.response);
    }
}
```

To allow you to send the push events, the output buffering in `HttpResponse` must be disabled. In addition to this, you need to specify content type, cache control, and the type of connection via the header attributes, as follows:

```
writeHead(HttpResponse response) {
    response.bufferOutput = false;
    response.headers.set(HttpHeaders.CONTENT_TYPE, EVENT_STREAM);
    response.headers.set(HttpHeaders.CACHE_CONTROL, 'no-cache');
    response.headers.set(HttpHeaders.CONNECTION, "keep-alive");
}
```

To send a message, you can use the `writeln` method of response. It automatically assigns to a newline character to each string, so you need to add only one newline character at the end of your event. Finally, the `flush` method of the response pushes the event to the client, as shown here:

```
sendMessageEvent(HttpResponse response, int num) {
    print("Send Message event $num");
    response.writeln('id: 123');
    response.writeln('data: {"msg": "hello world", "num": $num, "value": ${fetchData()}}\n');
    response.flush();
}
```

For the `logon` message, we create a custom-defined `userLogon` event type. If the connection opened via the `EventSource` terminates, the web browser will automatically re-establish the connection to the server after three seconds. You can change this value via the `retry` property of the event. This value must be an integer that specifies the reconnection time in milliseconds:

```
sendLogonEvent(HttpResponse response) {
    print("Send Logon event");
    response.writeln('event: userlogon');
    response.writeln('retry: 15000');
    response.writeln('id: 123');
    response.writeln('data: {"username": "John", "role": "admin"}\n');
    response.flush();
}
```

The SSE code implementation at the client side is as follows:

```
import 'dart:html';
import 'urls.dart' as urls;

var url = "http://${urls.serverAddress}:${urls.serverPort}${urls.
    dataUrl}";
String get timeStamp {
    DateTime dt = new DateTime.now();
    return " (${dt.minute}:${dt.second}) ";
}

responseHandler(DivElement log, String data) {
    DivElement item = new DivElement();
    item.text = data + timeStamp;
    log.insertAdjacentElement('beforebegin', item);
}
```

In the preceding code, we created an instance of `EventSource` with the specified URL. From now, we will start listening to the events from the server. We add the `open` and `error` event listeners. The `EventSource` class informs the code about the closed connection from the server side via changes in the `readyState` property. The message listener handles all the unknown events. A special event listener for the `userlogon` event handles the sort of events that could be added via the `addEventListener` method of the `EventSource` class. The event keeps the message information in the `data` property. An event identifier assigned on the server side is available via the `lastEventId` property, as shown in the following code:

```
main() {
   DivElement log = querySelector("#log");
   EventSource sse = new EventSource(url);
   sse.onOpen.listen((Event e) {
        responseHandler(log, "Connected to server: ${url}");
   });

    sse.onError.listen((Event e) {
        if (sse.readyState == EventSource.CLOSED) {
            responseHandler(log, "Connection closed");
        } else {
            responseHandler(log, "Error: ${e}");
        }
    });
    sse.onMessage.listen((MessageEvent e) {
        responseHandler(log,
            "Event ${e.lastEventId}: ${e.data}");
    });
}
```

```
});  
sse.addEventListener("userlogon", (Event e) {  
    responseHandler(log,  
        "User Logon: ${(e as MessageEvent).data}");  
, false);  
}  
}
```

Now, create a Dartium launcher, run the server, and open `index.html` in the web browser to see the following result:

```
Connected to server: http://localhost:8080/data (43:33)  
User Logon: {"username": "John", "role": "admin"} (43:33)  
Event 123: {"msg": "hello world", "num": 0, "value": 45} (43:38)  
Event 123: {"msg": "hello world", "num": 1, "value": 1} (43:43)  
Event 123: {"msg": "hello world", "num": 2, "value": 65} (43:48)  
Event 123: {"msg": "hello world", "num": 3, "value": 10} (43:53)  
//...
```

Let's take a look at the server log to see the server-generated messages:

```
Server runs on 127.0.0.1:8080  
Send Logon event  
Send Message event 0  
Send Message event 1  
Send Message event 2  
Send Message event 3  
//...
```

WebSocket

WebSocket is a bidirectional, message-oriented streaming transport between a client and a server. It is a built in TCP that uses an HTTP upgrade handshake. WebSocket is a message-based transport. Its simple and minimal API abstracts all the complexity and provides the following extra services for free:

- The same-origin policy enforcement
- Interoperability with the existing HTTP infrastructure
- Message-oriented communication
- Availability of subprotocol negotiation
- Low-cost extensibility

WebSocket is one of the most flexible transports available in the browser. The API enables the layer and delivers arbitrary application protocols between the client and server in a streaming fashion, and it can be initiated on either side at any time.



WebSocket is a set of multiple standards: the WebSocket API and the WebSocket protocol and its extensions.



WebSocket can be useful for the following cases:

- When the relevance of data is very critical
- When the solutions very often are based on high-volume data or data transmission

The advantages of WebSocket are as follows:

- It's a full-duplex, bidirectional communications channel that operates through a single socket
- It provides a quick file exchange based on the socket protocol
- It supports the binary format

The disadvantages of WebSocket are as follows:

- It is not HTTP and can be blocked by proxy servers
- Debugging is complicated
- It is supported only by the modern version of all browsers

The following browsers are supported:

- Chrome 14.0+
- Firefox 11.0+
- Opera 8.0+
- Safari 6.0+
- Internet Explorer 10.0+

Bear in mind that the WebSocket uses a custom WS protocol instead of HTTP. The use case for the WebSocket protocol was to provide an optimized, bidirectional communication channel between applications running in the browser and the server where using the HTTP protocol is obvious. WebSocket uses a custom URL schema because the WebSocket wire protocol can be used outside the browser and could be established via a non-HTTP exchange and the BiDirectional or Server-Initiated HTTP (HyBi) working group chooses to adopt a custom URL schema. Let's take a look at the client code of our example:

```
import 'dart:html';
import 'dart:convert';
import 'dart:typed_data';
import 'urls.dart' as urls;

var url = "ws://${urls.serverAddress}:${urls.serverPort}${urls.
  dataUrl}";

String get timeStamp {
    DateTime dt = new DateTime.now();
    return " (${dt.minute}:${dt.second})";
}

responseHandler(DivElement log, String data) {
    DivElement item = new DivElement();
    item.text = data + timeStamp;
    log.insertAdjacentElement('beforebegin', item);
}
```

First, we create a `WebSocket` class instance to initiate a new connection with the specified URL. Then, we create listeners for the `open`, `error`, `close`, and `message` events. As you can see, the API looks very similar to the `EventSource` API that we saw in the last topic. This is intentional because `WebSocket` offers a similar functionality and could help in the transition from the SSE solution quickly. When a connection is established, we can send data to the server. As `WebSocket` makes no assumption and no constraints on the application payload, we can send any data types such as `Map`, `String`, and `Typed`:

```
void main() {
    DivElement log = querySelector("#log");
    var webSocket = new WebSocket(url);
    if (webSocket != null) {
        webSocket.onOpen.listen((Event e) {
```

```
        responseHandler(log, "Connected to server: ${url}");
        sendData(webSocket);
    });
    webSocket.onError.listen((Event e)
        => responseHandler(log, "Error: ${e}"));
    webSocket.onClose.listen((CloseEvent e)
        => responseHandler(log, "Disconnected from server"));
    webSocket.onMessage.listen((MessageEvent e)
        => responseHandler(log, "Event ${e.type}: ${e.data}"));
}
}

sendData(WebSocket webSocket) {
    webSocket.send(JSON.encode({'name':'John', 'id':1234}));
    webSocket.sendString("Hello World");
    webSocket.sendTypedData(new Int16List.fromList([1, 2, 3]));
}
```

The browser automatically converts the received text-based data into a DOMString object and the binary data into a Blob object, and then passes them directly to the application. We can force an ArrayBuffer conversion when a binary message is received via the `binaryType` property of the `WebSocket` class:

```
webSocket.binaryType = "arraybuffer";
```

This is the perfect hint to the user agents on how to handle incoming binary data depending on the value in `binaryType`:

- If it is equal to `Blob`, then save it to the disk
- If it is equal to `arrayBuffer`, then keep it in the memory

The data in a `Blob` object is immutable and represents raw data. It could be the optimal format to keep the images downloaded from the server. We can pass that data directly to an image tag. On the other hand, the `arrayBuffer` object is likely a better fit for additional processing on binary data. In our example, we used a UTF-8 encoded text message, a UTF-8 encoded JSON payload, and the `arrayBuffer` object of the binary payload. All the `send` methods of the `WebSocket` class are asynchronous, but they are delivered in the exact order in which they are queued up by the client. As a result, a large number of messages in the queue will be delayed in delivery. To solve this problem, the application can split a large message into small chunks and monitor sending data via the `bufferingAmount` property of the `WebSocket` class as follows:

```
if (webSocket.bufferingAmount == 0) {
    ws.send(nextData);
}
```



The application that uses WebSocket should pay close attention to how and when to send messages in a queued socket.



Now let's take a look at the server side:

```
import 'dart:io';
import 'package:route/server.dart';

import 'urls.dart' as urls;
import 'files.dart';

main() {
    final allUrls = new RegExp('/(.*)');

    HttpServer.bind(urls.serverAddress, urls.serverPort)
        .then((server) {
            print("Server runs on ${server.address.host}:${server.port}");
            new Router(server)
                ..serve(urls.dataUrl).listen(processWS)
                ..serve(allUrls).listen(serveDirectory('', as: '/'))
                ..defaultStream.listen(send404);
        });
}
```

WebSocket upgrades the HTTP request to start listening to the data:

```
processWS(HttpRequest request) {
    WebSocketTransformer.upgrade(request)
        .then((WebSocket webSocket) {
            webSocket.listen((data) {
                webSocket.add("Received $data");
            });
        });
}
```

Finally, create a Dartium launcher, start the server, and open the `index.html` file in web browser to see the result of the short communication:

```
Connected to server: ws://127.0.0.1:8080/ws (8:55)
Event message: Received {"name":"John","id":1234} (8:55)
Event message: Received Hello World (8:55)
Event message: Received [1, 0, 2, 0, 3, 0] (8:55)
```

Summary

To summarize what has been discussed so far, I would like to highlight some of the important aspects of the client-to-server communication. Communications based on a system of special rules and formats for messages to enable data exchange between clients and servers is known as a communication protocol. Any device connected to the Internet has a unique IP address. A message transmitted from one device to other over the Internet follows a long route to be delivered via the protocol stack. HTTP is a text-based application protocol that makes the Web work. Web browsers or standalone applications send a request and open a connection to the web server. The web server complies with the request and closes the connection to the web client.

AJAX is a technology that allows client-side JavaScript code to request data from a server without reloading the current page in the web browser. `HttpRequest` (formerly `XMLHttpRequest`) has an important role in the AJAX web development technique. The AJAX polling request periodically issues a request to gain new information. A long polling request always keeps an open connection to the server. This connection is still alive until the server decides to submit the information back to the client where there are changes and then closes the connection. The client will again open a connection to the server to start a new long polling request.

In SSE, a client opens a connection to the server via an initial HTTP request, and the server sends events to the client when there is new information available. The response sent back via SSE is plain text served with a `text/event-stream` content type. It contains one or more lines of string that begins with a `data: string` and ends with a newline character. Finally, the whole message ends with an extra newline character.

WebSocket is a bidirectional, message-oriented streaming transport between a client and server. It is built on TCP that uses an HTTP upgrade handshake. The API enables the layer and delivers arbitrary application protocols between a client and server in a streaming fashion and is initiated on either side at any time.

In the next chapter, we will discuss the ability to store data locally on a client and break the storage limit of cookies in our web applications. We will also demonstrate how to use Web Storage and explore a more powerful and useful IndexedDB to store a large amount of data in the user's web browser.

10

Advanced Storage

In this chapter, we will talk about Dart's ability to store data locally on a client, break the storage limit, and prevent security issues in our web applications. We will also take a look at the good old cookies, show you how to use Web Storage, and elaborate on the more powerful and useful IndexedDB to store large amount of data in the user's web browser. The following topics will be covered in this chapter:

- Cookies
- Web Storage
- Web SQL
- IndexedDB

Cookies

The concept of **cookies** was introduced for the first time in 1994 in the Mosaic Netscape web browser. A year later, this concept was introduced in Internet Explorer. From that time, the idea of tracking contents of shopping cart baskets across browser sessions remains relevant until today. So, what problem do cookies solve?

An HTTP protocol is stateless and doesn't allow the server to remember the request later. Because it was not designed to be stateful, each request is distinct and individual. This simplifies contracts and minimizes the amount of data transferred between the client and the server. In order to have stateful communication between web browsers, you need to provide an area in their subdirectories where the state information can be stored and accessed. The area and the information stored in this area is called a cookie. Cookies contain the following data:

- A name-value pair with the actual data
- An expiry date after which the cookie is no longer valid
- The domain and path of the server it should be sent to

Cookies are handled transparently by the web browser. They are added to the header of each HTTP request if its URL matches the domain and the path is specified in the cookie. Cookies are accessible via the `document.cookie` property. Bear in mind that when you create a cookie, you specify the name, value, expiry date, and optionally the path and domain of cookies in a special text format. You can find a cookie by its name and fetch the value only, as all the other information that is specified while creating a cookie is used by the web browser and is not available to you. We created the `cookies.dart` file to keep small library wrappers of the original API of the cookie to avoid boilerplate and code duplication, which allow you to easily set, get, and remove cookies from any application. The cookies use the date format derived from the RFC 1123 International format, that is, Wdy, DD Mon YYYY HH:MM:SS GMT. Let's look at them in more detail:

- **Wdy:** This is a three-letter abbreviation that represents the day of the week, and is followed by a comma
- **DD:** These are two numbers that represent the day of the month
- **Mon:** These are three letters that represent the name of the month
- **YYYY:** These are four numbers that represent the year
- **HH:** These are two numbers that represent the hour
- **MM:** These are two numbers that represent the minutes
- **SS:** These are two numbers that represent the seconds

You can see the date in this example: Thu, 09 Oct 2014 17:16:29 GMT.

Let's go through the following code and see how cookies can be reached by Dart. We import the `dart:html` package especially to make `document.cookies` available in our code. The `intl.dart` package is imported because of `DateFormat` usage in the `toUTCString` function to calculate the expiry date in the UTC format of the cookie based on the value in the `days` attribute and transforms it into a string. If the value of `days` is less than one, then the `toUTCString` function returns an empty string. To create cookies with the `setCookie` function, we need to specify the name of the cookie, value, and the number of days to expire. We can provide the optional path and domain information as well. In practice, you cannot use non-ASCII characters in cookies at all. To use Unicode, control codes, or other arbitrary byte sequences, you must use the most popular UTF-8-inside-URL-encoding that is produced by different encoding methods of the `Uri` class. To return cookies via `getCookie`, we only need the name of the cookie. At the end, you will find the `removeCookie` function.

```
library cookie;

import 'dart:html';
import 'package:intl/intl.dart';
```

```
// Number milliseconds in one day
var theDay = 24*60*60*1000;

DateFormat get cookieFormat =>
    new DateFormat("EEE, dd MMM yyyy HH:mm:ss");

String toUTCString(int days) {
    if (days > 0) {
        var date = new DateTime.now();
        date = new DateTime.fromMillisecondsSinceEpoch(
            date.millisecondsSinceEpoch + days*24*60*60*1000);
        return " expires=${cookieFormat.format(date)} GMT";
    } else {

        return " ";
    }
}

void setCookie(String name, String value, int days,
               {String path: '/', String domain:null}) {
    StringBuffer sb = new StringBuffer(
        "${Uri.encodeQueryComponent(name)}=" +
        "${Uri.encodeQueryComponent(value)};");
    sb.write(toUTCString(days));
    sb.write("path=${Uri.encodeFull(path)}; ");
    if (domain != null) {
        sb.write("domain=${Uri.encodeFull(domain)}; ");
    }
    document.cookie = sb.toString();
}

String getCookie(String name) {
    var cName = name + "=";
    document.cookie.split("; ").forEach((String cookie) {
        if (cookie.startsWith(cName)) {
            return cookie.substring(cName.length);
        }
    });
    return null;
}

void removeCookie(String name) {
    setCookie(name, '', -1);
}
```

The domain of the cookie tells the browser the domain where the cookie should be sent, and in its absence, the domain of the cookie becomes the domain of the page. It allows cookies to cross subdomains, but it does not allow the cookies to cross domains. The path of the domain is the directory present in the root of the domain where the cookie is active.

For a better understanding of the use of cookies in the real world, you can look at the `shopping_cart_cookie` project. This project is very big, so I will show you small code snippets and point you in the right direction. This project contains the following main classes:

- `Product`: This class describes the items in a cart with the ID, description, and price. Users can specify the quantity of items they want to buy, so the amount can be calculated by multiplying the quantity with the price.
- `ShoppingModel`: This class helps you to fetch products from the `product.json` file and returns the `Future` instance with a list of products.
- `ShoppingController`: This class renders the grid of products and updates the amount per product and the total amount. We can send a reference in the body of the table. This reference on the element keeps the total amount and the instance of the `ShoppingModel` and `StorageManager` class via a constructor injection.

We will call `getProducts` from the model in the constructor especially to return data from the server and when the data is ready, we will call the `_init` method to initialize our application. The `readQuantity` method is called for the first time to check and return the quantity of the product saved in the cookie. Later in the code, we will call `calculateAmount` of the product based on the quantity and price, as shown in the following code:

```
ShoppingController(this.tBody, this.totalAmount, this.service,
    this.storage) {
    model.getProducts().then((List<Product> products) {
        _products = products;
        _init();
    });
}

_init() {
    update().then((value) {
        draw();
        drawTotal();
    });
}
```

```
Future update() {
    return Future.forEach(_products, (Product product) {
        // Read quantity of product from cookie
        return readQuantity(product);
    });
}

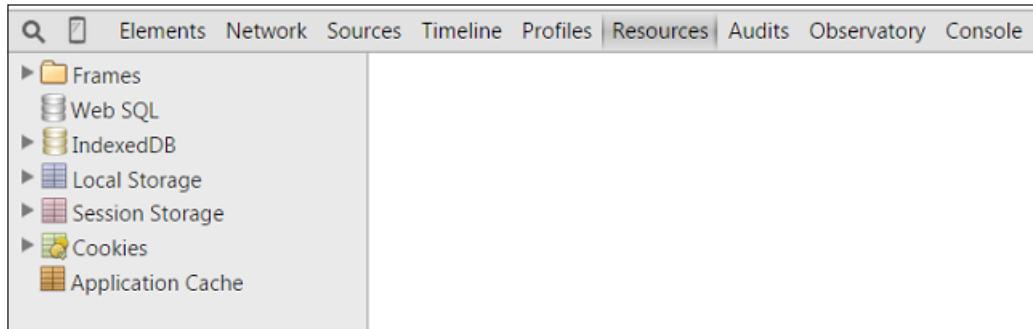
Future readQuantity(Product product) {
    return storage.getItem(Product.toCookieName(product))
        .then((String quantity) {
            if (quantity != null && quantity.length > 0) {
                product.quantity = int.parse(quantity);
            } else {
                product.quantity = 0;
            }
        });
}
//...
```

The `Product.toCookieName` method creates a unique string identifier with a combination of `Product.NAME` and `product.id`. Now launch the application and open the `index.html` file on the Dartium web browser to see a list of products.

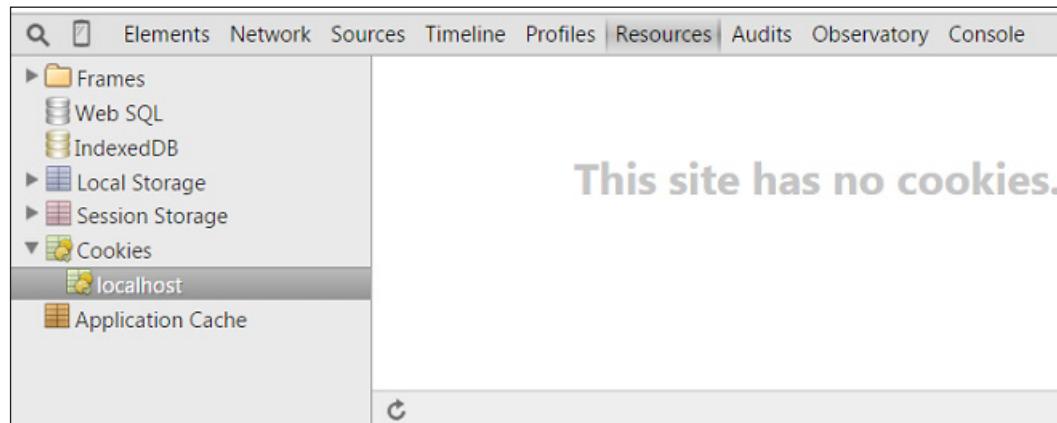
Shopping Cart				
Product	Quantity	Price	Amount	
15 inch display, Desktop Mount, Black	<input type="text" value="0"/>	150.0	0.0	
17 inch display, Wall mount, Black	<input type="text" value="0"/>	250.0	0.0	
19 inch display, Desktop Mount, White	<input type="text" value="0"/>	340.0	0.0	
24 inch display, Wall mount, Silver	<input type="text" value="0"/>	470.0	0.0	
Total				0.0
<input type="button" value="Check Out"/>				

Advanced Storage

We can check the existence of cookies in the web browser. In Dartium, open **Developer tools** from the **Tools** menu item and choose the **Resources** tab, as shown in the following screenshot:



Select **Cookies** and choose **localhost** to ensure that no cookies are associated with our web server.



If the user changes the quantity of any product via the text input field, then the number of products and the total number of selected products will be recalculated. At the same time, our code calls the `saveQuantity` method of the `ShoppingController` class, as follows:

```
saveQuantity(Product product) {
    if (product.quantity == 0) {
        storage.removeItem(Product.toCookieName(product));
    } else {
        storage.setItem(Product.toCookieName(product),
            product.quantity.toString());
    }
}
```

The product is removed from the cookie if the number in the **Quantity** field equals zero. In other cases, we will create or update the quantity value in cookies.

Shopping Cart				
Product	Quantity	Price	Amount	
15 inch display, Desktop Mount, Black	1	150.0	150.0	
17 inch display, Wall mount, Black	2	250.0	500.0	
19 inch display, Desktop Mount, White	0	340.0	0.0	
24 inch display, Wall mount, Silver	0	470.0	0.0	
	Total		650.0	
	Check Out			

Let's check the preceding information. Return to the **Resources** tab and navigate to the **localhost** tree item. Click on the **Refresh** icon at the bottom of the window to see the list of cookies associated with our server.

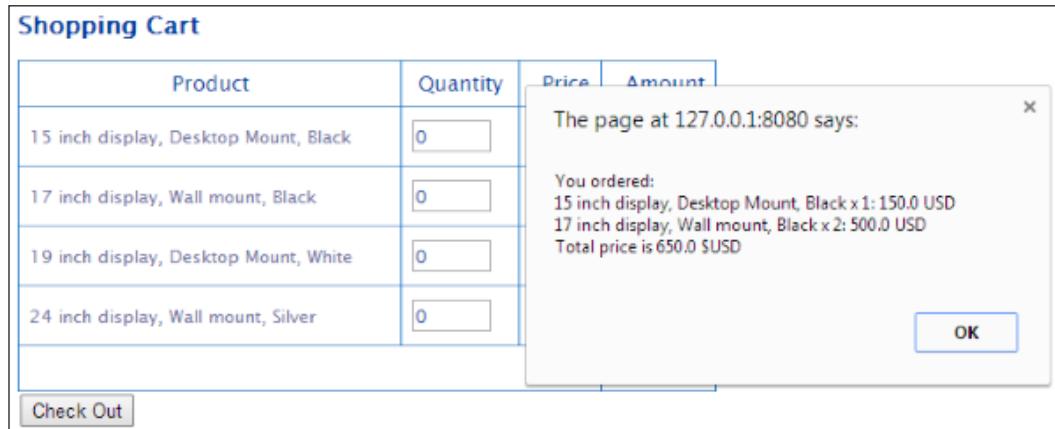
The screenshot shows the Chrome DevTools Resources tab with the 'Cookies' section selected. It lists two cookies: 'PRODUCT_02-0902' and 'PRODUCT_02-0903'. Both cookies have a value of 1 and 2 respectively, and they both expire on 2015-10-13T05:13:32.000Z.

Name	Value	Domain	Path	Expires / Max-Age	Size	HTTP	Secure
PRODUCT_02-0902	1	localhost	/	2015-10-13T05:13:32.000Z	16		
PRODUCT_02-0903	2	localhost	/	2015-10-13T05:17:24.000Z	16		

Now, close the `index.html` page and open it again. Information about the selected products along with their specified quantity and number will be available here. Click on the **Check Out** button to invoke a `cart.checkOut` method to show a message about the paid items and remove the cookies from them. The code is as follows:

```
checkOut.onClick.listen((Event event) {
  cart.checkOut();
});
```

The following screenshot shows the resulting message:



Cookies are a universal mechanism that help to persist user-specific data in the web browser. They can be very useful while essaying the following roles:

- Tracking a cookie helps compile long-term records of the browsing history.
- The authentication cookie used by the web server helps you know whether the user was logged in or not.
- Session cookies exist until the user navigates to a website and expires after a specified interval, or if the user logs in again via the server logic.
- Persistent cookies live longer than session cookies and may be used to keep vital information about the user, such as the name, surname, and so on.
- Secure cookies are only used via HTTPS with a secure attribute enabled to ensure that the cookie transmission between the client and the server is always secure.
- Third-party cookies are the opposite of first-party cookies, and they don't belong to the same domain that is displayed in the web browser. Different content presented on web pages from third-party domains can contain scripts that help track users' browser history for effective advertising.

You can combine different roles to achieve specific requirements of your business logic. The following are the benefits of having cookies:

- **Convenience:** Cookies can remember every website you have been to and also the information in forms such as residential or business address, e-mail address, username, and so on

- **Personalization:** Cookies can store preferences, which helps every user to personalize the website content
- **Effective advertising:** Cookies can help run marketing campaigns to offer products or services relevant to a specific user
- **Ease of control:** Cookies can be cleared or disabled via the client's web browser

The following are the disadvantages of cookies:

- **Privacy:** Cookies keep a track of all the websites that you have visited
- **Security:** Implementation of cookies on different web browsers is accompanied by the detection of various security holes
- **Limitation:** Cookies have a limit of 4095 bytes per cookie
- **Data:** Cookies can overhead each request with excessive extra data

So, the main purpose of cookies is to make the client-to-server communication stateful. If you need to only save data on a client or work offline, you can use other techniques and one of them is Web Storage.

Web Storage

Web Storage (or DOM storage) represents a mechanism for a better and more secured way of persisting data on the client than cookies. Web Storage is better in the following situations:

- When you need greater storage capacity (it can keep 5 - 10 MB per the available storage, depending on the web browser)
- When you don't need to communicate with the server to manage the client data
- When you don't want the stored data to expire
- When the stored data spans across different tabs or windows of the same browser

There are two Web Storage objects (`Session` and `Local`) that can be used to persist the user data in the web browser for the length of the session and indefinitely. Both of them have a similar simple API declared via the `Storage` interface. Web Storage has an event-driven implementation. The `storage` event is fired whenever a storage area changes.

The Session storage

The **Session storage** is available through the `window.sessionStorage` attribute. It is intended to keep short-lived data opened in the same tab or window and shared only between the pages of the same domain. The browser creates a new session storage database every time a user opens a new tab or a window.

The Local storage

As opposed to the Session storage, the **Local storage** is available via the `window.localStorage` attribute and allows you to keep data longer than a single session. The Local storage saves all data in a special local storage area and is not limited to the lifetime of the tab or a window. The local storage area is shared between different tabs and windows and can be very handy in multithreaded scenarios.

Let's see how we can change the examples from the previous *Cookies* section to use them in Web Storage. We will not delete the cookie completely so that we have chance to compare different persisting techniques. Open the `shopping_cart_web_storage` project. In the following code, we add the business logic to check whether Web Storage is supported by the web browser with the `StorageManager` class:

```
abstract class StorageManager {  
    factory StorageManager() {  
        if (WebStorageManager.supported) {  
            return new WebStorageManager();  
        } else {  
            return new CookieStorageManager();  
        }  
    }  
  
    Future<String> getItem(key);  
    Future setItem(key, value);  
    Future removeItem(key);  
}
```

In the preceding code, you can see the new `WebStorageManager` class. It is quite difficult to determine whether the web browser supports Web Storage or not. You may find one of the possible solutions in the following code that uses the `supported` getter method. The `getItem` function returns the value associated with the given key. The `setItem` function sets a key-value pair in this method and the `last` method removes the item with the specified key. The code is as follows:

```
class WebStorageManager implements StorageManager {  
    static bool get supported {  
        if (window.localStorage != null) {  
            try{  
                window.localStorage["__name__"] = "__test__";  
            } catch(e){  
                return false;  
            }  
        }  
        return true;  
    }  
  
    Future<String> getItem(key);  
    Future setItem(key, value);  
    Future removeItem(key);  
}
```

```

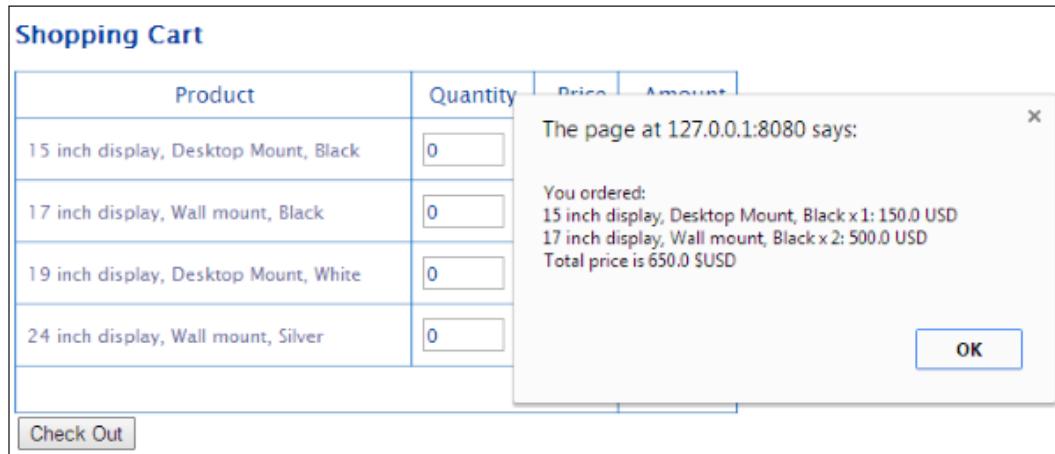
        window.localStorage.remove ("__name__");
        return true;
    } catch(e) {
        return false;
    }
} else {
    return false;
}
}

Future<String> getItem(key) {
    return new Future.sync(() {
        return window.localStorage[key];
    });
}

Future setItem(key, value) {
    return new Future.sync(() {
        window.localStorage[key] = value;
    });
}
Future removeItem(key) {
    return new Future.sync(() {
        window.localStorage.remove(key);
    });
}
}

```

The following screenshot shows the result of executing the program:



You can find information about the status of the local storage in the same place where we saw the cookies. Expand the **LocalStorage** tree item and choose the **localhost** option to see the local storage data:

The screenshot shows the Chrome DevTools Resources panel. The left sidebar lists 'Frames', 'Web SQL', 'IndexedDB', 'LocalStorage' (which is expanded), 'Session Storage', 'Cookies', and 'Application Cache'. Under 'LocalStorage', there is an entry for 'http://localhost:8081'. The main pane has two columns: 'Key' and 'Value'. There are two entries: 'PRODUCT_02-0902' with value '1' and 'PRODUCT_02-0903' with value '2'.

Key	Value
PRODUCT_02-0902	1
PRODUCT_02-0903	2

Now, we can use Web Storage and cookies to store a date for the client. We will continue our trip across advanced storages and the next target is Web SQL.

Web SQL

Web SQL was introduced by Apple and is based on a free database SQLite. It will be discontinued very soon, and I intend to add it here only to show you how it could be used and how we can migrate to other advanced techniques without glitches. In the following code, we created a new `shopping_cart_web_sql` project as a copy of the project from the *Web Storage* section and added `WebSQLStorageManager` into the `StorageManager` class:

```
abstract class StorageManager {  
    factory StorageManager() {  
        if (WebSQLStorageManager.supported) {  
            return new WebSQLStorageManager();  
        } else if (WebStorageManager.supported) {  
            return new WebStorageManager();  
        } else {  
            return new CookieStorageManager();  
        }  
    }  
  
    Future<String> getItem(key);  
    Future setItem(key, value);  
    Future removeItem(key);  
}
```

First of all, check whether the web browser supports Web SQL and instantiate it if successful. You should specify the version and initial size of the database to be used. The web browser support for Web SQL can be quickly checked with the supported property `SqlDatabase` class. Web SQL needs a big preparation before it can be used. First of all, we need to open the database. After the database is open, we can create a table if it does not exist. Web SQL has a more complex API than a cookie and Web Storage; each method increases in size exponentially. It is vital that all methods must execute very specific SQL statements. WebSQL also supports read and write transactions. When a transaction begins, you need to specify the key that will be assigned to the SQL parameters. The instance of `SqlResultSet` keeps a track of the transactions. The `rows.isEmpty` property of `SqlResultSet` is an important property that tells us exactly how many rows were returned, as shown in the following code:

```
class WebSQLStorageManager implements StorageManager {

    static const SHOPPING = "SHOPPING";
    static const PRODUCT = "PRODUCT";
    static const TRANS_MODE = "readwrite";
    static final String VERSION = "1";
    static const int SIZE = 1048576;

    SqlDatabase _database;
    static bool get supported => SqlDatabase.supported;
    Future<SqlDatabase> _getDatabase(
        String dbName, String storeName) {
        if (_database == null) {
            _database = window.openDatabase(dbName, VERSION,
                dbName, SIZE);
            var sql = 'CREATE TABLE IF NOT EXISTS ' +
                storeName +
                ' (id NVARCHAR(32) UNIQUE PRIMARY KEY, value TEXT)';
            var completer = new Completer();
            _database.transaction((SqlTransaction tx) {
                tx.executeSql(sql, [],
                    (SqlTransaction txn, SqlResultSet result) {
                        completer.complete(_database);
                    },
                    (SqlTransaction transaction, SqlError error) {
                        completer.completeError(error);
                    });
            }, (error) => completer.completeError(error));
            return completer.future;
        } else {
            return new Future.sync(() => _database);
        }
    }
}
```

```
        }

Future<String> getItem(key) {
    var sql = 'SELECT value FROM $PRODUCT WHERE id = ?';
    var completer = new Completer();
    _getDatabase(SHOPPING, PRODUCT).then((SqlDatabase database) {
        database.readTransaction((SqlTransaction tx) {
            tx.executeSql(sql, [key],
                (SqlTransaction txn, SqlResultSet result) {
                    if (result.rows.isEmpty) {
                        return completer.complete(null);
                    } else {
                        Map row = result.rows.first;
                        return completer.complete(row['value']);
                    }
                },
                (SqlTransaction transaction, SqlError error) {
                    completer.completeError(error);
                });
        });
    });
    return completer.future;
}
```

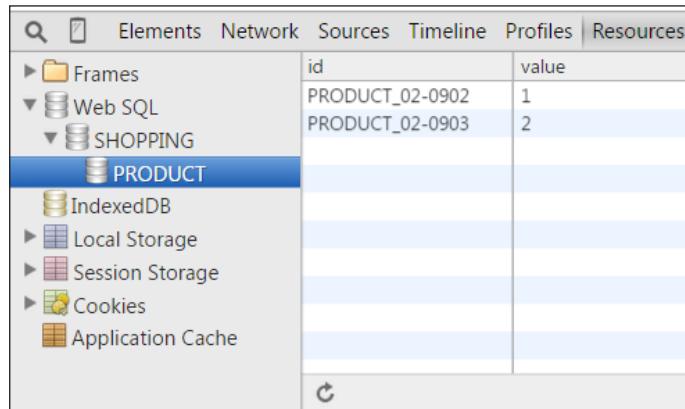
For the create and update operations, we use the following transaction and specify the special SQL statement, and we need to specify the SQL parameters key and value as well:

```
Future setItem(key, value) {
    var sql = 'INSERT OR REPLACE INTO $PRODUCT (id, value) ' +
        'VALUES (?, ?)';
    var completer = new Completer();
    _getDatabase(SHOPPING, PRODUCT).then((SqlDatabase database) {
        database.transaction((SqlTransaction tx) {
            tx.executeSql(sql, [key, value],
                (SqlTransaction txn, SqlResultSet result) {
                    return completer.complete(value);
                },
                (SqlTransaction transaction, SqlError error) {
                    completer.completeError(error);
                });
            }, (error) => completer.completeError(error));
        });
    return completer.future;
}
```

To remove an item, we use the following transaction and specify the `key` parameter that will be assigned to the SQL parameters:

```
Future removeItem(key) {
    var sql = 'DELETE FROM $PRODUCT WHERE id = ?';
    var completer = new Completer();
    _getDatabase(SHOPPING, PRODUCT).then((SqlDatabase database) {
        database.transaction((SqlTransaction tx) {
            tx.executeSql(sql, [key],
                (SqlTransaction txn, SqlResultSet result) {
                    return completer.complete();
                },
                (SqlTransaction transaction, SqlError error) {
                    completer.completeError(error);
                });
            }, (error) => completer.completeError(error));
        });
        return completer.future;
    });
}
```

Launch our application and change the number of products. Expand the **Web SQL** tree item from the **Resources** tab to see the Web SQL storage data. The following screenshot shows the **SHOPPING** database and the **PRODUCT** table with the stored data:



The screenshot shows the Chrome DevTools Resources panel. The left sidebar lists various storage types: Frames, Web SQL, Local Storage, Session Storage, Cookies, and Application Cache. Under Web SQL, the SHOPPING database is expanded, showing the PRODUCT table. The table has two rows with columns 'id' and 'value'. The first row has id 'PRODUCT_02-0902' and value '1'. The second row has id 'PRODUCT_02-0903' and value '2'.

	id	value
	PRODUCT_02-0902	1
	PRODUCT_02-0903	2

The Web SQL database API isn't actually a part of the HTML5 specification. Therefore, it's time to migrate to IndexedDB if you have a code that uses Web SQL.

IndexedDB

IndexedDB was introduced by Oracle and became popular very quickly. It's a Not Only SQL (NoSQL) database. The IndexedDB API is a more capable and far more complex API. IndexedDB has the following significant benefits:

- It improves the responsiveness and speed of web programs by minimizing the number of HTTP requests
- It provides more space for data without Web Storage limits
- It provides the ability to work offline
- A NoSQL database helps you work directly with Dart and JavaScript objects
- It allows fast indexing, object searching, and granular locking per transaction
- It supports synchronous and asynchronous APIs

One of the major disadvantages can be the difficulty in understanding it if you are coming from the world of rational databases. In IndexedDB, we can store a large amount of structured data, images, arrays, and whole objects; you just need to index them with a key. It follows the same origin policy, so we cannot access data across different domains. If you still use Web SQL database with your products, it's time to migrate to IndexedDB because the Web SQL database was deprecated by **World Wide Web Consortium (W3C)** in November 2010.



IndexedDB is an indexed table system.



IndexedDB doesn't have any limits on a single database item's size, but it may impose a limit on each database's total items. We will use the asynchronous API because it works in most scenarios, including Web Workers. IndexedDB is a real database; therefore, before we use it, we need to specify a database schema, open a connection, and start using it to retrieve and update data within transactions. In this case, it gets very close to the Web SQL solution but is much simpler. Let's take a look at how we can use our example with IndexedDB. We will use the `dart:indexed_db` package from the Dart SDK to work with IndexedDB. You can make a copy of the project from the *Web SQL* section, rename it `shopping_cart_indexed_db`, and use `IndexedDBStorageManager` instead of `websQLStorageManager`. The code is as follows:

```
abstract class StorageManager {  
    factory StorageManager() {  
        if (IndexedDBStorageManager.supported) {  
            return new IndexedDBStorageManager();  
        }  
    }  
}
```

```
    } else if (WebStorageManager.supported) {
        return new WebStorageManager();
    } else {
        return new CookieStorageManager();
    }
}

Future<String> getItem(key);
Future setItem(key, value);
Future removeItem(key);
}
```

The preceding code shows the `IndexedDBStorageManager` class. We constructed a special `_getDatabase` method to retrieve an instance of the `Database` class. As we mentioned earlier, before we use `IndexedDB`, we need to open the `IndexedDB` database. In the following code, we use the `window.indexedDB.open` method to open our database. Next, we need to check whether the store exists in an `objectStoreNames` array of the database. If it doesn't exist, we must close the database and open it again with a higher version number. Because this process is asynchronous, we create a new instance of the `store` object inside the `onUpgradeNeeded` listener. Each manipulation of the objects of the database happens inside a transaction. So, we will create a new transaction every time and return the `ObjectStore` instance in the `startTransaction` method. We will return the value of `ObjectStore` via the `getObject` method. To set an item in the database, we use the `put` method of `ObjectStore`. To remove the object from the store, just call the `delete` method, as shown in the following code:

```
class IndexedDBStorageManager implements StorageManager {

    Database _database;

    static const SHOPPING = "SHOPPING";
    static const PRODUCT = "PRODUCT";
    static const TRANS_MODE = "readwrite";
    Future _getDatabase(String dbName, String storeName) {
        if (_database == null) {

            return window.indexedDB.open(dbName).then((Database d) {
                _database = d;
                if (!_database.objectStoreNames.contains(storeName)) {
                    _database.close();
                    return window.indexedDB.open(dbName,
                        version: (d.version as int) + 1,
                        onUpgradeNeeded: (e) {
```

```
        Database d = e.target.result;
        d.createObjectStore(storeName);
    }).then((Database d) {
        _database = d;
        return _database;
    });
}
return _database;
});
} else {
    return new Future.sync(() => _database);
}
}
Future
<ObjectStore> startTransaction(String storeName) {
    return _getDatabase(SHOPPING, PRODUCT)
    .then((Database database) {
        Transaction transaction =
            _database.transactionStore(storeName, TRANS_MODE);
        return transaction.objectStore(storeName);
    });
}
Future
<String> getItem(key) {
    return new Future.sync(() {
        return startTransaction(PRODUCT).then((ObjectStore store) {
            return store.getObject(key);
        });
    });
}
Future
setItem(key, value) {
    return new Future.sync(() {
        return startTransaction(PRODUCT).then((ObjectStore store) {
            return store.put(value, key);
        });
    });
}
Future
removeItem(key) {
    return new Future.sync(() {
        return startTransaction(PRODUCT).then((ObjectStore store) {
            return store.delete(key);
        });
    });
}
}
```

That's it. We can perform all manipulations on the data within `ObjectStore`. The fact that the `ObjectStore` class instance was returned via a `Transaction` class indicates that all steps in the original one will be surrounded by a transaction. Let's expand the **IndexedDB** tree item in the **Resources** tab to see the **SHOPPING** database.

The screenshot shows the Chrome DevTools Resources tab. In the left sidebar, under the 'IndexedDB' section, the 'SHOPPING - http://localhost:8081' database is selected. On the right, the database details are shown: Security origin: `http://localhost:8081`, Name: `SHOPPING`, Integer Version: `2`, and String Version: `null`. The 'PRODUCT' object store is expanded, showing two entries:

#	Key	Value
0	<code>"PRODUCT_02-0902"</code>	<code>"1"</code>
1	<code>"PRODUCT_02-0903"</code>	<code>"2"</code>

The version number of the database is **2**. Now, choose the **PRODUCT** tree item and you will see that the same name contains our data.

The screenshot shows the Chrome DevTools Resources tab. In the left sidebar, under the 'IndexedDB' section, the 'SHOPPING - http://localhost:8081' database is selected. On the right, the database details are shown: Security origin: `http://localhost:8081`, Name: `SHOPPING`, Integer Version: `2`, and String Version: `null`. The 'PRODUCT' object store is expanded, showing two entries:

#	Key	Value
0	<code>"PRODUCT_02-0902"</code>	<code>"1"</code>
1	<code>"PRODUCT_02-0903"</code>	<code>"2"</code>

When compared to Web SQL, IndexedDB is more simple and flexible. It uses indexes to access objects in the database within transactions.

Summary

To summarize, we will highlight some important facts about advanced storage space.

The concept of cookies was introduced for the first time on the Netscape web browser and was later migrated to Internet Explorer. It still remains relevant today. In order to have stateful communication, web browsers provide an area in their subdirectories where state information can be stored and accessed. The area and information stored in this area is called a cookie. The domain of cookies tells the browser the domain where the cookie should be sent. It allows a cookie to cross subdomains but does not allow it to cross domains. The path is the directory in the root of the domain where the cookie is active. Cookies are the universal mechanism that helps persist user-specific data in web browsers.

Web Storage represents a mechanism for better and a more secured way to persist data on the client than cookies. There are two Web Storage objects (Session and Local) that can be used to persisting user data in a web browser for the length of a session and indefinitely. Both of them have a similar simple API declared via a Storage interface.

Web SQL was introduced by Apple and is based on a free database SQLite. It will be discontinued very soon, and we used it here only to see how it could be used and how migration to other advanced technologies can be done painlessly.

In IndexedDB, we can store large amount of structured data, images, arrays, and whole objects, and just index them with a key. IndexedDB doesn't have any limits on a single database item's size, but it might impose a limit on each database's total items.

In the next chapter, we will demonstrate how different HTML5 features can be used in Dart.

11

Supporting Other HTML5 Features

HTML5 was designed to deliver rich, cross-platform content without the need to use additional plugins. In this chapter, we will learn how different HTML5 features can be used in Dart. We will cover the following topics:

- The notification APIs
- The native drag-and-drop APIs
- The geolocation APIs
- Canvas

The notification APIs

Processes that occur in web applications are asynchronous, and as time passes, they generate event messages to alert end users about the start, end, or progress of the process execution. The **web notification** API allows you to display notifications outside the context of web pages. The user agent determines the optimum presentation of the notification. This aspect depends on the device on which it is running. Usually, it can present notifications in the following areas:

- At a corner of the display
- In an area within the user agent
- On the home screen of a mobile device

The web notification API, available as part of the `dart:html` package, allows you to send information to a user even if the application is idle. Notifications can be useful in the following use cases:

- To notify users about new incoming messages
- To notify users about special events in game applications
- To notify users about the connection status of an application, the battery status of a device, and so on

When to notify

When you build web applications, you can use the notification API in the event handlers or polling functions to notify the users. Event handlers are the obvious choice when it comes to responding to various happenings. Event handlers use simple, required conditions that can detect events from the DOM elements and send a notification event to the user. Sometimes, the conditions required can be a lot more complex and event handlers may not be suitable to cover them. In such cases, you can use a polling function (implemented as a combination of event handlers) to periodically check for given conditions to send notifications. Notifications can be of the following two types:

- **DOM notifications:** These come from within a web application and are very useful when detecting manipulations with the DOM structure or properties
- **System notifications:** These come from an external source (outside the application) and are used to notify users about the status of a program or system

Let's see how we can use the web notification API for our needs. You can find the source code in the notification project. In the following code, we used the standard button event handler to send notifications to the user:

```
void main() {  
    var notifyBtn = querySelector("#notify_btn")  
    ..onClick.listen((Event event) {  
        sendNotification();  
    });  
}  
  
void sendNotification() {  
    Notification.requestPermission().then((String permission) {  
        if (permission == "granted") {
```

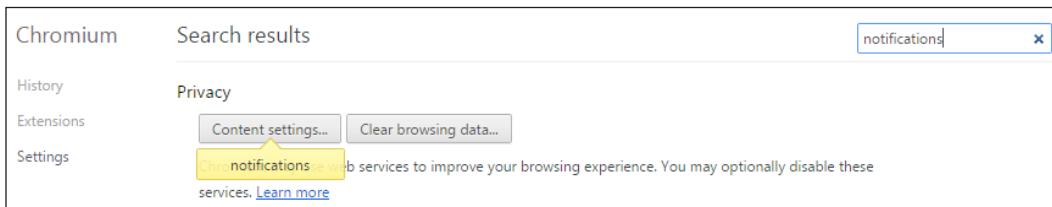
```

        Notification notification =
            new Notification('My Notification', body: "Hello World");
    }
})
);
}

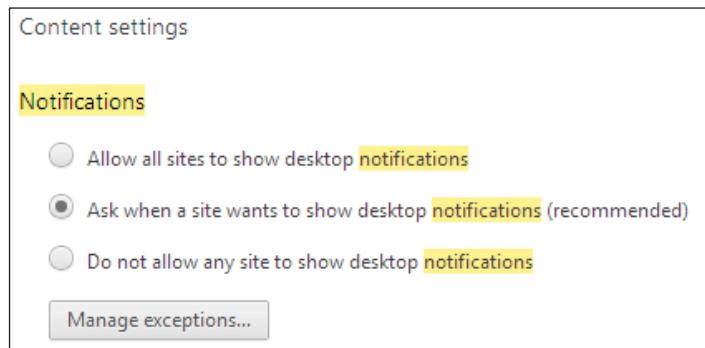
```

Before you send any notifications to the user, the website must have permissions. You can let websites send a notification automatically or by means of a permission request first. This is a common requirement when an API tries to interact with something outside a web context, especially to avoid sending useless notifications to the user. To see the notification settings, follow these steps:

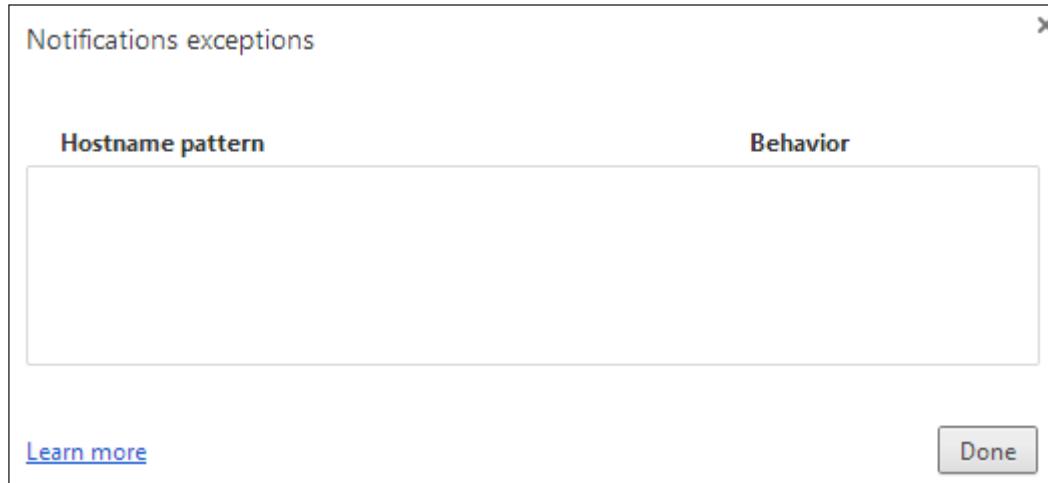
1. Open the **Settings** option in Dartium and type **notifications** in the search field, as shown in the following screenshot:



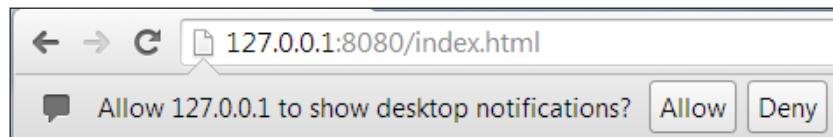
2. Open the **Content** settings pop-up dialog by clicking on the button of the same name, and then scroll down to find the **Notification** settings:



3. Choose the recommended option and open the **Notifications exceptions** dialog by clicking on the **Manage exceptions** button. For now, it will not contain our website, as shown in the following screenshot:



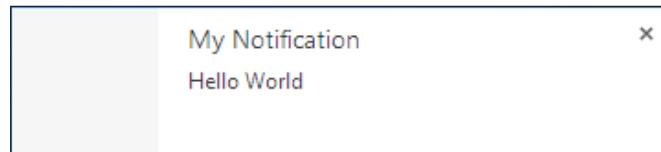
4. Now, run the application and click on the **Notify** button. As shown in the following screenshot, the `requestPermission` static method of the **Notification** class requests a permission to allow desktop notifications for the localhost:



5. You can allow or deny notifications for this website. Your choice will complete the future permission requests with the value of the chosen permission. The desktop notifications that are allowed are added to the list of **Notifications exceptions**. Now, open the **Notifications exceptions** dialog again to see your website, as shown in the following screenshot:



6. The next step is to create a notification. It is enough to specify only the title of the notification to create an original one. The constructor of the `Notification` class has optional properties that help us create notifications with a body and icon. A notification when instantiated is displayed as soon as possible, as shown in the following screenshot:



A notification triggers the following four events that track the current state of the notification:

- `show`: This event is triggered when the notification is displayed
- `click`: This event is triggered when the user clicks on the notification
- `close`: This event is triggered when the notification is closed
- `error`: This event is triggered when something goes wrong while displaying notifications

Notifications are still open until the user closes them, but you can use the `close` method of the `Notification` class to close them via a program, as shown in the following code:

```
void sendNotification() {
    Notification.requestPermission().then((String permission) {
        if (permission == "granted") {
            Notification notification =
                new Notification('My Notification', body: "Hello World");
            notification.onShow.listen((Event event) {
                new Timer(new Duration(seconds:2), () {
                    notification.close();
                });
            });
        }
    });
}
```

Preventing repeated notifications

Every time you click on the **Notify** button in your application, the web notification API generates new notifications and puts them on top of the previous one. Similar notifications can be marked with the following `tag` attribute to prevent crowding a user's desktop with hundreds of analogous notifications:

```
void sendNotification() {
    Notification.requestPermission().then((String permission) {
        if (permission == "granted") {
            Notification notification =
                new Notification('My Notification', body: "Hello World",
                    tag: "myNotification");
            notification.onShow.listen((Event event) {
                new Timer(new Duration(seconds:2), () {
                    notification.close();
                });
            });
        }
    });
}
```

Now when you generate a notification with the same tag, the web notification API removes the previous one and adds the new one instead. Web notification specifications are not stable yet, and they are supported only by the latest version of web browsers such as Chrome, Firefox, Safari, and Opera.

The native drag-and-drop APIs

Drag-and-drop is a way to convert the pointing device's movements and clicks to special commands that are recognized by software to provide quick access to common functions of a program. The user grabs a virtual object, drags it to a different location or another virtual object, and drops it there. Drag-and-drop support for a native browser means faster, more responsive, and more intuitive web applications. Before you use the drag-and-drop feature, make sure you have draggable content.

Draggable content

An abstract `Element` class has a `draggable` attribute that indicates whether the element can be dragged and dropped. As all the DOM elements emerge from the `Element` abstract class, this means all of them support the drag-and-drop operation by default. To make elements draggable, we need to set their `draggable` attribute to `true`. This can be done using the following code:

```
var dragSource = querySelector("#sample_drag_id");
dnd.draggable = true;
```

Alternatively, you can do this using the following HTML markup:

```
<p id="sample_dnd_id" draggable="true">Drag me!</p>
```

If you want to prevent the text contents of draggable elements from being selected, you can style the element, as shown in the following code:

```
[draggable] {
  -moz-user-select: none;
  -khtml-user-select: none;
  -webkit-user-select: none;
  user-select: none;
  -khtml-user-drag: element;
  -webkit-user-drag: element;
}
```

Let's open the `drag_and_drop` project and run it. In the following screenshot, you will see that you can drag the text element within the window of the browser but cannot drop it:



To manage the drag-and-drop operations in the example, add the drag-and-drop event listeners described in the next section.



During a drag operation, the native drag-and-drop API is fired only by the drag events and not the mouse events.



The drag-and-drop events

The native drag-and-drop API fires the following events:

- `dragstart`: This event is fired on an element when a drag starts. Information such as the drag data and image to be associated with the drag operation can be set in the listener.
- `dragenter`: This event is fired when the cursor is hovered over an element for the first time while a drag begins. A drop operation is not allowed by default. There are one or more listeners that perform drag-and-drop operations. Usually, the listener highlights or marks the drop element where the drop can occur.
- `dragover`: This event is fired when the cursor is hovered over an element and a drag is in process.
- `dragleave`: This event is fired when the cursor leaves an element while a drag is in process. The listener will remove highlights or markers from the element where the drop can occur.
- `drag`: This event is fired on an element where the `dragstart` event was fired.

- `drop`: This event is fired on an element where the drop occurred. It is fired only if the drop is allowed. Users can cancel the drag operation by pressing the `Esc` key or releasing the mouse button on an invalid drop area.
- `dragend`: This event is fired on an element on which the drag was started to inform that the drag operation is complete, regardless of whether it is successful or not.

We will continue to make elements draggable from our example. We need to add a listener for the `dragstart` event and set the drag data within the listener, as follows:

```
var dragSource = querySelector("#sample_drag_id")
..draggable = true
..onDragStart.listen((MouseEvent event) {
  //...
});
```

If an element is made draggable, you cannot select the text by clicking-and-dragging with the mouse.



User must hold down the `Alt` key to select text with the mouse.



Dragging data

Each drag event has a `dataTransfer` property that is used to hold data associated with the drag operation. If you drag the selected text, then the associated data is text. If you drag an image, then the associated data is the image itself. The drag data combines the string representation of the format of the data and the data value. We will use the format of the data in the event listeners for the `dragenter` and `dragover` events to check whether the drop operation is allowed or not. You can set multiple drag data to call the `setData` method multiple times with different formats. To delete them, call the `clearData` method of the `dataTransfer` property, as shown in the following code:

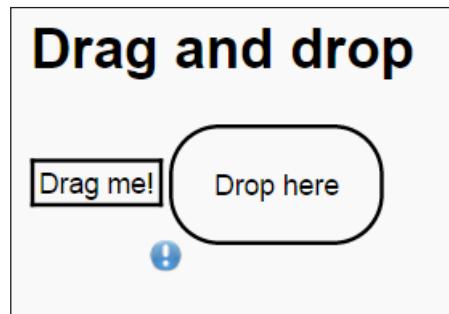
```
var dragSource = querySelector("#sample_dnd_id")
..draggable = true
..onDragStart.listen((MouseEvent event) {
  event.dataTransfer.setData("text/plain", "I'm draggable");
  event.dataTransfer.setData("text/data", "1234");
});
```

Dragging the feedback image

Usually, native drag-and-drop APIs automatically create translucent images that are generated from the drag target of the `dragstart` event, which follows the mouse pointer during the drag operation. You can use the `setDragImage` method of the `dataTransfer` property to specify a custom drag image. The first argument of this function is a custom drag image, which could be a reference to a real image, canvas, or other elements. The second and third arguments are offsets where the image should appear relative to the mouse pointer. The code is as follows:

```
var dragSource = querySelector("#sample_drag_id")
..draggable = true
..onDragStart.listen((MouseEvent event) {
  event.dataTransfer.setData("text/plain", "I'm draggable");
  event.dataTransfer.setDragImage(new
    ImageElement(src:'notification.png'), 0, 0);
});
```

The following feedback image will appear instead of the standard translucent image:



Dragging effects

The drag-and-drop API supports operations such as copy, move, link, and their combinations that may be performed on data that is draggable. We can use the `copy` operation to indicate that the data being dragged will be copied from its present location to the drop location. Similarly, you can use the `move` operation to indicate that the data being dragged will be moved, and the `link` operation indicates that connections will be created between the source and drop locations. You should specify which operation or combinations are allowed and are performed by setting the `effectAllowed` property of the `dragstart` event within a listener, as shown in the following code:

```
var dragSource = querySelector("#sample_drag_id")
..draggable = true
```

```
..onDragStart.listen((MouseEvent event) {
  event.dataTransfer.setData("text/plain", "I'm draggable");
  event.dataTransfer.setDragImage(new
    ImageElement(src:'notification.png'), 0, 0);
  event.dataTransfer.effectAllowed = 'copy';
});
```

In the preceding example, we allowed only the `copy` operation. Let's see all the values that we can use as the name for an operation:

- `none`: This operation means that no operation is permitted
- `copy`: This operation means that the drag data can only be copied from source to drop location
- `move`: This operation means that the drag data can only be moved from source to drop location
- `link`: This operation means that the drag data can only be linked from source to drop location
- `copyMove`: This operation means that the drag data can be copied or moved from source to drop location
- `copyLink`: This operation means that the drag data can be copied or linked from source to drop location
- `all`: This operation means that the drag data can be copied, moved, or linked from source to drop location

By default, the `effectAllowed` property allows all three operations. The permitted operation can be checked in a listener for the `dragenter` or `dragover` events via the `effectAllowed` property, and it should be set in a related `dropEffect` property to specify which single operation should be performed. The valid operations for the `dropEffect` property are `none`, `move`, or `link` only, and any other combinations are prohibited. The desired operation will change the mouse pointer, so the cursor might appear with a plus for the `copy` operation. The desired effect can be modified by a user by pressing the modifier keys. The exact keys vary by platform. On Windows OS, a user typically uses the `Shift` and `Ctrl` keys to switch between the `copy`, `move`, and `link` operations. During the `dragenter` and `dragover` events, we can modify both the `effectAllowed` and `dropEffect` properties to specify the supported operations by a drop target. The effect specified in `dropEffect` must be the one that is listed within the `effectAllowed` property. The value of the `dropEffect` property can tell us exactly the result of the drag operation. If the value of the `dropEffect` property is `none`, then the drag was cancelled; otherwise, the specified effect holds the performed operation. The drag-and-drop operation is considered complete after the dragevent is finished.



The `none` value can be used for either of the property to indicate that no drop operation is allowed at the target location.



The drop target

The drop target is a place where the dragged item may be dropped. The drop target is very important because most areas of a web page are not permitted to drop. Event listeners for the `dragenter` and `dragover` events are used to indicate a valid drop target through preventing default handling by canceling events, as shown in the following code:

```
var dropTarget = querySelector("#sample_drop_id")
..onDragOver.listen((MouseEvent event) {
    if (checkTarget(event.target)) {
        event.preventDefault();
        event.dataTransfer.dropEffect = 'copy';
    }
});
```

In the preceding code, we call the `checkTarget` method to be sure that the target is in the right place to be dropped. In our case, the drop target must have the `draggable` attribute, as shown in the following code:

```
bool checkTarget(Element target) {
    return target.attributes.containsKey('draggable');
}
```

However, it is common that the drop will be accepted or rejected based on the type of drag data within the `dataTransfer` property. In this case, we should check the property types of the `dataTransfer` property to decide whether the data can be accepted to be dropped. The code is as follows:

```
var dropTarget = querySelector("#sample_drop_id")
..onDragOver.listen((MouseEvent event) {
    if (checkTarget(event.target) &&
        checkTypes(event.dataTransfer.types)) {
        event.preventDefault();
        event.dataTransfer.dropEffect = 'copy';
    }
});
```

In the following code, the `checkTypes` function accepts only the `text/data` types specified through the `setData` method of `dataTransfer` inside the listener of the `dragstart` event:

```
bool checkTypes(List<String> types) {
    return types.contains("text/data");
}
```

You can now run the web application, drag your box with the `Drag me!` text, and drop it inside the box with the `Drop here` text. Let's polish our application and change the text of the drop zone.

Finishing a drop

When a user releases the mouse, the drag-and-drop operation ends. If this happened over an element that was identified as a valid drop target, the drag-and-drop API will fire a `drop` event at the target. The `dataTransfer` property of the `drop` event holds the data that is being dragged. To retrieve the dragged data, we will use the `getData` method of the `dataTransfer` property, as follows:

```
var dropTarget = querySelector("#sample_drop_id")
..onDragOver.listen((MouseEvent event) {
    if (checkTarget(event.target) &&
        checkTypes(event.dataTransfer.types)) {
        event.preventDefault();
        event.dataTransfer.dropEffect = 'copy';
    }
})
..onDrop.listen((MouseEvent event) {
    Element dropTarget = event.target;
    dropTarget.innerHTML = event.dataTransfer.getData('text/plain');
    event.preventDefault();
});
```

The `getData` method will retrieve the string value that was set when the `setData` method was called. When an empty string is returned from `getData`, this means that data of the specified type does not exist. At the end of the `getData` method, you need to call the `preventDefault` method of the event if you have accepted the drop. Here is the result of the drag-and-drop operation:



Finishing a drag

Finally, when the drag operation is complete, the drag-and-drop API generates a `dragend` event at the source element that received the `dragstart` event. The API generates that event regardless of the result of the drag-and-drop operation. The value of the `dropEffect` property can tell us the exact result of the drag operation. If the value of the `dropEffect` property is `none`, then the drag was cancelled; otherwise, the specified effect holds the performed operation. The drag-and-drop operation is considered complete after `dragevent` is finished.

The geolocation APIs

A **geolocation** API is a high-level interface used to locate information. It lets you find out where the user is and keep a track of his/her location when he/she moves. The geolocation API is device-agnostic of the underlying location source and doesn't care how the web browser determines the location. The following are the common sources for the location:

- GPS
- The network IP address
- RFID
- Wi-Fi
- The Bluetooth MAC address
- The GSM/CDMA cell ID
- User inputs

The API represents location by latitude and longitude coordinates.



The geolocation APIs do not guarantee returning the actual location of the device.



The geolocation API has the following classes:

- **Geolocation:** This class is used to determine the location information associated with the hosting device
- **Geoposition:** This class is used to store the coordinates and timestamp
- **Coordinates:** This class is used to store the location information and speed of the device

Determining the current location

Let's see how we can use the geolocation API to obtain information about the current location:

```
import 'dart:html';

void main() {
  window.navigator.geolocation.getCurrentPosition()
    .then((Geoposition geoposition) {
      querySelector("#latitude").text = geoposition.coords.latitude
        .toStringAsFixed(6);
      querySelector("#longitude").text = geoposition.coords.longitude
        .toStringAsFixed(6);
    }, onError: (PositionError error) {
      print(error.message);
    });
}
```

We can request the geolocation instance from the navigator property of the window object. The `getCurrentPosition` method of `geolocation` returns `Geoposition` in the `Future` object. When the `Future` will be resolved, we will assign the `geoposition` coordinates to the latitude and longitude HTML fields.

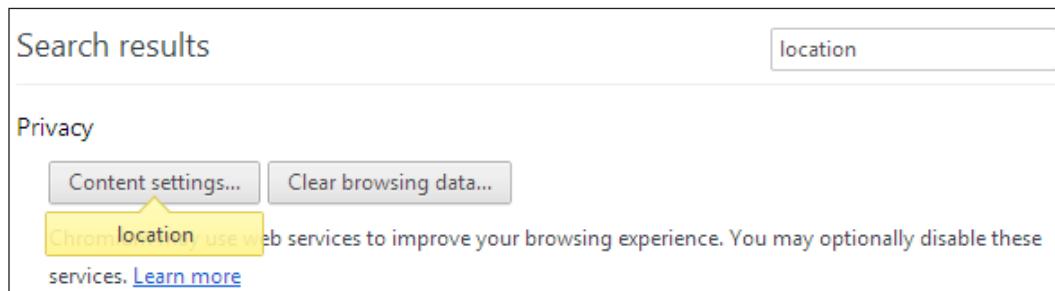


Dartium doesn't support geolocation, so run `pub serve` and iterate with Chrome web browser.

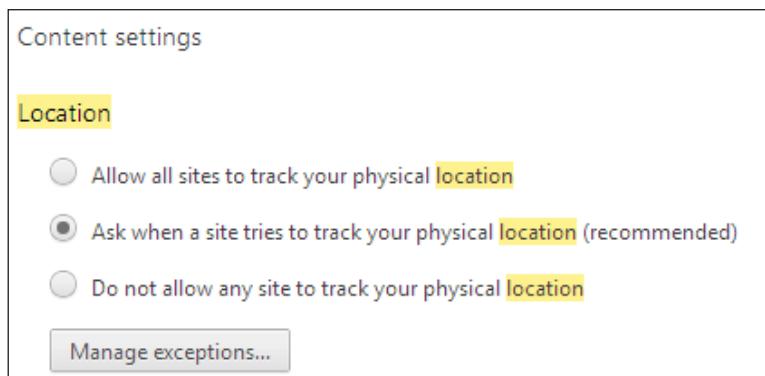


The website must have permission before use your location information. You can let websites use your location information automatically or obtain a permission request first. This is a common requirement when an API tries to interact with something outside a web context, especially to avoid sharing user-specific information. To see the geolocation information, follow these steps:

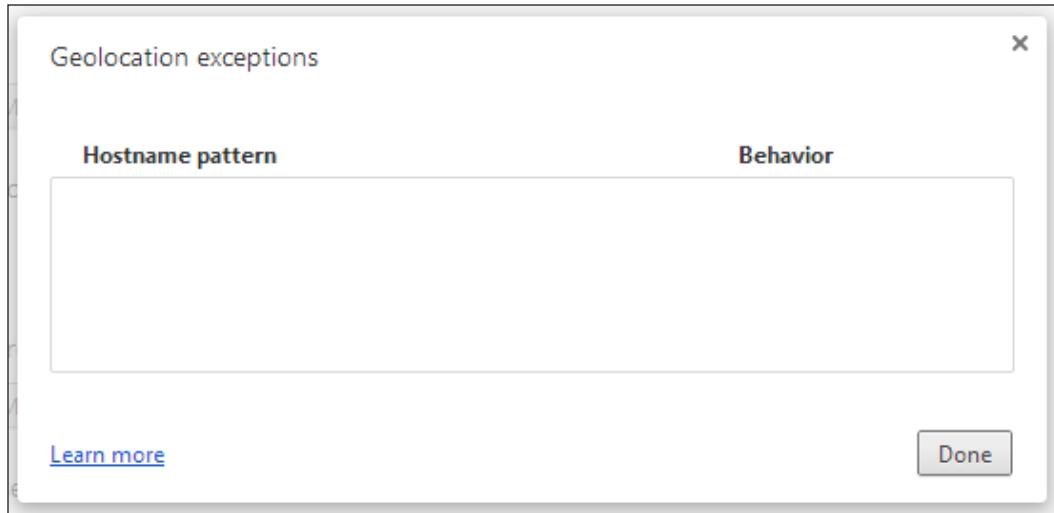
1. Open the **Settings** option in Chrome and type `location` in the search field, as shown in the following screenshot:



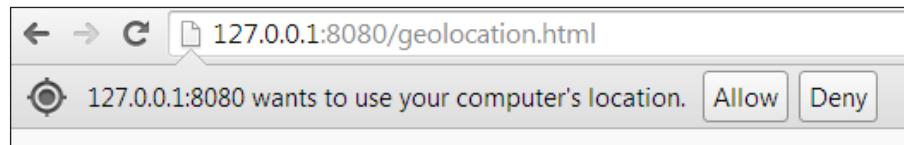
2. Open the **Content settings** pop-up dialog by clicking on the button of the same name, scroll down the content, and you'll find the **Location** settings:



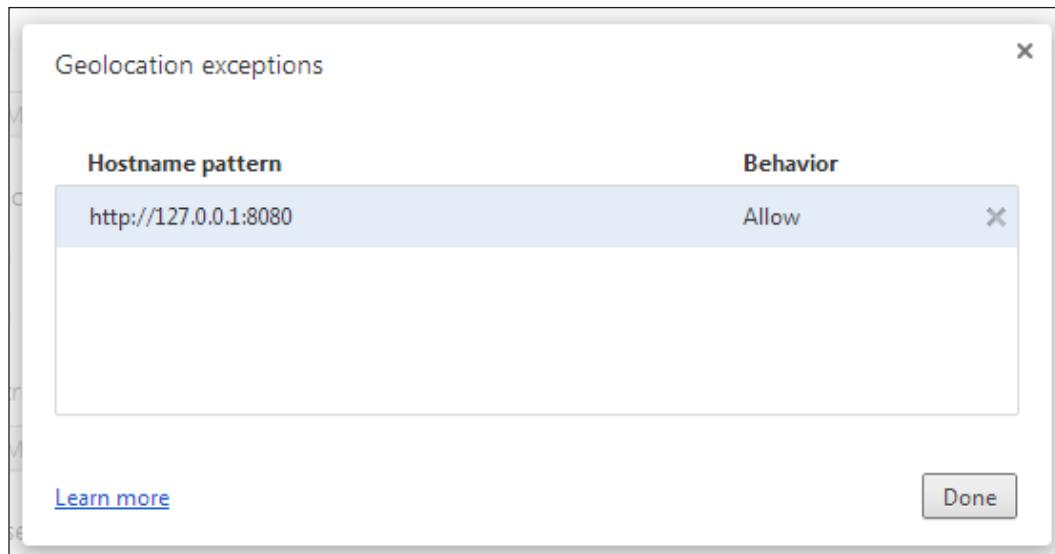
3. Choose the **recommended** option and open the **Geolocation exceptions** dialog by clicking on the **Manage exceptions** button. For now, it will not contain our website, as shown in the following screenshot:



4. Run the geolocation application. The Chrome web browser will ask you for permission to use your location information, as shown in the following screenshot:



5. You can allow or deny location requests for this website. Your choice completes the future permission requests with the value of geolocation. The website is then added to the list of **Geolocation exceptions**. Open the **Geolocation exceptions** dialog again to see your website:



Any error that occurs in the geolocation service will be printed:

```
Network location provider at 'https://www.googleapis.com/'  
: Returned error code 400.
```

6. We will now run `pub serve` from the root folder of our project:

```
>Loading source assets...  
Serving geolocation web on http://localhost:8080  
Build completed successfully  
[web] GET /geolocation.html => geolocation|web/geolocation.html  
[web] GET /geolocation.css => geolocation|web/geolocation.css  
[web] GET /packages/browser/dart.js => browser|lib/dart.js  
[Info from Dart2JS]:  
Compiling geolocation|web/geolocation.dart...  
[Info from Dart2JS]:  
Took 0:00:27.723586 to compile geolocation|web/geolocation.dart.  
Build completed successfully  
[web] GET /geolocation.dart.js => geolocation|web/geolocation.  
dart.js  
[web] GET /geolocation.dart.js.map => geolocation|web/geolocation.  
dart.js.map
```

7. Then, open the web application in the Chrome web browser to get the following result:



Your example will be more interesting if you add Google Maps to show your current position on a real map.

Geolocation on maps

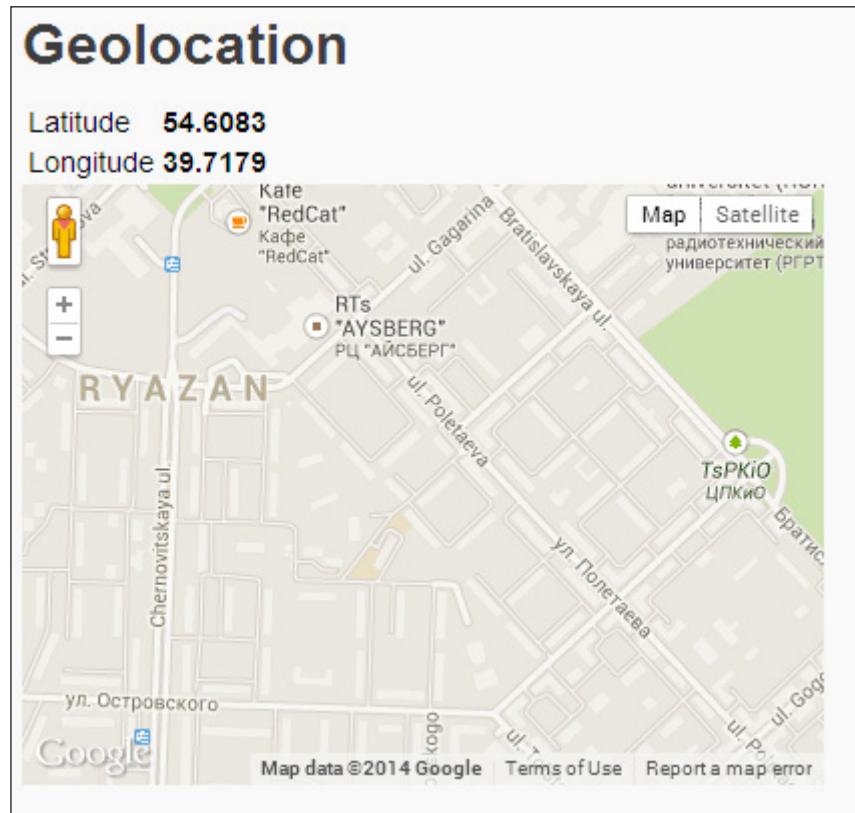
Add the `google_maps` package to your project and make the following changes to the code:

```
import 'dart:html';
import 'package:google_maps/google_maps.dart';

void main() {
    final mapOptions = new MapOptions()
        ..zoom = 15
        ..mapTypeId = MapTypeId.ROADMAP;
    final map = new GMap(querySelector("#map_canvas"), mapOptions);

    window.navigator.geolocation.getCurrentPosition()
        .then((Geoposition geoposition) {
            Coordinates coords = geoposition.coords;
            querySelector("#latitude").text = coords.latitude
                .toStringAsPrecision(6);
            querySelector("#longitude").text = coords.longitude
                .toStringAsPrecision(6);
            //
            map.center = new LatLng(coords.latitude, coords.longitude);
        }, onError: (PositionError error){
            print(error.message);
        });
}
```

In `mapOptions`, we specified the zoom and type of view. Maps can be presented in terms of the satellite, terrain, or hybrid view. Using the Google Maps API is very simple. Just add the `div` element with the specified ID to your HTML page. Then, execute the `pub serve` command to compile your code and run the server. When you open the **Geolocation** page in the Chrome web browser, you will get the following result:



Tracking the present location

Geolocation APIs can monitor the current location of your device using the `watchPosition` method. With the `enableHighAccuracy` parameter, the geolocation API starts to use more accurate hardware available on your device. This method returns a stream of geoposition coordinates. You only need to listen to the events to track changes in your current position. The code is as follows:

```
import 'dart:html';
import 'package:google_maps/google_maps.dart';
```

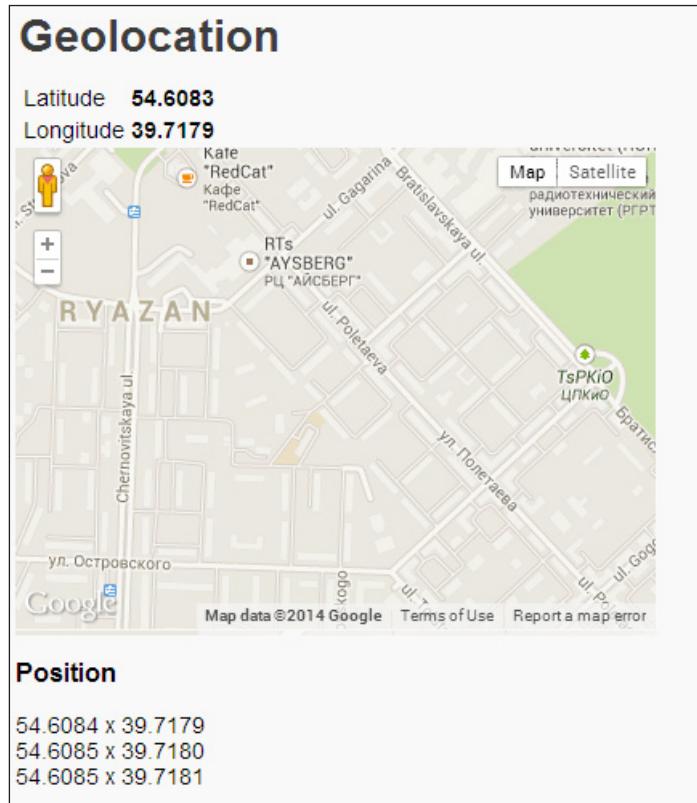
```

void main() {
    final mapOptions = new MapOptions()
    ..zoom = 25
    ..mapTypeId = MapTypeId.ROADMAP;
    final map = new GMap(querySelector("#map_canvas"), mapOptions);

    window.navigator.geolocation
    .watchPosition(enableHighAccuracy:true)
    .listen((Geoposition geoposition) {
        Coordinates coords = geoposition.coords;
        map.center = new LatLng(coords.latitude, coords.longitude);
        //
        querySelector("#location_tracker").append(newDivElement()
            ..text = coords.latitude.toStringAsPrecision(6) + " x " +
                coords.longitude.toStringAsPrecision(6));
    }, onError: (PositionError error){
        print(error.message);
    });
}
}

```

When you run the web application, you will receive the following result:



Canvas

The HTML5 **canvas** is a fantastic feature that allows you to code programmatic drawing operations. It has become very popular because it allows you to create and manipulate imagery directly within web pages. The canvas is one of the most flexible tags in new HTML5 features. This tag is a blank state. It defines a context object that users can draw inside. The actual drawing operations can be split in the following ways:

- Drawing a 2D context
- Drawing a 3D context formally known as a WebGL

A 2D context is available in all modern web browsers. It is more established and stable, while the 3D context is in the early process of being defined. Let's discuss the 2D context as it is more widely supported.

The canvas API is simple and powerful at the same time. You can only draw on a 2D bitmap surface using script. In the process of drawing, you do not have any DOM nodes for the shapes you draw. All that you produce is pixels, so you can forget about the performance penalties even if the image complexity increases.

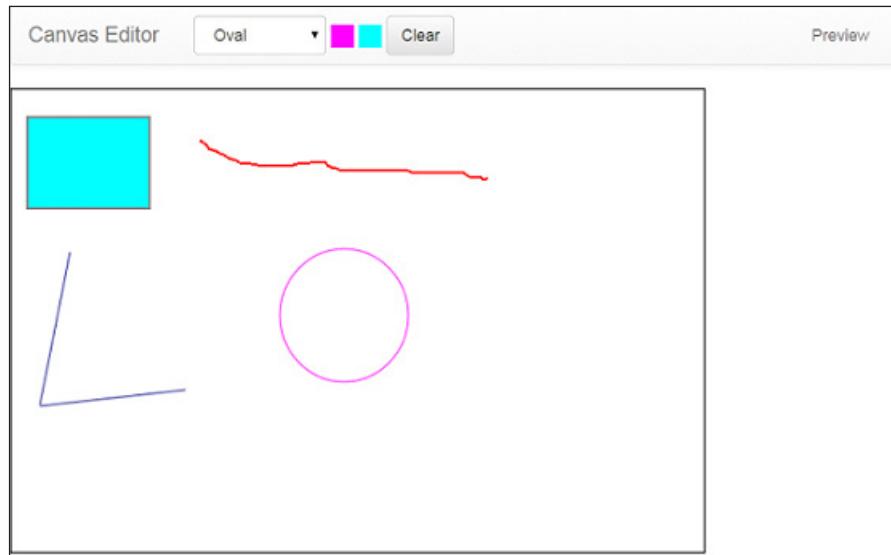
Drawing on a canvas is all about adding pixels in the appropriate coordinates on the screen. In general, the coordinates of the pixels on the screen correlate to the points in the canvas that are represented as a grid, but they can vary when we zoom in or out of the screen or when a canvas is resized with CSS. The key point on the grid is the origin located in the left-hand side corner of the canvas with coordinates (0, 0). Each shape drawn on the canvas has an offset of x and y axes and size by width and height.

Example – the canvas editor

The canvas API gives you access to perform the following actions:

- Draw shapes such as rectangles, ellipses, lines, and so on
- Render text
- Pixel manipulation
- Fill colors in areas, shapes, or text
- Create gradients and patterns to fill areas, shapes, or text
- Copy images, other canvases, or video frames
- Export the content of a canvas to a file

We don't want to pass through all these features with simple examples, so let's just create a canvas painting application to discover how to use many of them practically. You can find a prepared project in the code that accompanies this book. The application is based on Bootstrap 3.2 and jQuery 1.11.1. I have made a port of `bootstrap-colorselector` (<https://github.com/flaute/bootstrap-colorselector>) to Dart specially to show you how easily it can be done. In the following screenshot, you can see it running in the web browser application:



At the top of the application, we placed a **Bootstrap** navigation bar. It contains a `select` component with a drop-down option representing a list of available tools, such as **Pen**, **Rectangle**, **Line**, and **Oval**. There are two color selectors that we use to stroke and fill canvas styles. The **Clear** button helps wipe out the content of a canvas. Last but not least, the **Preview** button opens another window with the content of a canvas. Let's go deeper to discover different parts of the application and how they communicate with each other.

Beginning with HTML

The content of the head tag is shown in the following code:

```
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-
scale=1">
<title>Canvas</title>
```

Supporting Other HTML5 Features

```
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">

<link rel="stylesheet" type="text/css" href="css/bootstrap-colorselector.css" />
<link rel="stylesheet" type="text/css" href="css/index.css">

<!-- HTML5 Shim and Respond.js IE8 support of HTML5 elements and media queries -->
<!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
<!--[if lt IE 9]>
    <script src="https://oss.maxcdn.com/html5shiv/3.7.2/html5shiv.min.js"></script>
    <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
<!--[endif]-->

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>

<script async type="application/dart" src="index.dart"></script>
<script async src="packages/browser/dart.js"></script>
```

All bootstrap style sheets and the JavaScript code included in the HTML page are downloaded from a free and public **content delivery network (CDN)**. The html5shiv and respond libraries help run HTML5 features on IE Version 8 or higher.

The body of the web page is split into two sections. The first one is the responsive navbar component that serves as a navigation header for our application. It is collapsed and toggled in a mobile view, and then becomes horizontal if the available width space increases. We split the navigation bar into two subsections. The first subsection is the primary toolbar that keeps all components related to the direct canvas manipulation. The other one is the secondary toolbar that contains components that play a supporting role in the application.

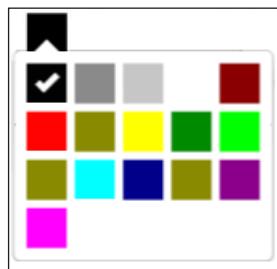
The first component of the primary toolbar is `toolSelector`, which keeps all the registered tools, as shown in the following code:

```
<div class="form-group">
<select class="form-control" id="toolSelector"
    name="toolSelector">
</select>
</div>
```

The next two components are `stroke_color_selector` and `fill_color_selector`, which contain lists of colors presented as palettes. We will include only the first one in the following code snippet because of their similarity:

```
<div class="form-group">
<select class="stroke_color_selector">
<option value="#000000" data-color="black">Black</option>
<option value="#808080" data-color="gray">Gray</option>
<option value="#C0C0C0" data-color="silver">Silver</option>
<option value="#FFFFFF" data-color="white">White</option>
<option value="#800000" data-color="maroon">Maroon</option>
<option value="#FF0000" data-color="red">Red</option>
<option value="#808000" data-color="olive">Olive</option>
<option value="#FFFF00" data-color="yellow">Yellow</option>
<option value="#008000" data-color="green">Green</option>
<option value="#00FF00" data-color="lime">Lime</option>
<option value="#008080" data-color="olive">Teal</option>
<option value="#00FFFF" data-color="aqua">Aqua</option>
<option value="#000080" data-color="navy">Navy</option>
<option value="#0000FF" data-color="olive">Blue</option>
<option value="#800080" data-color="purple">Purple </option>
<option value="#FF00FF" data-color="fuchsia">Fuchsia</option>
</select>
</div>
```

Both of the components, `stroke_color_selector` and `fill_color_selector`, are represented by the component ported from `bootstrap-colorselector`. The result of the color palette is shown in the following screenshot:



The last one in this group of components is the **Clear** button. The user can click on this button to clear the content of the canvas, as shown in the following code:

```
<div class="form-group">
  <button type="button"
    class="btn btn-default clear-btn">Clear</button>
</div>
```

The secondary toolbar is right aligned. It has only a **Preview** button component, as shown in the following code:

```
<ul class="nav navbar-nav navbar-right">
  <li><a href="#" class="previewBtn">Preview</a></li>
</ul>
```

Now, let's see our canvas components:

```
<div class="col-lg-12 text-center canvas-container">
  <canvas class="canvas view-canvas" width="600px"
    height="400px">
    <p>Unfortunately, your browser is currently unsupported by
      our web application.</p>
  </canvas>
  <canvas class="canvas draw-canvas" width="600px"
    height="400px">
    <p>Unfortunately, your browser is currently unsupported by
      our web application.</p>
  </canvas>
</div>
```

We use two canvases in our application. The first one is marked with a **view-canvas** class. It is only used to present data opposite to the second one marked with the **draw-canvas** class. The purpose of the second canvas is that all the tools must draw here. When their drawing operation ends, the pixels that they generated are then moved into the first canvas.

Moving to the main function

The foundation of our application is the components separated in to two types: widgets and tools. Widgets are components dealing with HTML elements and include extra behavior. All tool components are based on the **Tool** class. We use the tool-based components to draw in the canvas. Here is the main function of our application:

```
// Calculate absolute value of number
num abs(num value) => value < 0 ? -value : value;
```

```
// Return offset of mouse pointer from any mouse event
Point offset(MouseEvent event) => event.offset;

void main() {
    // Create an instance of [CanvasWidget]
    CanvasWidget canvas = new CanvasWidget(
        ".view-canvas", ".draw-canvas");
    // Create an instance of [BrushSelectorWidget]
    ToolSelectorWidget tools = new ToolSelectorWidget(
        ".tool-selector")
        ..onToolSelected.pipe(canvas);
    // Create and add tools to [ShapeSelectorWidget]
    tools.addTool(new Pen());
    tools.addTool(new Rectangle(), select:true);
    tools.addTool(new Line());
    tools.addTool(new Oval());
    // Create a stroke color selector
    new ColorSelectorWidget(".stroke_color_selector",
        ColorSelectedEvent.STROKE_COLOR, 'black')
        ..onColorSelected.pipe(canvas);
    // Create a fill color selector
    new ColorSelectorWidget(".fill_color_selector",
        ColorSelectedEvent.FILL_COLOR, 'aqua')
        ..onColorSelected.pipe(canvas);
    // Register a clear button listener
    querySelector(".clear-btn").onClick.listen((MouseEvent event) {
        canvas.clear();
    });
    // Register a preview button listener
    querySelector(".preview-btn").onClick.listen((MouseEvent event)
    {
        event.preventDefault();
        window.open(canvas.viewCanvas.toDataUrl("image/png"),
            "Image Preview");
    });
}
```

Now, let's discuss each component in order to better understand how they interact with each other.

The CanvasWidget component

As mentioned earlier, the application uses two canvas components – one on top of the other. Both of them are available via selectors. When we create an instance of the `CanvasWidget` component, we pass these selectors as arguments of the constructor. This component references the `Tool` component that is used to draw. The code is as follows:

```
/**  
 * Canvas widget listens for mouse events from [CanvasElement] to draw  
 * with selected tool.  
 */  
class CanvasWidget implements StreamConsumer {  
  
    CanvasElement _viewCanvas, _drawCanvas;  
  
    CanvasElement get viewCanvas => _viewCanvas;  
    CanvasElement get drawCanvas => _drawCanvas;  
    CanvasRenderingContext2D get context => _viewCanvas.context2D;  
    CanvasRenderingContext2D get drawContext => _drawCanvas.context2D;  
  
    Tool _tool;  
  
    /**  
     * Create an instance of CanvasWidget. The [viewCanvasSelector] and  
     * [drawCanvasSelector] need to find CanvasElements.  
     */  
    CanvasWidget(String viewCanvasSelector, String drawCanvasSelector) {  
        // Find canvas elements  
        _viewCanvas = querySelector(viewCanvasSelector);  
        _drawCanvas = querySelector(drawCanvasSelector);  
        // Add mouse event listeners  
        _drawCanvas.onMouseDown.listen((evt) =>  
            _tool.beginDraw(drawContext, offset(evt)));  
        _drawCanvas.onMouseMove.listen((evt) =>  
            _tool.drawing(drawContext, offset(evt)));  
  
        var _finishDraw = (evt) {  
            _tool.finishDraw(drawContext, offset(evt));  
            copyContext();  
        };  
  
        _drawCanvas.onMouseUp.listen(_finishDraw);  
        _drawCanvas.onMouseLeave.listen(_finishDraw);  
    }  
}
```

The user holds down the mouse button when he/she begins to draw. As a result, the `onMouseDown` listener invokes the `beginDraw` method of the `Tool` class. The `drawContext` method and the `offset` of the mouse coordinates are passed as parameters of this method. The drawing method of the `Tool` class is called every time the mouse is moved. Finally, the `finishDraw` method is called when the user releases the mouse button, and we call the local `copyContext` method to copy the content of the draw canvas to the view canvas. The `CanvasWidget` component implements the `StreamConsumer` interface via the `addStream` method to listen to two sorts of events, that is, `ColorSelectedEvent` from `ColorSelectorWidget` and `ToolSelectedEvent` from `ToolSelectorWidget`. The `addStream` method processes the incoming events, as shown in the following code:

```
/**  
 * Copy drawn image from draw canvas into view context.  
 * After all it clears the draw canvas.  
 */  
copyContext() {  
    context.drawImage(_drawCanvas, 0, 0);  
    _drawCanvas.context2D.clearRect(0, 0,  
        _drawCanvas.width, _drawCanvas.height);  
}  
  
/**  
 * Clear the view canvas  
 */  
clear() {  
    context.clearRect(0, 0, _viewCanvas.width, _viewCanvas.height);  
}  
  
/**  
 * Consumes the elements of [stream].  
 * Listens on [stream] and does something for each event.  
 */  
Future addStream(Stream stream) {  
    return stream.listen((event) {  
        if (event is ColorSelectedEvent) {  
            if (event.type == ColorSelectedEvent.STROKE_COLOR) {  
                drawContext.strokeStyle = event.value;  
            } else {  
                drawContext.fillStyle = event.value;  
            }  
        } else if (event is ToolSelectedEvent) {  
            //  
        }  
    });  
}
```

```
        _tool = event.tool;
    }
}).asFuture();
}
```

We use the `clear` method to wipe out the content of `_viewCanvas`.

The ToolSelector widget

This component keeps the tool-based components and presents them in `SelectElement`. A tool-based component can be added via the `addTool` method, as shown in the following code:

```
class ToolSelectorWidget {
    SelectElement _selectElement;
    Tool selectedTool;
    Map<String, Tool> _tools = new Map<String, Tool>();

    Iterable<String> get toolsNames => _tools.keys;

    StreamController<ToolSelectedEvent> _toolSelectedController =
        new StreamController<ToolSelectedEvent>();
    Stream<ToolSelectedEvent> get onToolSelected =>
        _toolSelectedController.stream;

    ToolSelectorWidget(String selector) {
        _selectElement = querySelector(selector);
        _selectElement.onChange.listen((Event event) {
            selectTool(_selectElement.value);
        });
    }

    void addTool(Tool tool, {bool select: false}) {
        _tools[tool.name] = tool;
        OptionElement item = new OptionElement(
            data: tool.name, value: tool.name);
        _selectElement.append(item);
        if (select) {
            selectTool(tool.name);
        }
    }

    Tool getTool(String name) {
        if (_tools.containsKey(name)) {
            return _tools[name];
        }
    }
}
```

```
        }
        throw new Exception("Brush with $name not found");
    }

    selectTool(String name) {
        selectedTool = getTool(name);
        _toolSelectedController.add(new ToolSelectedEvent(selectedTool));
        _selectElement.value = selectedTool.name;
    }
}
```

When the user selects a new tool, this component generates a `ToolSelectedEvent` method with the selected tool instance as the parameter. The `StreamController` method is used to broadcast `ToolSelectedEvent` to any listener, that is, `CanvasWidget`.

The ColorSelector widget

This widget is a port of `bootstrap-colorselector` to Dart. This component creates a drop-down color palette from a predefined set of colors only. We have set predefined colors for and via the HTML markup. Every time the user chooses a new color, the `StreamController` method broadcasts a `ColorSelectedEvent` event to the `CanvasWidget` class.

The Tool class

Our application has an abstract `Tool` class to abstract the common behavior and properties of all the tool-based components, as shown in the following code:

```
/**
 * Abstract class defines common behavior and properties
 * for all tools.
 */
abstract class Tool {
    bool isDrawing = false;
    Point startPoint;

    String get name;

    void beginDraw(CanvasRenderingContext2D context, Point point) {
        isDrawing = true;
        startPoint = point;
    }
}
```

```
void drawing(CanvasRenderingContext2D context, Point point);

void finishDraw(CanvasRenderingContext2D context, Point point) {
    if (isDrawing) {
        drawing(context, point);
        isDrawing = false;
    }
}
```

The `isDrawing` property reflects the status of the drawing operation. The `startPoint` property simply holds the cursor coordinates relative to the canvas when the user starts drawing on the canvas. The read-only property `name` returns the name of the tool. This name is used when the tool is added to `ToolSelectorWidget`. Each tool has three methods, and the whole drawing process is split into the following three phases:

- **Start the drawing phase:** The program calls the `beginDraw` method when a user starts holding down the mouse button. We always switch on the `isDrawing` property and remember the cursor coordinates in `startPoint`.
- **Drawing phase:** The drawing method is called every time the user moves the cursor. Implementation of this method strongly depends on the tool, so we do not implement it in the abstract class `Tool`.
- **End the drawing phase:** The program invokes the `finishDraw` method when the user releases the mouse button. In this method, we need to call the drawing method for the last time and switch off `isDrawing` only if the drawing process has happened.

Now, it's time to look at our tools implementation in detail.

The Pen tool

As the canvas element doesn't directly support drawing a single point, we use lines instead. Drawing in a canvas is similar to using a virtual pen. At the beginning, we must call `beginPath` of `context` where we begin drawing. This method creates a new drawing path, so future drawing commands will be directed to the path and will be used to build the path. We start our path by moving to the `startPoint` coordinates with the `moveTo` method of `context`, as shown in the following code:

```
/**
 * Simple pen tool
 */
class Pen extends Tool {
```

```
String get name => "Pen";

@Override
void beginDraw(CanvasRenderingContext2D context, Point point) {
    super.beginDraw(context, point);
    context.beginPath();
    context.moveTo(startPoint.x, startPoint.y);
}

@Override
void drawing(CanvasRenderingContext2D context, Point point) {
    if (isDrawing) {
        context.lineTo(point.x, point.y);
        context.stroke();
    }
}
}
```

Every time a user moves the mouse, the program calls the `drawing` method to connect our drawn path to next line's point with the `lineTo` method of `context`. Finally, it calls the `stroke` method to draw the shape by stroking its outline.

The Line tool

The behavior of the **Line** tool is similar to that of the **Pen** tool, but with a different drawing logic. We only implement the `drawing` method of the `Tool` class to draw our shape in a path. A path is a list of subpaths, and each of them is a list of points that are connected by straight or curved lines. Each one also contains a flag that indicates whether it is closed, so the last point of the closed subpath is connected to the first point by a straight line, as illustrated by the following code:

```
/**
 * Line tool is used to create lines.
 */
class Line extends Tool {

    String get name => "Line";

    @Override
    void drawing(CanvasRenderingContext2D context, Point point) {
        if (isDrawing) {
            context.clearRect(0, 0,
                context.canvas.width, context.canvas.height);
            context.beginPath();
        }
    }
}
```

```
        context.moveTo(startPoint.x, startPoint.y);
        context.lineTo(point.x, point.y);

        context.stroke();
        context.closePath();
    }
}
}
```

We must always clear the whole drawing canvas and start a new path via the `beginPath` method of `context`. We move the first point to the `startPoint` position and draw the line within the current cursor coordinates. Finally, we draw the line shape by stroking its outline and close the drawing path.

The Rectangle tool

We can draw a rectangle with individual lines, but the **Rectangle** tool makes the task much easier. The `context` object has the following methods to draw rectangles:

- `strokeRect`: This method uses the current stroke style to draw the box that outlines the given rectangle onto the canvas
- `fillRect`: This method uses the current fill style to draw the given rectangle onto the canvas
- `clearRect`: This method clears all the pixels on the canvas in the given rectangle to transparent black

The easiest way to draw a rectangle on the canvas is use the `fillRect` method of `context`. The `fillRect` method uses color from the `fillStyle` property, black by default. The `Rectangle` class, as shown in the following code:

```
/**
 * Rectangle tool
 */
class Rectangle extends Tool {

    String getName => "Rectangle";

    @override
    void drawing(CanvasRenderingContext2D context, Point point) {
        if (isDrawing) {
            context.clearRect(0, 0,
                context.canvas.width, context.canvas.height);

            int x = min(point.x, startPoint.x).round(),

```

```
    y = min(point.y, startPoint.y).round(),
    w = abs(point.x - startPoint.x).round(),
    h = abs(point.y - startPoint.y).round();

    context.fillRect(x, y, w, h);
    context.strokeRect(x, y, w, h);
}
}
}
```

In the preceding code, we drew a rectangle with `fillRect` and finally called `strokeRect` to draw the border line shape by stroking its outline.

The Oval tool

Drawing ovals is a breeze too. The easiest way to draw ovals is using the `arc` method of `context`. The `arc` method takes the following five parameters:

```
arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

The `x` and `y` parameters are the coordinates of the center of the oval on which the arc should be drawn. The `radius` parameter is the radius of the oval. The `startAngle` and `endAngle` parameters define the start and end coordinates of the arc in radians, measured from the `x` axis along with the curve of the oval. Finally, the last parameter is `anticlockwise`, which tells the canvas to draw the arc anticlockwise. The code is as follows:

```
/**
 * This tool transforms a drawing context into a rectangle
 * enclosing the oval and uses the arc method to draw it.
 */
class Oval extends Tool {

    String getName => "Oval";

    @override
    void drawing(CanvasRenderingContext2D context, Point point) {
        if (isDrawing) {
            context.save();
            context.clearRect(0, 0,
                context.canvas.width, context.canvas.height);
            context.beginPath();

            var rx = (point.x - startPoint.x) / 2;
            var ry = (point.y - startPoint.y) / 2;
```

```
context.translate(startPoint.x + rx, startPoint.y + ry);

rx = abs(rx);
ry = abs(ry);
if (rx < ry)
{
    context.scale(1, abs(ry / rx));
    context.arc(1, 1, rx, 0, 2 * PI, false);
}
else
{
    context.scale(abs(rx / ry), 1);
    context.arc(1, 1, ry, 0, 2 * PI, false);
}

context.stroke();
context.restore();
}
}
}
```

First, we saved the current context's settings, cleared it, and started a new path. Then, we calculated the coordinates of the center of our oval and placed the result in `rx` and `ry`. After that, we moved the origin from `(0, 0)` to a new place via the `translate` method of `context`. For now, we took the absolute values for `rx` and `ry` so that we can draw an oval in different directions. Depending on the drawing direction, we can draw an oval by scaling it along the `x` or `y` axis. Finally, we draw the oval shape by stroking its outline and restored the parameters of `context` to prepare it for further use.

How to clear the context

Many times, we try to clear the drawing context before we actually start to draw a new shape in the drawing context. So, now it's time to look at the following self-explanatory code from the `CanvasWidget` class that demonstrates how the view context can be cleaned:

```
/**
 * Clear the view canvas
 */
clear() {
    context.clearRect(0, 0,
        _viewCanvas.width, _viewCanvas.height);
}
```

How to preview the context

If you need to preview the result of what you've done in the view canvas, you can open a new window with the content of the view canvas as shown here:

```
// Register a preview button listener
querySelector(".preview-btn").onClick
.listen(MouseEvent event) {
    event.preventDefault();
    window.open(canvas.viewCanvas.toDataUrl("image/png"),
    "Image Preview");
});
```

Now if you click on the **Preview** link, you will see the next result in a new window in your web browser, as shown in the following screenshot:



Here, we opened a new browser window with the image data URI directly and we could save it in the represented format. The data URI format shown in the address of the web browser is as follows:

```
data: [< MIME-type >] [< charset=<encoding> >] [< base64 >], < data >
```

The `toDataUrl` method has an optional second parameter, `quality`. It represents the image quality in the range of 0.0 to 1.0 when the requesting type is `image/jpeg` or `image/webp`.

Summary

To summarize, we will discuss the important facts about how to support other HTML5 features in Dart.

The web notification API allows you to display notifications outside the context of the web page. The user agent defines the best presentation of notifications, which depends on the location of the device. The web notification API is available as a part of the `dart:html` package and allows you to send information to a user even if the application is idle. When you build web applications, you can use the notification API in event handlers or polling functions to notify users.

Before you send any notifications to the user, the website must have permissions. Users can let websites send notifications automatically or with the permission request first. All the websites that request access to the notification API are added to the list of the notification exceptions. The constructor of the `Notification` class has optional properties that help you create notifications with the body and icon.

Dragging-and-dropping is a way to convert pointing device movements and clicks to special commands that are recognized by software to provide quick access to common functions of a program. Native drag-and-drop support in a browser means faster and more responsive web applications. Each drag event has a `dataTransfer` property that is used to hold data associated with a drag operation. The drop target is very important because most areas of the web page are not permitted to drop.

The geolocation API is a high-level interface used for location information. The API is device-agnostic of the underlying location source and is not affected by how the web browser determines a location. Using the Google Maps API with the geolocation API is very simple.

The HTML5 canvas is a fantastic feature that allows you to code programmatic drawing operations. It has become very popular because it allows you to create and manipulate imagery directly within web pages. It defines a `context` object that users can draw inside.

In the next chapter, we will discuss the different aspects of security. As a best practice, we will focus on validation input and escape and filter output data in our web applications.

12

Security Aspects

In this chapter, we will talk about different aspects of security. You will learn about validation of user data input and security best practices in our web application. This chapter covers the following topics:

- Web security
- Securing a server
- Securing a client
- Security best practices

Web security

Crime is a disease that plagues the minds of many individuals. Hackers are interested in everything from personal mailbox credentials to bank account details. The responsibility of maintaining security lies with web developers. Developers should use HTTPS to access web pages and resources with sensitive data.

Transport Layer Security and Secure Socket Layer at a glance

Secure Socket Layer (SSL) is one of the most common protocols in use on the Internet today. SSL is capable of securing any transmission over TCP. The **Transport Layer Security (TLS)** protocol is a successor of SSL and is based on the older SSL specifications. TLS Version 1.0 was defined for the first time in January 1999 as an upgrade of SSL Version 3.0. Both of them are based on asymmetric keys to encrypt data and digital certificates for authentication through an untrusted third party. We use TLS in a client-server model, but the client usually does not provide a certificate. Instead, the server is responsible for its own authentication through signed certificates and encryption via public and private keys.

There are several versions of protocols used in web browsing, e-mail, internet faxing, instant messaging, and **Voice over IP (VoIP)**. TLS has the following benefits:

- It encrypts information
- It provides authentication
- It accepts credit card payments on websites
- It protects against phishing
- It adds power to brands and improves customer trust

Information submitted on the Internet passes through more than one node in the network before reaching the final destination, so it can be obtained by a third party. A TLS certificate inserts random characters into the original information to change it beyond recognition so that only the proper encryption key can help decrypt it. Server certification is another type of protection issued when the server's owner obtains a TLS certificate. This certificate is available to the client to validate that the TLS certificate is up to date and the client's information is being delivered to the right place. Online businesses that use credit card payments must be in compliance with the Payment Card Industry standards. This means that the server needs a TLS certificate with the proper encryption of at least 128 bits. Online businesses often offer site seals and other brand images to indicate that a trusted encryption is in use. This information gives customers an added level of assurance and creates trust between the customer and the business.

The TLS certificate

It is really complicated to decide at which level we can address the TLS protocol in the TCP/IP stack. The TLS security protocol describes how algorithms should be used and how the TLS certificate establishes a secure connection. To get a certificate, you must create a **Certificate Signing Request (CSR)** on your server. This process creates a pair of private and public keys on your server. Then, you must send the CSR datafile that contains the public key to a **Certified Authority (CA)** in order to obtain the TLS certificate. The CA creates a data structure from the CSR file to match a private key in the future. The CA never sees the private key and it can't be compromised. Once you receive the TLS certificate, you can install it on the server. Dart uses the **Network Security Services (NSS)** library of Mozilla to handle TLS. We need to use certutil, a certificate database tool from NSS Security Tools, to manipulate the certificate database. You can obtain the source code and quickly build certutil for your platform, but I have installed the following prebuilt version of the program on my Ubuntu workstation:

```
sudo apt-get install libnss3-tools
```

The process of installation is successful and you can now check the result by running the program with the following command:

```
certutil
```

On receiving the request, the program returns information on how to use it and gives a list of available command options, as follows:

```
certutil - Utility to manipulate NSS certificate databases
Usage: certutil <command> -d <database-directory> <options>
...
```

For now, we want to create a command-line application project with the name `server` in Dart Editor. Then, open the terminal and go into the `bin` directory of our project.



In real life, you must obtain a real certificate from a CA such as Thawte, Entrust, and others.

It is recommended to use self-signed TLS certificates for development and testing, but they are not recommended for production sites.

Follow the next steps to create a self-signed CA certificate for development and testing purposes inside the `bin` directory:

1. Create an NSS database in the `pkcert` folder. The folder name should be the name of the NSS database used on our server:

```
mkdir -p pkcert
certutil -N -d sql:/home/akserg/Project/server/bin/pkcert
```

The `-N` command option creates a new certificate and key databases. Specify the prefix `sql` in front of the full path to the database folder as Dart uses the new SQLite database (`cert9.db`, `key4.db`, and `pkcs11.txt`) rather than a legacy security database (`cert8.db`, `key3.db`, and `secmod.db`). The `certutil` command will ask us to enter a password that will be used to encrypt our keys. Let's set the password to `changeit`.

2. Create a self-signed CA certificate with the following command:

```
certutil -S -s "CN=CA Issuer" -n CACert -x -t "C,C,C" -v 120 -m
1234 -d sql:/home/akserg/Project/server/bin/pkcert
```

The `-S` command option creates an individual certificate and adds it to a certificate database. The text after `-S` option provides a subject that identifies an owner of certificate. The `-X` option tells the `certutil` command that the created certificate is self-signed. The `-V` option sets the number of months for which a new certificate will be valid. The `-M` option sets a unique serial number to the certificate being created. When we run the `certutil` command, it asks us to press the keys on the keyboard to create a random seed that will be used in the creation of our key.

3. We now have a CA certificate and need to generate the key and certificate signing request. Let's do that with the following command:

```
certutil -R -s "CN=localhost, O=Mastering Dart, L=Cape Town,  
ST=WC, C=CA" -p "+27 21 1234567" -o mycert.req -d sql:/home/  
akserg/Project/server/bin/pkcert
```

The `-R` command option creates a certificate request file that can be submitted to a CA to be processed into a finished certificate. We specify the subject to identify the certificate owner; in this case, it's me. Extra information such as your telephone number can be an input as well. Output defaults to the output file marked with the `-O` option. When we run the `certutil` command, it asks for a password, and we can generate the key with a random seed again.

4. Now, we can see the list of keys in the database with the following command:

```
certutil -K -d sql:/home/akserg/Project/server/bin/pkcert
```

The result will be as follows:

```
< 0> rsa e22c881d9eb382ea69257410cf464dfedcd49354 NSS  
Certificate DB: CACert  
< 1> rsa b909266e0d5a14523158bfc7903ea9460fad2da6 (orphan)
```

5. We need to sign in the key with the following command:

```
certutil -C -m 2345 -i mycert.req -o mycert.crt -c CACert -  
d sql:/home/akserg/Project/server/bin/pkcert
```

6. Finally, it's time to add a certificate to the database with the following command:

```
certutil -A -n localhost_cert -t "p,p,p" -i mycert.crt -d sql:/  
home/akserg/Project/server/bin/pkcert
```

The name of our certificate is `localhost_cert` after the `-n` option.

7. You can see the information about a specific certificate with the following command:

```
certutil -L -n localhost_cert -d  
sql:/home/akserg/Project/server/bin/pkcert
```

The result is as follows:

```
Certificate:  
Data:  
    Version: 3 (0x2)  
    Serial Number: 2345 (0x929)  
    Signature Algorithm: PKCS #1 SHA-1 With RSA  
Encryption  
    Issuer: "CN=CA Issuer"  
    Validity:  
        Not Before: Thu Aug 21 17:15:05 2014  
        Not After : Fri Nov 21 17:15:05 2014  
    Subject: "CN=Sergey Akopkokhyants,O=Mastering  
Dart,L=Cape Town,ST=WC, C=CA"  
    Subject Public Key Info:  
        Public Key Algorithm: PKCS #1 RSA Encryption  
...
```

8. Alternatively, you can validate a specific certificate with the following command:

```
certutil -V -n localhost_cert -b 9803201212Z -u SR -e -l -d  
sql:/home/akserg/Project/server/bin/pkcert
```

The result is as follows:

```
localhost_cert : Peer's Certificate has expired.  
localhost_cert : Peer's certificate has been marked as not  
trusted by the user.
```

Now that we are done with our self-signed certificate, it's time to go back to our server code and secure it.

Securing a server

Open the `server.dart` file and type the following lines:

```
import 'dart:io';

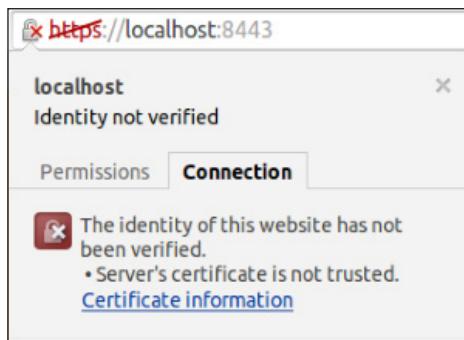
main() {
    var pkcertDB = Platform.script.resolve('pkcert').toFilePath();
    SecureSocket.initialize(database: pkcertDB,
        password: 'changeit');

    HttpServer
        .bindSecure(InternetAddress.ANY_IP_V6, 8443,
            certificateName: 'localhost_cert')
        .then((server) {
            server.listen((HttpRequest request) {
                request.response.write('Hello, world!');
                request.response.close();
            });
        });
    });
}
```

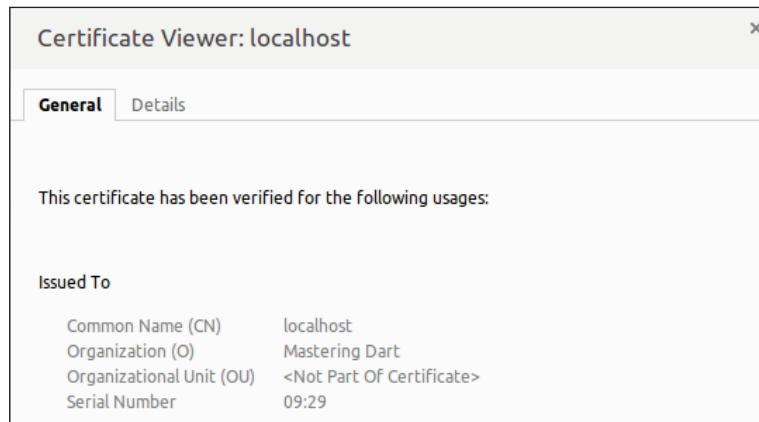
This is an implementation of the well-known `Hello, World!` example. I always keep the password of my certificate in the code only for demonstration purposes. Please keep your password in an external encrypted file. The code of the server is pretty straightforward. One small exception is that it references `SecureSocket` instead of the `Socket` class. By calling a static `initialize` method of this class, we initialize the NSS library. Now, we should organize binding with the static `bindSecure` method of `HttpServer` to create an HTTPS server. Let's run it and open the following URL in the Dartium web browser:

`https://localhost:8443/index.html`

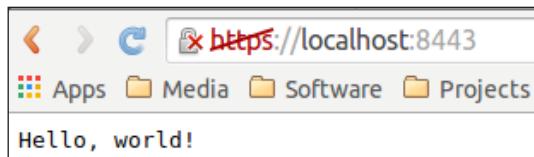
All the magic, such as TLS handshaking, keys, and message exchange, happens behind the scenes. As our server's certificate is self-signed, a web browser informs us about that fact, as shown in the following screenshot:



Click on the **Certificate** information link to see the full certificate information, as shown in the following screenshot:



Now, close the warning message and click on the **Proceed anyway** button to see the result of the HTTP request:



We were successful in achieving the following goals:

- We generated a self-signed certificate and registered it in the CA database
- Dart's `HttpServer` accepts the self-signed certificate and works with it
- The web browser shows the self-signed certificate information
- Client-to-server communication is granted

Securing a client

We have prepared our server side to secure the communication, and now it's time to talk about the security of the client side of our web application. For all our content, we will start using secure communication with TLS and we will start updating our client side using cookies.

Attributes of cookies

A cookie has two special attributes: `Secure` and `HttpOnly`. The `Secure` attribute of a cookie allows it to be sent only to the TLS connection. The other attribute, `HttpOnly`, marks the cookie that is accessible only via HTTP or HTTPS connections. Mark both of them as `true` and this small improvement in cookies prevents the web browser from sending a cookie via an insecure connection. With each request sent, the cookies are accompanied to follow the server inside headers. Let's check what we can improve in other headers.

HTTP Strict Transport Security

The well-known SSL man-in-the-middle attacks can be safely fixed with a **HTTP Strict Transport Security (HSTS)** header sent from the server via the HTTP response header, which obliges the web browser to interact with the server through a secure HTTPS connection. The code is as follows:

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

The server must specify the `max-age` option in seconds; this is the time for which the pages should be served with HTTPS. In our example, this value is equal to 365 days. The `includeSubDomains` option is optional and tells the web browser that all subdomains must be served with a secure connection as well. This header is supported in the following browsers:

- Firefox 4
- Chromium and Google Chrome 4.0.211.0
- Safari 7
- Opera 12
- Internet Explorer in the next major release after IE 11

Content Security Policy

The web application security model is based on the same-origin policy principle. The origin is a combination of schema, hostname, and port number. The policy permits us to download and run scripts from the same origin. As time has shown, this policy may be broken very easily and quickly with **Cross Site Scripting (XSS)** or data injection attacks. **Content Security Policy (CSP)** is an added layer of security. It allows the web server to define the origin of each resource by securing the website and mitigates and reports on XSS attacks. Blocking all the inline scripts and styles can prevent the execution of code injected in comments or posts. CSP is backward compatible, so web browsers that don't support it still work using the standard same-origin policy.

The web browser assumes that all origins are allowed if a directive is not set. CSP can be set via an HTTP response header on a server or an HTML meta tag on a web page, as shown in the following code:

```
Content-Security-Policy: policy
```

The `policy` string is the one that contains the policy directives describing CSP with semicolon separation as source of whitelists.



Not all web browsers support HTML meta elements to configure a policy.



The policy should include the `default-src` or `script-src` directives. This has the following advantages:

- This restricts inline scripts from running
- This blocks the use of the `eval` function
- This restricts inline styles from being applied from a `style` element or the `style` attribute of an element

The inline JavaScript code includes the `eval` function; hence, the JavaScript URLs will not be executed. You need to slightly change your mind about development with CSP. Here is an example of restricting all the content that comes only from the site's own domain and subdomains:

```
Content-Security-Policy: default-src 'self' *.mydomain.com
```

The following example shows how to restrict all the content from being loaded via a secure connection:

```
Content-Security-Policy: default-src https://ibank.mydomain.com
```

In the following example, we will allow all the assets to be loaded from our site and scripts from the Google API server:

```
Content-Security-Policy: default-src: 'self'; script-src: https://apis.google.com;
```

We created the `csp` project to see how CSP works. The server-side code is a slightly modified version of the server code from the previous topic, and it includes the `route` library, as shown in the following code:

```
import 'dart:io';
import 'dart:async';
import 'package:route/server.dart';
import 'urls.dart';
```

Security Aspects

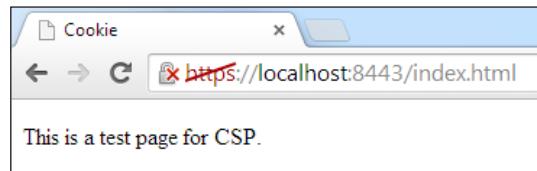
```
import 'files.dart';

main() {
  var pkcertDB = Platform.script.resolve('pkcert').toFilePath();
  SecureSocket
    .initialize(database: pkcertDB, password: 'changeit');

  HttpServer
    .bindSecure(InternetAddress.ANY_IP_V6, 8443,
      certificateName: 'localhost_cert')
    .then((server) {
      new Router(server)
        ..filter(allUrls, filter)
        ..serve(allUrls).listen(serveDirectory('', as: '/'))
        ..defaultStream.listen(send404);
    });
}

Future<bool> filter(HttpRequest request) {
  HttpResponse response = request.response;
  response.headers.add("Content-Security-Policy",
    "default-src 'self'; style-src 'self'");
  return new Future.value(true);
}
```

We now use a filter method of the Route class to intercept each request and inject the Content-Security-Policy header in response, as shown in the following screenshot:



From the content of our header, it should be clear that all the scripts and styles from our website are permitted. Let's imagine a use case where you need to add a Google +1 button to your web application to allow users to recommend the content to their circles and drive traffic to your website, so simply include a +1 button on the web page via a JavaScript resource and add a +1 button tag. The script must be loaded using the HTTPS protocol. Here is the code of the changed web page:

```
<!DOCTYPE html>
<html>
```

```

<head>
    <meta charset="utf-8">      <meta name="viewport"
        content="width=device-width, initial-scale=1">
    <title>Cookie</title>
    <script type="text/javascript"
        src="https://apis.google.com/js/plusone.js"></script>
    <link type="text/css" href="index.css">
</head>
<body>
    <p>This is a test page for CSP. <g:plusone></g:plusone></p>
</body>
</html>

```

Let's run the server and open the modified web page in Dartium. In a moment, you will receive the following CSP violation message about loading an untrusted script:

① Refused to load the script '<https://apis.google.com/js/plusone.js>' because it violates the following Content Security Policy directive: "default-src 'self'". Note that 'script-src' was not explicitly set, so 'default-src' is used as a fallback.
[index.html:1](#)

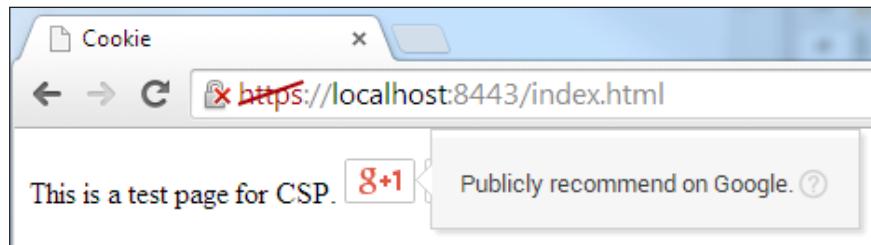
To make the **+1** button work, you need to add different policies to the server code to allow trusted resources on your web page, as shown in the following code:

```

Future<bool> filter(HTTPRequest request) {
    HttpResponse response = request.response;
    response.headers.add("Content-Security-Policy",
        "default-src 'self';" +
        "style-src 'self' 'unsafe-inline';" +
        "script-src 'self' https://apis.google.com;" +
        "frame-src https://*.google.com;" +
        "img-src https://*.gstatic.com;" +
    );
    return new Future.value(true);
}

```

The result of the preceding code is as follows:



CSP is very flexible and useful when it is used properly. This header is supported by the following browsers:

- Chrome 25 (from v14 with the prefix `webkit`)
- Firefox 23 (from v4 with the prefix `moz`)
- Safari 7 (from v5 with the prefix `webkit`)
- Opera 15
- Internet Explorer 10 only supports the `sandbox` directive with the prefix `ms`

Cross Origin Resource Sharing versus JSON with padding

JSON with padding (JSONP) is a client-side technique used to request data from a server in a different domain. This is possible because web browsers do not enforce the same-origin policy on the HTML `script` tag. The parameters of the JSONP request are passed as arguments to a script. The format of a JSONP result is different from the format of JSON, so the server must know how to respond to it. JSONP supports only the `GET` request method and accepts the callback function as the recipient of data, as shown in the following code:

```
<script src="http://my.com/data?format=jsonp&callback=cb"></script>
```

A web browser will call a `cb` function at the end of the request. With this script, we will get the JavaScript code and the web browser will run it as a normal script file. This could be a big risk because the server from which we are getting this script could be compromised and easily cause an XSS attack. **Cross-origin resource sharing (CORS)** can be used as a modern alternative to JSONP, which allows cross-domain communication from the web browser. As opposed to JSONP, CORS supports all the HTTP methods and allows you to do the following tasks:

- Make an AJAX request, but in a cross-site manner
- Load web fonts for use in `@font-face` within CSS
- Load WebGL textures
- Load images drawn on a canvas with the help of the `drawImage` method

CORS headers must be returned in the header of the requested web server. To initiate a cross-origin request, we need to add new HTTP headers that allow the web browser to communicate freely with the API on another domain. The cors project contains two servers. The first one, located in the `server.dart` file is a web server listening to the secure connection on port 8443 from our previous topic, as shown in the following code:

```
import 'dart:io';
import 'dart:async';

import 'package:route/server.dart';

import 'urls.dart';
import 'files.dart';

main() {
    var pkcertDB = Platform.script.resolve('pkcert').toFilePath();
    SecureSocket
        .initialize(database: pkcertDB, password: 'changeit');

    HttpServer
        .bindSecure(InternetAddress.ANY_IP_V6, 8443,
            certificateName: 'localhost_cert')
        .then((server) {
            new Router(server)
                ..filter(allUrls, filter)
                ..serve(allUrls).listen(serveDirectory('', as: '/'))
                ..defaultStream.listen(send404);
        });
}

Future<bool> filter(HttpRequest request) {
    HttpResponse response = request.response;
    response.headers.add("Content-Security-Policy",
        "default-src 'self';" +
        "style-src 'self' 'unsafe-inline';" +
        "script-src 'self' https://apis.google.com;" +
        "frame-src https://*.google.com;" +
        "img-src https://*.gstatic.com;"
    );
    return new Future.value(true);
}
```

Security Aspects

The second server, located in cors_server.dart, is the CORS web server listening on port 8080 and is not using HTTPS, as shown in the following code:

```
import 'dart:io';
import 'dart:async';

import 'package:route/server.dart';

import 'files.dart';

final allUrls = new RegExp('/(.*)');
final productUrl = new UrlPattern('/product');

main() {
    HttpServer
        .bind(InternetAddress.ANY_IP_V6, 8080)
        .then((server) {
            new Router(server)
                ..filter(allUrls, filter)
                ..serve(productUrl).listen(serverProduct)
                ..defaultStream.listen(send404);
        });
}

Future<bool> filter(HttpRequest request) {
    return new Future.value(true);
}

serverProduct(HttpRequest request) {
    return serveFile('products.json')(request);
}
```

The function filter in the second web server intends to set the header with CORS and allows any client to make cross-domain requests to this server. Our client will now look like the following code:

```
import 'dart:html';
import 'dart:convert';

void main() {
    onloadHandler();
}

onloadHandler() {
    var xhr = new HttpRequest();
```

```

xhr.open('GET', 'http://localhost:8080/product', async:true);
xhr.onLoad.listen((e) {
  Map repos = JSON.decode(xhr.response);
  var reposHTML = "";
  for (int i = 0; i < repos["repositories"].length; i++) {
    reposHTML += "<p>" +
      repos["repositories"][i]["name"] + "<br>" +
      repos["repositories"][i]["description"] + "</p>";
  }
  document.getElementById("allRepos").innerHTML(reposHTML);
}).onError((e) {
  print('error making the request. ${e.toString()}');
});
xhr.send();
}

```

The client code makes a cross-domain request and prints the markup with the result. Let's run both the servers and open our web page in Dartium on the address `https://localhost:8443/index.html`. It immediately comes with the cross-domain violation exception, as shown in the following screenshot:

✖ Refused to connect to 'http://localhost:8080/product' because it violates the following Content Security Policy directive: "default-src 'self'". Note that 'connect-src' was not explicitly set, so 'default-src' is used as a fallback. index.html:1

Let's add `connect-src` in the following server code for a quick fix:

```

Future<bool> filter(HttpServletRequest request) {
  HttpServletResponse response = request.response;
  response.headers.add("Content-Security-Policy",
    "default-src 'self';" +
    "style-src 'self' 'unsafe-inline';" +
    "script-src 'self' https://apis.google.com;" +
    "frame-src https://*.google.com;" +
    "img-src https://*.gstatic.com;" +
    "connect-src http://localhost:8080/product"
  );
  return new Future.value(true);
}

```

Restart the server and refresh the web page, and you will get the following exception:

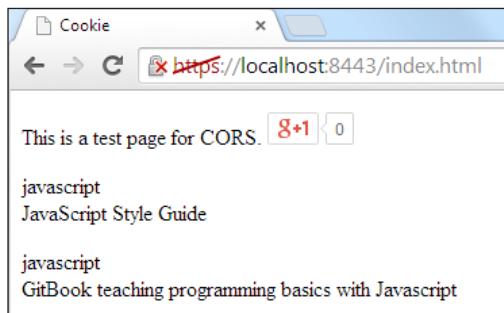
✖ XMLHttpRequest cannot load http://localhost:8080/product. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'https://localhost:8443' is therefore not allowed access. index.html:1

Security Aspects

Our request cannot pass the border of origins, so we will change the filter method in the CORS web server, as follows:

```
Future<bool> filter(HttpServletRequest request) {  
    HttpResponse response = request.response;  
    response.headers.add("Access-Control-Allow-Origin", "*");  
    return new Future.value(true);  
}
```

The preceding code will give the following result:



This was a simple demonstration of how we can use CORS on data provided by the web server. Dartium sends an initial request to the CORS server with an Origin HTTP header that matches the origin of our web page, as shown in the following screenshot:

Request Headers	
Accept:	/*
Accept-Encoding:	gzip, deflate, sdch
Accept-Language:	en-US,en;q=0.8
Cache-Control:	no-cache
Connection:	keep-alive
Host:	localhost:8080
Origin:	https://localhost:8443

We intend to specify Access-Control-Allow-Origin in the CORS server to allow all domains and a server-sent response with an asterisk symbol (*), as shown in the following screenshot:

Response Headers	
access-control-allow-origin:	*
content-length:	3360
content-type:	null

This pattern is widely used to organize accessible resources by anyone who knows the secret. The asterisk symbol is special as it tells the web browser that it doesn't allow requests without the following credentials:

- HTTP authentication
- Client-side SSL certificates
- Cookies

In order to include the credentials from the preceding list, you can use the other CORS header as follows:

```
Future<bool> filter(HttpServletRequest request) {
    HttpServletResponse response = request.response;
    response.headers.add("Access-Control-Allow-Origin", "*");
    response.headers.add("Access-Control-Allow-Credentials",
        "true");
    return new Future.value(true);
}
```

It works in conjunction with the credentials on `HttpRequest`, as shown in the following code:

```
var xhr = new HttpRequest();
xhr.open('GET', 'http://localhost:8080/product');
xhr.withCredentials = true;
...
```

It will also include any cookies from a remote domain in the request.



Do not set the `Access-Control-Allow-Credentials` header if you don't want to include cookies in the CORS request.

The CORS server can set any header, but the `getResponseBody` method of the `HttpRequest` class can read only the following simple headers:

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

Security Aspects

If you need access to other headers, you must expose them via the Access-Control-Expose-Headers header as follows:

```
Future<bool> filter(HttpServletRequest request) {
    HttpServletResponse response = request.response;
    response.headers.add("Access-Control-Allow-Origin", "*");
    response.headers.add("Access-Control-Expose-Headers",
        "session-id");
    response.headers.add("session-id", "123456");
    return new Future.value(true);
}
```

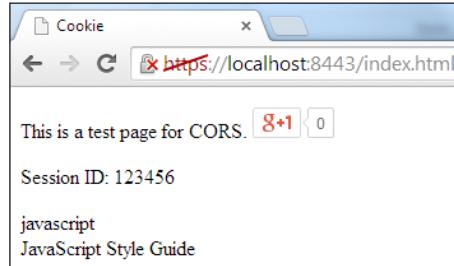
We added a sessionId span element to the web page as follows:

```
<p>This is a test page for CORS. <g:plusone></g:plusone></p>
Session ID: <span id="sessionId"></span>
<div id="allRepos"></div>
```

The following web page source code was updated as well:

```
var xhr = new HttpRequest();
xhr.open('GET', 'http://localhost:8080/product');
xhr.onLoad.listen((e) {
    var sessionId = xhr.getResponseHeader("session-id");
    document.getElementById("sessionId").text = sessionId;
```

Now, restart the CORS server and reload the web page to get the following result:



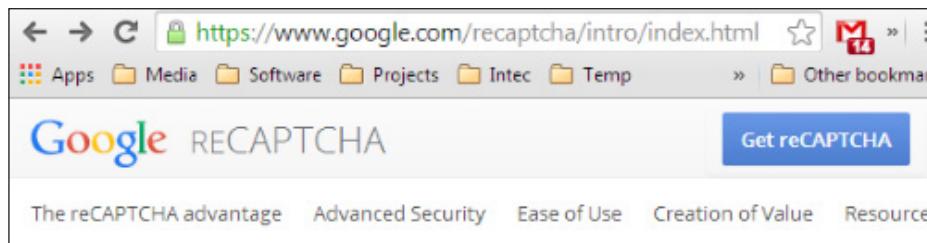
CORS is supported across the following well-known web browsers:

- Chrome 3
- Firefox 3.5
- Opera 12
- Safari 4
- Internet Explorer 8

CAPTCHA

Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) is a program whose main purpose is differentiating a human from a machine. Actually, CAPTCHA is a reverse Turing test because it is administrated by a computer. It is a barrier that prevents bots from using web services or collecting certain types of sensitive information. One of the ways of using CAPTCHA in Dart is using the free service reCAPTCHA of Google, so I decided to create a project with a sensible name, `captcha`, that contains one web page for user registration. We can follow several simple steps to add the reCAPTCHA solution into our project, but first we need to sign up for the API keys for our website with the following steps:

1. Visit <https://www.google.com/recaptcha/intro/index.html> and click on the **Get reCAPTCHA** button, as shown in the following screenshot:



2. On the **Get reCAPTCHA** page, click on the **Sign up Now!** button and type your web server name in the **Domain** field, as shown in the following screenshot:

HOME Get reCAPTCHA My account Protect your email Resources: docs & plugins	Domain <input type="text"/> <small>Enter domain addresses with comma as separator. If more than one domain is specified they will all share the same keys. e.g. recaptcha.net, example.com</small> Tips <small>Your reCAPTCHA key is restricted to the specified domains, and any subdomains for additional security. A key for foo.com works on test.foo.com.</small> <input type="button" value="CREATE"/>
--	---

You can type as many domain names as you need; just separate them with commas. You can also use `localhost` or `127.0.0.1` as the name of your server, because all the API keys work on it and you can develop and test your solution on your local machine.

Security Aspects

3. Click on the **CREATE** button to create new API key. The server move you to the list of your domains, as shown in the following screenshot:

The screenshot shows a web application interface for managing reCAPTCHA sites. At the top right is a red button labeled '+ Add a New Site'. Below it, a message says 'If you have many sites, you may want to export the keys as a CSV file.' On the left, there's a sidebar with links: 'HOME', 'Get reCAPTCHA', 'My account' (which is highlighted in red), 'Protect your email', and 'Resources: docs & plugins'. The main area is titled 'Sites you Administer' and contains a single entry: 'localhost'.

4. Choose your domain to see the following details:

The screenshot shows the details for the 'localhost' domain. It includes:

- Domain Name:** localhost
reCAPTCHA will work on these domains and subdomains.
- Public Key:** 6Lc8a_kSAAAAABk-6joEQu_wurhopTGt4xCPndnX
Use this in the JavaScript code that is served to your users.
- Private Key:** 6Lc8a_kSAAAAB49Z1belTOeM2e3SDmPG4ZvXVNL
Use this when communicating between your server and our server. Be sure to keep it a secret.

A blue link at the bottom says 'Delete these keys'.

There are public and private keys that we will use in our solution. To integrate the reCAPTCHA solution in the captcha project, we used the small library `recaptcha` created by me and which is available on the <https://pub.dartlang.org/> server. So, we can add it in the `pubspec.yaml` file under the dependencies packages. Open the `captcha` project in Dart Editor and navigate to `index.html`, which is shown in the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Registration</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
```

```
<link type="text/css" href="index.css">
</head>
<body>
    <H1>Registration form with CAPTCHA</H1>
    <form name="captcha_form" method="post" action="/register">
        <label for="username">Username:</label>
        <input type="text" name="username"><br>
        <label for="password">Password:</label>
        <input type="password" name="password">
        <script type="text/javascript"
            src="http://www.google.com/recaptcha/api/challenge?k=6Lc8a_
            kSAAAAA
            Bk-6joEQu_wurhopTGT4xCPndnX">
        </script>
        <noscript>
            <iframe src="http://www.google.com/recaptcha/api/
            noscript?k=6Lc8a_kSAAAAAB
            k-6joEQu_wurhopTGT4xCPndnX"
                height="300" width="500" frameborder="0"></iframe><br>
            <textarea name="recaptcha_challenge_field" rows="3"
                cols="40"></textarea>
            <input type="hidden" name="recaptcha_response_field"
                value="manual_challenge">
        </noscript>
        <button type="submit" value="Submit">Submit</button>
    </form>
</body>
</html>
```

Copy and paste the public key of the domain registered on reCAPTCHA as the parameter for JavaScript and the parameter for the source of the iframe tag. You need to change these values for your public key. Now, let's open the `server.dart` file and move to line 10 where we created an instance of the `ReCaptcha` class. Again, copy and paste a pair of private and public keys of your domain here so that the `class` instance can pass them via the free service reCAPTCHA on Google in order to ensure that the sender is correct and has a registered domain, as shown in the following code:

```
...
final ReCaptcha reCaptcha = new ReCaptcha(
    '6Lc8a_kSAAAAABk-6joEQu_wurhopTGT4xCPndnX', // public key
    '6Lc8a_kSAAAAAB49Z1belTOeM2e3SDmPG4ZvXVNL'); // private key
```

Then, create a map of the error code and human-readable text as shown in the following code:

```
final Map MESSAGES = {
    'invalid-site-private-key':'Incorrect private key',
    'invalid-request-cookie':'The challenge parameter of the verify
script was incorrect',
    'incorrect-captcha-sol':'The CAPTCHA solution was incorrect',
    'captcha-timeout':'The solution was received after the CAPTCHA
timed out',
    'recaptcha-not-reachable':"Unknown error in CAPTCHA"};
```

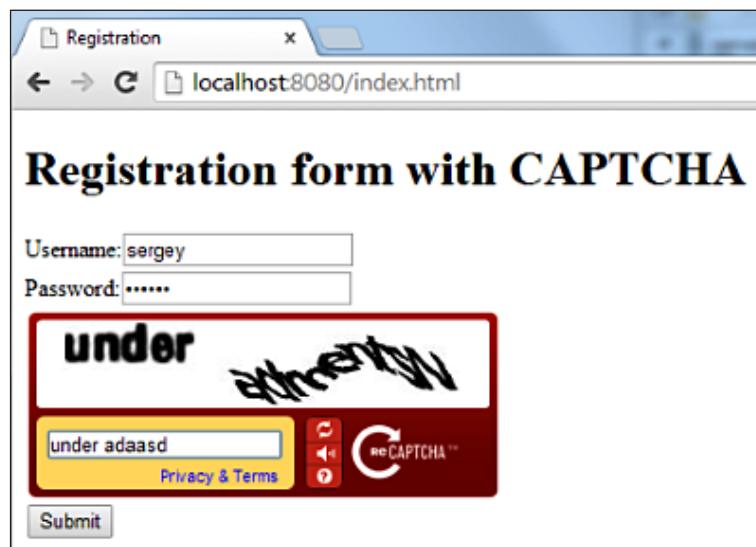
Then, read the POST method parameters in the `serverRegister` function and convert them into a map to easily access them later. All the parameters follow the reCAPTCHA verification via the `checkAnswer` method of `ReCaptcha`, as shown in the following code:

```
serveRegister(HttpServletRequest request) {
    HttpServletResponse response = request.response;
    request.listen((List<int> buffer) {
        String strBuffer = new String.fromCharCodes(buffer);
        Map data = postToMap(strBuffer);
        //
        String userName = data.containsKey('username') ?
            data['username'] : '';
        String password = data.containsKey('password') ?
            data['password'] : '';
        String cptChallenge =
            data.containsKey('recaptcha_challenge_field') ?
            data['recaptcha_challenge_field'] : '';
        String cptResponse =
            data.containsKey('recaptcha_response_field') ?
            data['recaptcha_response_field'] : '';
        reCaptcha.checkAnswer(request.uri.host, cptChallenge,
cptResponse).then((ReCaptchaResponse cptResponse) {
            response.statusCode = HttpStatus.OK;
            setCORSHeader(response);
            if (cptResponse.valid) {
                response.write("Registration success.");
            } else {
                response.write(MESSAGES[cptResponse.errorCode]);
            }
            response.close();
        });
    });
}
```

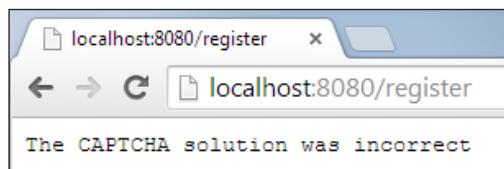
In the `setCORSHeader` function, add the CORS headers to allow the POST requests from the cross-origin web resources as follows:

```
setCORSHeader(HttpResponse response) {
    response.headers.add('Access-Control-Allow-Origin', '*');
    response.headers.add('Access-Control-Allow-Methods', 'POST,
OPTIONS');
    response.headers.add('Access-Control-Allow-Headers',
'Origin, X-Requested-With, Content-Type, Accept');
}
```

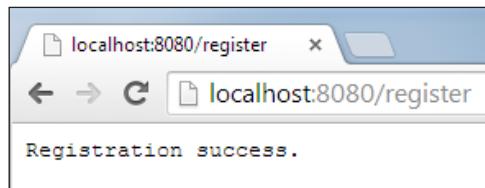
Let's run the server and open `http://localhost:8080/index.html` in Dartium to get the following result:



Try to input a wrong CAPTCHA solution and submit the form. After submitting it, you will see the following error message:



Let's go back quickly and type the correct CAPTCHA solution. After submitting, you will see the following success message:



Security best practices

It's time to discuss the best security practices, without which this story would not be complete:

- **Do not retain the password:** The HTTP basic authentication is deprecated, so use other techniques such as OAuth to make a more secure application following standards. Use safe OAuth tokens instead of passwords.
- **Perform the input validation:** You should always sanitize all input data. You need to check string length, validate file types, and check the minimum and maximum values to be sure that all the data sent to the server via the POST request is in the proper format and length.
- **Filter input and sanitize output:** You should always filter all the data that comes from the client to the web server and sanitize all the data coming back to the client.
- **Use a secure connection:** Use the TLS certificate to organize a secure connection between the web browser and server to provide all REST APIs or AJAX requests over TLS. TLS in conjunction with OAuth is a safe and suggested solution.
- **Do not expose the debug information:** Don't forget to switch off the debug logs because they can contain sensitive information.
- **Test boundaries:** Your tests must check all the possible positive and negative cases and scenarios.
- **Hide the server information:** Don't display the server information on any server-generated documents as this will allow hackers to select the right kind of hack from the hacks that are either available freely on the web or developed by hackers themselves.

Summary

In this chapter, you learned how to create a TLS certificate with NSS tools. You saw that the certificate can be quickly embedded into a Dart web server without extra effort on the developer's part.

We discovered how to secure the client side with the `Secure` and `HttpOnly` special attributes of cookies to prevent the web browser from sending cookies via an insecure connection.

We used HSTS to prevent SSL man-in-the-middle attacks. We applied CSP to make sure that only allowed content can be loaded and used by the web browser. We also used CORS to specify what resources from our web server can be shared and why that solution is much better than JSONP. Finally, we embedded the CAPTCHA solution based on the free service reCAPTCHA from Google in our project.

Module 3

Dart Cookbook

*Over 110 incredibly effective, useful, and hands-on recipes to design
Dart web client and server applications*

1

Working with Dart Tools

In this chapter, we will cover the following recipes:

- ▶ Configuring the Dart environment
- ▶ Setting up the checked and production modes
- ▶ Rapid Dart Editor troubleshooting
- ▶ Hosting your own private pub mirror
- ▶ Using Sublime Text 2 as an IDE
- ▶ Compiling your app to JavaScript
- ▶ Debugging your app in JavaScript for Chrome
- ▶ Using the command-line tools
- ▶ Solving problems when pub get fails
- ▶ Shrinking the size of your app
- ▶ Making a system call
- ▶ Using snapshotting
- ▶ Getting information from the operating system

Introduction

This chapter is about increasing our mastery of the Dart platform. Dart is Google's new language for the modern web, web clients, as well as server applications. Compared to JavaScript, Dart is a higher-level language so it will yield better productivity. Moreover, it delivers increased performance. To tame all that power, we need a good working environment, which is precisely what Dart Editor provides. Dart Editor is quite a comprehensive environment in its own right and it is worthwhile to know the more advanced and hidden features it exposes. Some functionalities are only available in the command-line tools, so we must discuss these as well.

Configuring the Dart environment

This recipe will help customize the Dart environment according to our requirements. Here, we configure the following:

- ▶ Defining a DART_SDK environment variable
- ▶ Making dart-sdk\bin available for the execution of the Dart command-line tools

Getting ready

We assume that you have a working Dart environment installed on your machine. If not, go to <https://www.dartlang.org/tools/download.html> and choose **Option 1** for your platform, which is the complete bundle. Downloading and uncompressed it will produce a folder named dart, which will contain everything you need. Put this in a directory of your choice. This could be anything, but for convenience keep it short, such as d:\dart on Windows or ~/dart on Linux. On OS X, you can just drop the directory in the App folder.

How to do it...

1. Create a DART_SDK environment variable that contains the path to the dart-sdk folder. On Windows, create and set DART_SDK to d:\dart\dart-sdk or <your-dart-sdk-path>\dart-sdk when using a dart from another folder (if you need more information on how to do this, refer to <http://www.c-sharpcorner.com/UploadFile/6cde20/use-of-environment-variable-in-windows-8/>). On Linux, add this to your configuration file .bashrc and/or .profile using the export DART_SDK=~/dart/dart-sdk code. On OS X, export DART_SDK=/Applications/dart/dart-sdk or in general export DART_SDK=/path/to/dart-sdk.
2. The installation directory has a subfolder dart-sdk\bin, which contains the command-line tools. Add this subfolder to the path of your environment. On Windows, add %DART_SDK%\bin instead to the front of the path (system environment) variable and click on **OK**. On Linux or OS X, add export PATH=\$PATH:\$DART_SDK/bin to your configuration file.
3. Reset your environment configuration file or reboot your machine afterwards for the changes to take effect.

How it works...

Setting the `DART_SDK` environment variable, for example, enables plugins such as `dart-maven` to search for the Dart SDK (`dart-maven` is a plugin that provides integration for Google Dart into a maven-build process). If the OS of your machine knows the path where the Dart tools reside, you can start any of them (such as the Dart VM or `dartanalyzer`) anywhere in a terminal or command-line session.

Test the environment variable by typing `dart` in a terminal and press *Enter*. You should see the following help text:

Usage: `dart [<vm-flags>] <dart-script-file> [<dart-options>]`

Executes the Dart script passed as `<dart-script-file>`

Setting up the checked and production modes

When developing or maintaining, an app's execution speed is not so important, but information about the program's execution is. On the other hand, when the app is put in a customer environment to run, the requirements are nearly the opposite; speed is of utmost importance, and the less information the program reveals about itself, the better. That's why when an app runs in the Dart **Virtual Machine (VM)**, it can do so in two runtime modes:

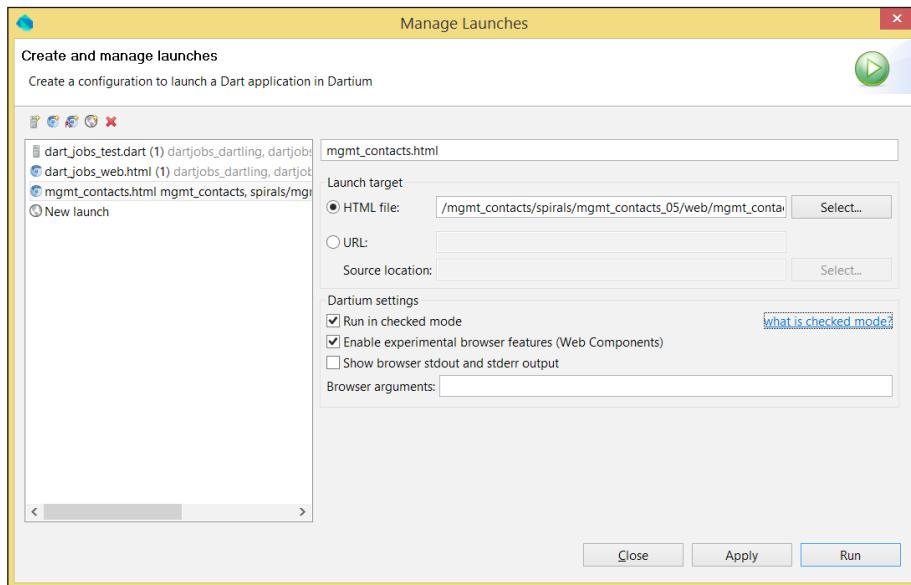
- ▶ **The Checked mode:** This is also known as the debug mode. The checked mode is used during development and gives you warnings and errors of possible bugs in the code.
- ▶ **The Production mode:** This is also known as the release mode. You deploy an app in the production mode when you want it to run as fast as possible, unhindered by code checks.

Getting ready

Open your app in Dart Editor and select the startup web page or Dart script, usually `web\index.html`.

How to do it...

- When working in Dart Editor, the checked mode is the default mode. If you want the production mode, open the **Run** menu and select **Manage Launches** (**Ctrl + Shift + M**). The **Manage Launches** window appears, as shown in the following screenshot:



The Manage Launches window

- Under **Dartium settings**, you will see the checkbox **Run in checked mode**. (If you have selected a Dart script, it will be under the header **VM settings**.) Uncheck this to run the script in the production mode. Next, click on **Apply** and then on **Close**, or on **Run** immediately. This setting will remain in place until you change it again.

Scripts that are started on the command line (or in a batch file) with the `dart` command run in the Dart VM and thus in the production mode. If you want to run the Dart VM in the checked mode, you have to explicitly state that with the following command:

```
dart -c script.dart or: dart --checked script.dart
```

You can start Dartium (this is Chromium with the Dart VM) directly by launching the Chrome executable from `dart\chromium`; by default, it runs Dart Editor in the production mode. If you would like to start Dartium in the checked mode, you can do this as follows:

- On Windows, in the `dart\chromium` folder, click on the `chrome` file
- On Linux, in the `~/dart/chromium` folder, open the `./chrome` file
- On OS X, open the `DART_FLAGS` folder and then open `path/Chromium.app`

Verify this setting by going to the following address in the Chrome browser that you just started `chromium://version`.

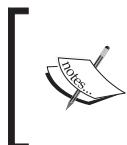
When a web app runs in the Dart VM in Chrome, it will run in the production mode, by default.

How it works...

In the checked mode, types are checked by calling assertions of the form `assert (var1 is T)` to make sure that `var1` is of type `T`. This happens whenever you perform assignments, pass parameters to a function, or return results from a function.

However, Dart is a dynamic language where types are optional. That's why the VM must, in the production mode, execute your code as if the type annotations (such as `int n`) do not exist; they are effectively thrown away. So at runtime, the following statement `int x = 1` is equivalent to `var x = 1`.

A binding `x` is created but the type annotation is not used.



Avoiding type checks makes the production mode a lot faster. Also, the VM uses the type inference to produce faster code; it observes the type of the value (here, 1) assigned to `x` and optimizes accordingly.

There's more...

With the checked mode, Dart helps you catch type errors during development. This is in contrast to the other dynamic languages, such as Python, Ruby, and JavaScript, where these are only caught during testing, or much worse, they provoke runtime exceptions. You can easily check whether your Dart app runs in the checked mode or not by calling the function `isCheckedMode()` from `main()` (see the script `test_checked_mode\bin\test_checked_mode.dart` in the Chapter 1 folder of the code bundle), as shown in the following code:

```
main() {  
    isCheckedMode();  
    // your code starts here  
}  
  
void isCheckedMode() {  
    try {  
        int n = '';  
        throw new Exception("Checked Mode is disabled!");  
    } on TypeError {
```

```
    print("Checked Mode is enabled!") ;
}
}
```

Downloading the example code



You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

The exception message will be shown in the browser console. Be sure to remove this call or comment it out before deploying it to the production mode; we don't want an exception at runtime!

See also

- ▶ The *Compiling your app to JavaScript* recipe of this chapter for how to enable the checked mode in the JavaScript version of the app
- ▶ The *Using the command-line tools* recipe of this chapter for other options

Rapid Dart Editor troubleshooting

Dart Editor is based upon the Eclipse Integrated Development Environment (IDE), so it needs the Java VM to run. Sometimes, problems can arise because of this; if this is the case, be sure to consult the Dart Editor Troubleshooting page on the Dart website at <https://www.dartlang.org/tools/editor/troubleshoot.html>.

Getting ready

Some of the JVM settings used by Dart Editor are stored in the `DartEditor.ini` file in the `dart` installation directory. This typically contains the following settings (on a Windows system):

```
-data
@user.home\ DartEditor
-vmargs
-d64
-Dosgi.requiredJavaVersion=1.6
-Dfile.encoding=UTF-8
-XX:MaxPermSize=128m
-Xms256m
-Xmx2000m
```



The line beneath `-data` will read `@user.home/.dartEditor` on a Linux system.



How to do it...

If you notice strange or unwanted behavior in the editor, deleting the `settings` folder pointed to by `-data` and its subfolders can restore things to normal. This folder can be found at different locations depending on the OS; the locations are as follows:

- ▶ On a Windows system, `C:\Users\{your username}\DartEditor`
- ▶ On a Linux system, `$HOME/.dartEditor`
- ▶ On an OS X system, `$HOME/Library/Application Support/DartEditor`

Deleting the `settings` folder doesn't harm your system because a new `settings` folder is created as soon as you reopen Dart Editor. You will have to reload your projects though. If you want to save the old settings, you can rename the folder instead of just deleting it; this way, you can revert to the old settings if you ever want to.

How it works...

The settings for data points to the `DartEditor` folder are in the users home directory, which contains various settings (the metadata) for the editor. Clearing all the settings removes the metadata the editor uses.

There's more...

The `-d64` or `-d32` value specifies the bit width necessary for the JVM. You can check these settings for your installation by issuing the command `java -version` in a terminal session, whose output will be as follows:

java version "1.7.0_51"

Java(TM) SE Runtime Environment (build 1.7.0_51-b13)

Java HotSpot(TM) 64-Bit Server VM (build 24.51-b03, mixed mode)

If this does not correspond with the `-d` setting, make sure that your downloaded Dart Editor and the installed JVM have the same bit width, by downloading a JVM for your bit width.



If you work with many Dart projects and/or large files, the memory consumption of the JVM will grow accordingly and your editor will become very slow and unresponsive.

Working within a 32-bit environment will pretty much limit you to 1GB memory consumption, so if you see this behavior, it is recommended to switch to a 64-bit system (Dart Editor and JVM). You can then also set the value of the `-Xmx` parameter (which is by default set to 2000m = 2 GB) to a higher setting, according to the amount of memory you have installed. This will visibly improve the loading and working speed of your editor!

If your JVM is not installed in the default location, you can add the following line to the `.ini` file in the line before `-vmargs`:

```
-vm  
/full/path/to/java
```

If you face a problem, it might be solved by upgrading Dart SDK and the Dart Editor to the latest version. In the Dart Editor menu, select **Help** and then **About Dart Editor**. If a new version is available, this will automatically download, and when done, click on **Apply the update**.

Hosting your own private pub mirror

Another possibility for when the pub repository is not reachable (because you have no Internet access or work behind a very strict firewall) is to host your own private pub mirror.

How to do it...

Follow these steps to host your own private pub mirror:

1. You need a server that speaks to the pub's HTTP API. Documentation on that standalone API does not yet exist, but the main pub server running at `pub.dartlang.org` is open source with its code living at <https://github.com/dart-lang/pub-dartlang>. To run the server locally, go through these steps:
 1. Install the App Engine SDK for Python.
 2. Verify that its path is in `$PATH`.
 3. Install the pip installation file, `beautifulsoup4`, and `pycrypto` `webtest` packages.
 4. From the top-level directory, run this command to start the pub server `dev_appserver.py app`.
 5. Verify that it works in your browser with `http://localhost:8080/`.

2. You need to set a `PUB_HOSTED_URL` environment variable to point to the URL of your mirror server, so that the pub will look there to download the hosted dependencies, for example, `PUB_HOSTED_URL = http://me:mypassword@127.0.0.1:8042`.
3. Manually upload the packages you need to your server, visit `http://localhost:8080/admin` (sign in as an administrator), go to the **Private Key** tab, and enter any string into the private key field.

How it works...

The server from <https://pub.dartlang.org/> is written in Python and is made to run on Google App Engine, but it can be run from an Intranet as well.

Using Sublime Text 2 as an IDE

Dart Editor is a great environment, but Sublime Text also has many functionalities and can be used with many other languages, making it the preferred editor for many developers.

Getting ready

You can download Sublime Text free of cost for evaluation, however, for continued use, a license must be purchased from <http://www.sublimetext.com/>.

Tim Armstrong from Google developed a Dart plugin for Sublime Text, which can be downloaded from GitHub at <https://github.com/dart-lang/dart-sublime-bundle>, or you can find it in the code download with this book. The easiest way to get started is to install the Package Control plugin first by following the instructions at <https://sublime.wbond.net/installation#st2>.

How to do it...

In Sublime Text, press `Ctrl + Shift + P` (Windows or Linux) or `Cmd + Shift + P` (OS X; this goes for all the following commands), click on **Install Package** to choose that option, and then click and choose **Dart** to install the plugin. Any Dart file you then open shows the highlighted syntax, matching brackets, and so on.

Also, click on **Menu Preferences, Settings**, and then on **User** and add the path to your `dart-sdk` as the first line in this JSON file:

```
{  
  "dart sdk_path": "path\to\dart-sdk",  
  ...  
}
```

If you want to manually install this plugin, copy the contents of the `dart-sublime-bundle-master` folder to a new directory named `Dart` in the Sublime packages directory. This directory has different locations on different OS. They are as follows:

- ▶ On Windows, this will likely be found at `C:\Users\{your username}\AppData\Roaming\Sublime Text 2\Packages`
- ▶ On Linux, this will likely be found at `$HOME/Sublime Text 2/Pristine Packages`
- ▶ On OSX, this will likely be found at `~/Library/Application Support/Sublime Text 2/Packages`

How it works...

The plugin has a number of code snippets to facilitate working with Dart, for example, typing `lib` expands the `library` statement. Other snippets include `imp` for import, `class` for a class template, `method` for a method template, and `main` for a `main()` function. Typing a snippet in the pop-up window after pressing `Ctrl + SHIFT + P` lets you see a list of all the snippets. Use `Ctrl + /` to (un)comment the selected code text.

The plugin has also made a build system for you. `Ctrl + B` will invoke the `dartanalyzer` and then compile the Dart code to JavaScript with the `dart2js` compiler, as shown in the following screenshot. Editing and saving a `pubspec.yaml` file will automatically invoke the `pub get` command.

A screenshot of Sublime Text 2 showing a Dart file (`test.dart`) and a terminal window. The Dart file contains the following code:

```
library name;
import 'dart:math';

class name {
  void myMethod(one, two) {
    return 'hello world';
  }
}

main() {
  print('In main!');
}
```

The terminal window shows the output of running `dart2js.bat` and writing a file. The output is:

```
Running dart2js.bat --minify -oF:\Dartiverse\ADartCookbook\book\Chapter 1 - Working with the tools
of the Dart platform\code\test.dart.js F:\Dartiverse\ADartCookbook\book\Chapter 1 - Working with the
tools of the Dart platform\code\test.dart
Writing file /F:Dartiverse/ADartCookbook/book/Chapter 1 - Working with the tools of the Dart platform
/code/test.dart with encoding UTF-8 (atomic)
Running dart2js.bat --minify -oF:\Dartiverse\ADartCookbook\book\Chapter 1 - Working with the tools
of the Dart platform\code\test.dart.js F:\Dartiverse\ADartCookbook\book\Chapter 1 - Working with the
tools of the Dart platform\code\test.dart
```

Working in Sublime Text 2

See also

- ▶ Refer to the *Configuring the Dart environment* recipe for the path to the Dart SDK

Compiling your app to JavaScript

Deploying a Dart app in a browser means running it in a JavaScript engine, so the Dart code has to first be compiled to JavaScript. This is done through the `dart2js` tool, which is itself written in Dart and lives in the `bin` subfolder of `dart-sdk`. The tool is also nicely integrated in Dart Editor.

How to do it...

- ▶ Right-click on `.html` or the `.dart` file and select **Run as JavaScript**.
- ▶ Alternatively, you can right-click on the `pubspec.yaml` file and select **Pub Build** (generates JS) from the context menu. You can also click on the **Tools** menu while selecting the same file, and then on **Pub Build**.

How it works...

The first option invokes the `pub serve` command to start a local web server invoking `dart2js` along its way in the checked mode. However, the compiled `.dart.js` file is served from the memory by the internal development web server on `http://127.0.0.1:4031`. This is only good for development testing.

In the second option, the generated files are written to disk in a subfolder `build/web` of your app. In this way, you can copy this folder to a production web server and deploy your web app to run in all the modern web browsers (you only need to deploy the `.js` file, not the `.precompiled.js` file or the `.map` file). However, **Pub Build** in Dart Editor enables the checked mode by default; use the `pub build` command from a console for the production mode.

There's more...

The `dart2js` file can also be run from the command line, which is the preferred way to build non-web apps.

The command to compile the dart script to an output file `prorabbits.js` using `-o <file>` or `-out <file>` is `dart2js -o prorabbits.js prorabbits.dart`.

If you want to enable the checked mode in the JavaScript version, use the `-c` or `-checked` option such as `dart2js -c -o prorabbits.js prorabbits.dart`. The command `dart2js -vh` gives a detailed overview of all the options.

The `pub build` command, issued on a command line in the folder where `pubspec.yaml` is located, will do the same as in option 2 previously, but also apply the JavaScript shrinking step; the following is an example output for `app test_pub`:

```
f:\code\test_pub>pub build  
Loading source assets... (0.7s)  
Building test_pub... (0.3s)  
[Info from Dart2JS]:  
Compiling test_pub|web/test.dart...
```

[Info from Dart2JS]:	Took
0:00:01.770028 to compile test_pub web/test.dart.	Built 165
files to "build"	



You can minify both the JavaScript version and the Dart version of your app.

Producing more readable JavaScript code

To produce more readable JavaScript code (instead of the minified version of the production mode, refer to the *Shrinking the size of your app* recipe), use the command `pub build --mode=debug`, which is the default command in Dart Editor.

Alternatively, you can add the following transformers section to your app's `pubspec.yaml` file:

```
name: test_pub  
description: testing pub  
  
transformers:  
- $dart2js:  
  minify: false  
  checked: true  
  
dependencies:  
  js: any  
  
dev_dependencies:  
  unittest: any
```



For more information, refer to <https://www.dartlang.org/tools/pub/dart2js-transformer.html>.



Producing a single Dart file

The `dart2js` tool can also be used as Dart to Dart to create a single `.dart` file that contains everything you need for the app with this command:

```
dart2js --output-type=dart --minify -oapp.complete.dart app.dart
```

This takes the Dart app, tree shakes it, minifies it, and generates a single `.dart` file to deploy. The advantage is that it pulls in dependencies like third-party libraries and tree shakes it to eliminate the unused parts.

See also

You may be interested in the following recipes in this chapter:

- ▶ *Using the command-line tools*
- ▶ *Shrinking the size of your app*
- ▶ *Debugging your app in JavaScript for Chrome*

Debugging your app in JavaScript for Chrome

In this recipe, we will examine how to debug your app in the Chrome browser.

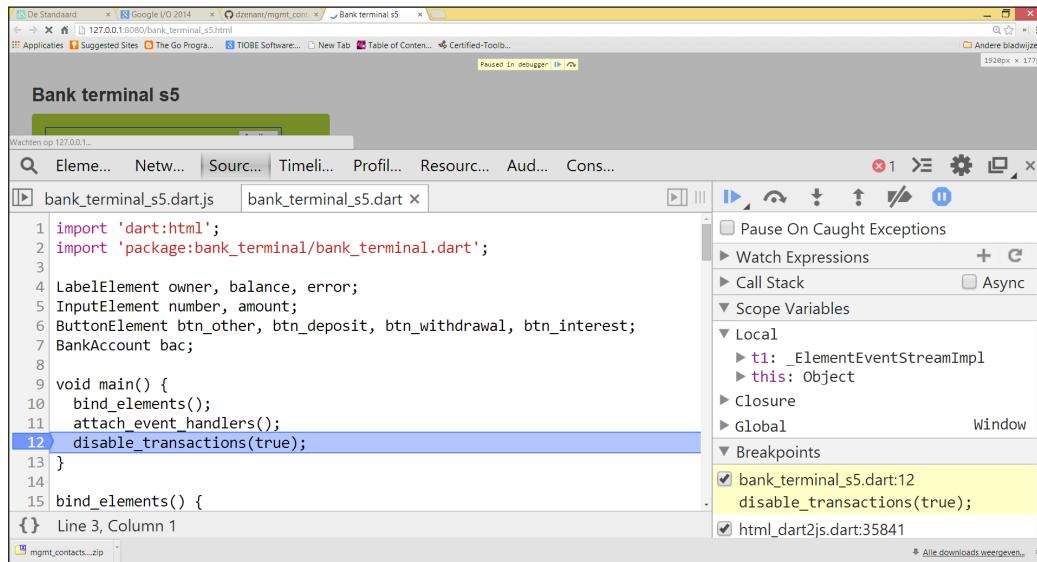
How to do it...

1. From the menu in the upper right-hand corner, select **Tools** and then **Developer Tools**.
2. Verify via **Settings** (which is the wheel icon in the upper right corner of the **Developer Tools** section) that the **Enable JavaScript source maps** option is turned on. Make sure that debugging is enabled, either on all the exceptions or only on uncaught exceptions.
3. Choose **Sources** in the **Developer Tools** menu, then press `Ctrl + O` to open a file browser and select the Dart script you wish to debug.



Clicking on the left margin before a line of code places a breakpoint, which is indicated by a fat blue arrow.

- Now reload the application and you will see that the execution stops at the breakpoint. On the right, you have a debug menu, which allows you to inspect scope variables, watch the call stack, and even create watch expressions, as shown in the following screenshot:



Debugging JS in Chrome

How it works...

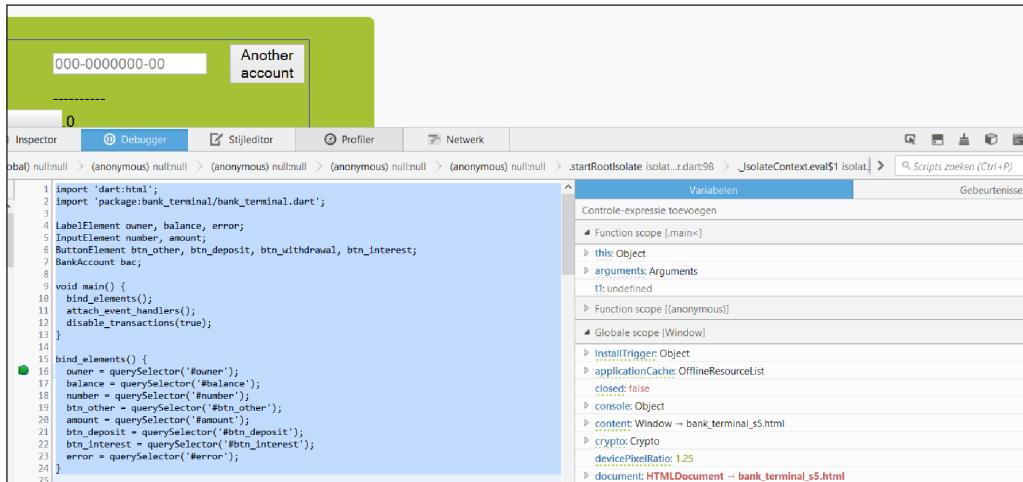
Chrome uses the source map file `<file>.js.map` generated while compiling the JavaScript code to map the Dart code to the JavaScript code in order to be able to debug it.

There's more...

In this recipe, we will examine how to debug your app in the Firefox browser.

Debugging your app in JavaScript for Firefox

In Firefox, the source maps feature is not yet implemented. Use **Shift + F2** to get the developer toolbar and the command line. In the top menu, you will see **Debugger**. Place a breakpoint and reload the file. Code execution then stops and you can inspect the value of the variables, as shown in the following screenshot:



Debugging JS in Firefox

Using the command-line tools

Some things can be done more easily on the command-line, or are simply not (yet) included in Dart Editor. These tools are found in `dart-sdk/bin`. They consist of the following:

- ▶ `dart`: The standalone Dart VM to run Dart command-line apps, such as server-side scripts and server apps
- ▶ `dartanalyzer`: This is used to check code statically
- ▶ `pub`: This is the package and repository manager
- ▶ `dartfmt`: This is the code formatting tool
- ▶ `docgen`: This is the documentation generator tool

How to do it...

1. For every tool, it might be useful to know or check its version. This is done with the `--version` option such as `dart --version` with a typical output of **Dart VM version: 1.3.0 (Tue Apr 08 09:06:23 2014) on "windows_ia32"**.
2. The `dart -v -h` option lists and discusses all the possible options of the VM. Many tools also take the `--package_root=<path>` or `-p=<path>` option to indicate where the packages used in the imports reside on the filesystem.
3. `dartanalyzer` is written in Java and works in Dart Editor whenever a project is imported or Dart code is changed; it is started `dartanalyzer prorabbits.dart` with output:

Analyzing `prorabbits.dart`...

No issues found (or possibly errors and hints to improve the code)

4. The previous output verifies that the code conforms to the language specification <https://www.dartlang.org/docs/spec/>, pub functionality is built into Dart Editor, but the tool can also be used from the command line (refer to `test_pub`). To fetch packages (for example, for the `test_pub` app), use the following command in the folder where `pubspec.yaml` lives, `pub get`, with a typical output as follows:

```
Resolving dependencies... (6.6s)
```

```
Got dependencies!
```

5. A packages folder is created with symlinks to the central package cache on your machine. The latest versions are downloaded and the package versions are registered in the `pubspec.lock` file, so that your app can only use these versions.
6. If you want to get a newer version of a package, use the `pub upgrade` command. You can use the `-v` and `--trace` options to produce a detailed output to verify its workings.



Always do a pub upgrade if the project you start working on already contains versions of packages!

7. The `dartfmt` tool is also a built in Dart Editor. Right-click on any Dart file and choose **Format** from the context menu. This applies transformations to the code so that it conforms to the Dart Style Guide, which can be seen at <https://www.dartlang.org/articles/style-guide/>. You can also use it from the command line, but then the default operation mode is cleaning up whitespace. Use the `-t` option to apply code transforms such as `dartfmt -t -w bank_terminal.dart`.

See also

- ▶ Solving problems when `pub get` fails
- ▶ Compiling your app to JavaScript (for `pub build`)
- ▶ Documenting your code from *Chapter 2, Structuring, testing, and deploying an application*
- ▶ Publishing your app to a pub (for `pub publishing`)
- ▶ Using snapshotting to start an app in Dart VM
- ▶ For additional information, refer to <https://www.dartlang.org/tools/>

Solving problems when pub get fails

The pub package manager is a complex tool with many functionalities, so it is not surprising that occasionally something goes wrong. The `pub get` command downloads all the libraries needed by your app, as specified in the `pubspec.yaml` file. Running `pub get` behind a proxy or firewall used to be a problem, but it was solved in the majority of cases. If this still haunts you, look at the corresponding section at <https://www.dartlang.org/tools/editor/troubleshoot.html>.

Getting ready

This recipe is especially useful when you encounter the following error in your Dart console while trying to open a project in Dart Editor during the `pub get` phase:

```
Pub install fails with 'Deletion failed'
```

How to do it...

First try this; right-click on your project and select **Close Folder**. Then, restart the editor and open your project again. In many cases, your project will load fine. If this does not work, try the `pub gun` command:

1. Delete the pub cache folder from `C:\Users\{your username}\AppData\Roaming\Pub`.
2. Delete all the packages folders in your project (also in subfolders).
3. Delete the `pubspec.lock` file in your project.
4. Run `pub get` again from a command line or select **Tools** in the Dart Editor menu, and then select **Pub Get**.

How it works...

The Pub\Cache subfolder contains all the packages that have been downloaded in your Dart environment. Your project contains symlinks to the projects in this cache, which sometimes go wrong, mostly on Windows. The pubspec.lock file keeps the downloaded projects constrained to certain versions; removing this constraint can also be helpful.

There's more...

Temporarily disabling the virus checker on your system can also help `pub get` to succeed when it fails with the virus checker on.

The following script by Richard Schmidt that downloads packages from the pub repository and unpacks it into your Dart cache may also prove to be helpful for this error, which can be found at <https://github.com/hangstrap/downloadFromPub>. Use it as `dart downloadFromPub.dart package m.n.l`.

Here, `package` is the package you want to install and `m.n.l` is the version number such as `0.8.1`. You will need to build this like any other dart package, and if during this process the `pub get` command fails, you will have to download the package and unpack it manually; however, from then on, you should be able to use this script to work around this issue.

When `pub get` fails in Dart Editor, try the following on the command line to get more information on the possible reasons for the `--trace 'upgrade'` failure.

There is now also a way to condense these four steps into one command in a terminal as follows:

```
pub cache repair
```

Shrinking the size of your app

On the web, the size of the JavaScript version of your app matters. For this reason, `dart2js` is optimized to produce the smallest possible JavaScript files.

How to do it...

When you're ready to deploy, minify the size of the generated JavaScript with `-m` or `--minify`, as shown in the following command:

```
dart2js -m -o prorabbits.js prorabbits.dart
```

Using `pub build` on the command line minifies JavaScript by default because this command is meant for deployment.

How it works...

The `dart2js` file utilizes a tree-shaking feature; only code that is necessary during execution is retained, that is, functions, classes, and libraries that are not called are excluded from the produced `.js` file. The minification process further reduces the size by replacing the names of variables, functions, and so on with shorter names and moving code around to use a few lines.

There's more...

Be careful when you use reflection.

More Information Section 1

Using reflection in the Dart code prevents tree shaking. So only import the `dart:mirrors` library when you really have to. In this case, include an `@MirrorsUsed` annotation, as shown in the following code:

```
library mylib;

@MirrorsUsed(targets: 'mylib')
import 'dart:mirrors';
```

In the previous code, all the names and entities (classes, functions, and so on) inside of `mylib` will be retained in the generated code to use reflection. So create a separate library to hold the class that is using mirrors.



Make sure your deployment web server uses gzipping to perform real-time HTTP compression.



See also

- ▶ You might want to consult the *Using Reflection* recipe in Chapter 4, Object Orientation.

Making a system call

A fairly common use case is that you need to call another program from your Dart app, or an operating system command. For this, the abstract class `Process` in the `dart:io` package is created.

How to do it...

Use the `run` method to begin an external program as shown in the following code snippet, where we start Notepad on a Windows system, which shows the question to open a new file `tst.txt` (refer to `make_system_call\bin\ make_system_call.dart`):

```
import 'dart:io';

main() {
    // running an external program process without interaction:
    Process.run('notepad', ['tst.txt']).then((ProcessResult rs){
        print(rs.exitCode);
        print(rs.stdout);
        print(rs.stderr);
    });
}
```

If the process is an OS command, use the `runInShell` argument, as shown in the following code:

```
Process.run('dir', [], runInShell:true).then((ProcessResult rs)
{ ... }
```

How it works...

The `Run` command returns a Future of type `ProcessResult`, which you can interrogate for its exit code or any messages. The exit code is OS-specific, but usually a negative value indicates an execution problem.

Use the `start` method if your Dart code has to interact with the process by writing to its `stdin` stream or listening to its `stdout` stream.



Both methods work asynchronously; they don't block the main app. If your code has to wait for the process, use `runSync`.

Using snapshotting

One of the advantages of running a Dart app on its own VM is that we can apply snapshotting, thereby reducing the startup time compared to JavaScript. A snapshot is a file with an image of your app in the byte form, containing all the Dart objects as they appear in the heap memory.

How to do it...

To generate a script snapshot file called `prorabbits` from the Dart script `prorabbits.dart`, issue the following command:

```
dart --snapshot=prorabbits prorabbits.dart
```

Then, start the app with `dart prorabbits args`, where `args` stands for optional arguments needed by the script.

How it works...

A script snapshot is the byte representation of the app's objects in the memory (more precisely in the heap of the started isolate) after it is loaded, but before it starts executing. This enables a much faster startup because the work of tokenizing and parsing the app's code was already done in the snapshot.

There's more...

This recipe is intended for server apps or command-line apps. A browser with a built-in Dart VM can snapshot your web app automatically and store that in the browser cache; the next time the app is requested, it starts up way faster from its snapshot. Because a snapshot is in fact a serialized form of an object(s), this is also the way the Dart VM uses to pass objects between isolates. The folder `dart/dart-sdk/bin/snapshots` contains snapshots of the main Dart tools.

See also

- ▶ Occasionally, your app needs access to the operating system, for example, to get the value of an environment variable to know where you are in the filesystem, or to get the number of processors when working with isolates. Refer to the *Using isolates in the Dart VM* and *Using isolates in web apps* recipes, in *Chapter 8, Working with Futures, Tasks, and Isolates*, for more information on working with isolates.

Getting information from the operating system

In this recipe, you will see how to interact with the underlying operating system on which your app runs by making system calls and getting information from the system.

Getting ready

The `Platform` class provides you with information about the OS and the computer the app is executing on. It lives in `dart:io`, so we need to import this library.

How to do it...

The following script shows the use of some interesting options (refer to the code files `tools\code\platform\bin\platform.dart` of this chapter):

```
import 'dart:io';

Map env = Platform.environment;

void main() {
    print('We run from this VM: ${Platform.executable}');
    // getting the OS and Dart version:
    print('Our OS is: ${Platform.operatingSystem}');
    print('We are running Dart version: ${Platform.version}');
    if (!Platform.isLinux) {
        print('We are not running on Linux here!');
    }
    // getting the number of processors:
    int noProcs = Platform.numberOfProcessors;
    print('no of processors: $noProcs');
    // getting the value of environment variables from the Map env:
    print('OS = ${env["OS"]} ');
    print('HOMEDRIVE = ${env["HOMEDRIVE"]} ');
    print('USERNAME = ${env["USERNAME"]} ');
    print('PATH = ${env["PATH"]} ');
    // getting the path to the executing Dart script:
    var path = Platform.script.path;
    print('We execute at $path');
    // on this OS we use this path separator:
    print('path separator: ${Platform.pathSeparator}');
}
```

When run, the above code gives the following output:

```
Our OS is: windows
We are running Dart version: 1.3.3 (Wed Apr 16 12:40:55 2014) on
"windows_ia32"
We are not running on Linux here!
```

```
no of processors: 8
OS = Windows_NT
HOMEDRIVE = C:
USERNAME = CVO
PATH = C:\mongodb\bin;C:\MinGW\bin;...
We execute at /F:/Dartiverse/platform/bin/platform.dart
path separator: \
```

How it works...

Most of the options are straightforward. You can get the running VM from `Platform.executable`. You can get the OS from `Platform.operatingSystem`; this can also be tested on a Boolean property such as `Platform.isLinux`. The Dart version can be tested with the `Platform.version` property. The `Platform.environment` option returns a nice map structure for the environment variables of your system, so you can access their values by name, for example, for a variable `envVar`, use `var envVar = Platform.environment["envVar"]`.

To get the path of the executing Dart script, you can use the `path` property of `Platform.script` because the latter returns the absolute URI of the script. When building file paths in your app, you need to know how the components in a path are separated; `Platform.pathSeparator` gives you this information.

There's more...

Don't confuse this class with `Platform` from `dart:html`, which returns information about the browser platform.

2

Structuring, Testing, and Deploying an Application

In this chapter, we will cover the following topics:

- ▶ Exiting from an app
- ▶ Parsing command-line arguments
- ▶ Structuring an application
- ▶ Using a library from within your app
- ▶ Microtesting your code with assert
- ▶ Unit testing a Polymer web app
- ▶ Adding logging to your app
- ▶ Documenting your app
- ▶ Profiling and benchmarking your app
- ▶ Publishing and deploying your app
- ▶ Using different settings in the checked and production modes

Introduction

In this chapter, we focus mainly on all the different tasks in the lifecycle of your project that make it more professional, and will save much more time in the maintenance phase. This includes structuring the project, installing a logging tool in it, testing, documenting, profiling, and publishing it. However, first we see how we can end an app, and how a server app can take command-line arguments.

Exiting from an app

A Dart program starts its execution from the `main()` function in one thread (or isolate) in the Dart VM. The Dart VM by design always starts up single threaded. The program can end in three different ways:

- ▶ It can end in a normal way by executing the last statement from `main()` and returning the exit code with the value 0, which means success
- ▶ It can terminate abnormally with a runtime exception, returning exit code different from 0, such as 255 in the case of an unhandled exception
- ▶ It can wait in an event loop for user interaction (such as in the browser or a web server waiting for requests), and then terminate when the browser is closed or another app is started in the same browser tab

However, how can we exit the app from the code itself? This can be useful, for example, in a server-side VM app with some Futures that may or may not return.

How to do it...

The first possibility is to use the `exit(int code)` top-level function from `dart:io`, as in `exit_app.dart`, to stop the app from an endless loop or at a certain condition, and return the exit code:

```
import 'dart:io';

void main() {
  var message = "Dart is fun!";
  int i = 0;
  while (true) {
    print(message);
    i++;
    if (i == 10) {
      print("That's enough!");
      exit(0);
    }
  }
}
```

```
    exit(10);  
}  
}  
}
```

You can also set the exit code value with the property `exitCode`, as shown in the following code:

```
exitCode = 10;  
// ... other code can be executed  
exit(exitCode);
```

How it works...

The `exit` (code) function will terminate the running Dart VM process and return the integer code as the exit value to the parent process or OS environment, indicating the success, failure, or other exit state of the program. You can choose the value of the code; there is a convention to use 0 for success, 1 for warnings, and 2 for errors. Another convention is zero for success, non-zero for failure, and a program returning a warning for a successful exit because it naturally reached its end. A concrete example is the `dartanalyzer` program, which returns 0 if the code generates warnings.

Setting the exit code is preferred because the program can still run to its natural completion, or some cleanup or finalizing code (such as closing a file or database connection) can be run before `exit(exitCode)` ends the app. It is also good practice to start `main()` with `exitCode = 0`, presuming success as the normal ending state.



What exit codes mean is platform-specific; you will not run into cross-platform issues if you use exit codes in the range of 0-127.

Parsing command-line arguments

A server app that runs a batch job often takes parameter values from the command line. How can we get these values in our program?

How to do it...

The obvious way to parse command-line arguments is as follows (see `command_line_arguments.dart`):

```
void main(List<String> args) {  
    print("script arguments:");
```

```
for(String arg in args)
    print(arg);
}
```

Now, the command `dart command_line_arguments.dart param1 param2 param3` gives you the following output:

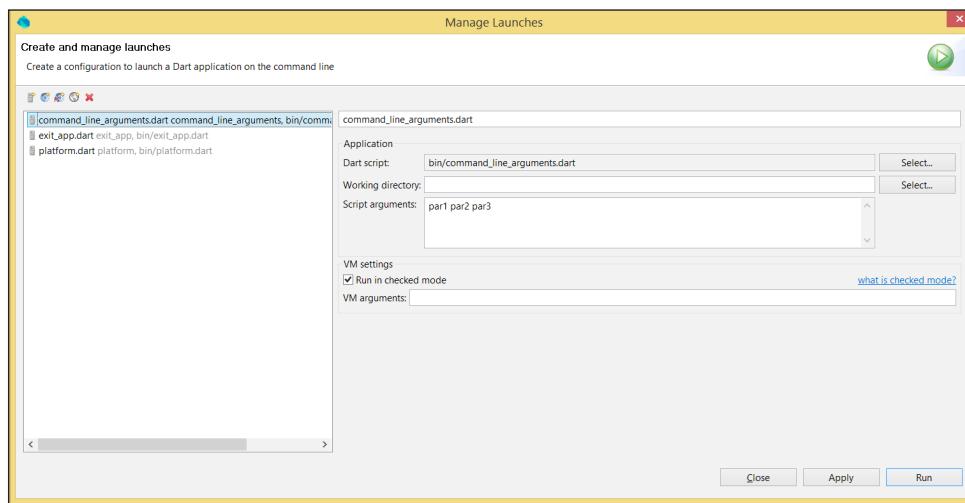
script arguments:

param1

param2

param3

However, you can also test this from within Dart Editor, open the menu **Run**, and select **Manage Launches** (**Ctrl + Shift + M**). Fill in the parameters in the **Script arguments** window:



Script arguments

What if your parameters are in the `key:value` form, as shown in the following code?

```
par1:value1 par2:value2 par3:value3
```

In this case, use the following code snippet:

```
for(String arg in args) {
    List<String> par = arg.split(':');
    var key = par[0];
    var value = par[1];
    print('Key is: $key - Value is: $value');
}
```

The previous code snippet gives you the following output:

Key is: par1 - Value is: value1

Key is: par2 - Value is: value2

Key is: par3 - Value is: value3

The `split` method returned `List<String>` with a key and value for each parameter. A more sophisticated way to parse the parameters can be done as follows:

```
final parser = new ArgParser();
argResults = parser.parse(args);
List<String> pars = argResults.rest;
print(pars); // [par1:value1, par2:value2, par3:value3]
```

Again, use `split` to get the keys and values.

How it works...

The `main()` function can take an optional argument `List<String> args` to get parameters from the command line. It only takes a split of the parameter strings to get the keys and values.

The second option uses the `args` package from the pub repository, authored by the Dart team. Include `args: any` in the dependencies section of the `pubspec.yaml` file. Then, you can use the package by including `import 'package:args/args.dart'`; at the top of the script. The `args` package can be applied both in client and server apps. It can be used more specifically for the parsing of GNU and POSIX style options and is documented at <https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/args/args>.

See also

- ▶ Refer to the *Searching in files* recipe in *Chapter 6, Working with Files and Streams*, for an example of how to use the `args` package with a flag

Structuring an application

All Dart projects that are meant to be used in a production environment should follow best software engineering practices and hence, must contain a particular structure of folders. A well-structured project breathes professionalism and gives developers a sense of recognition; it is much easier to find your way in a standardized structure. Moreover, it is also necessary if you want to use a bunch of application-specific libraries in your app, as we will see in the next recipe.

Getting ready

An app that is meant to run on its own, either as a command-line application or a web application, is an application package; it needs a `main()` entry point. A library package will be used as a dependency in other apps. All Dart projects depend on the configuration file `pubspec.yaml`, which describes the app and its dependencies, together with the `pubspec.lock` file. This dictates which libraries will be contained in the top-level `packages` folder. This file and the `packages` folder are generated by the `pub` tool (more specifically, the `pub get` and `pub upgrade` commands) and should not be edited.

How to do it...

If you develop an application in Dart Editor when starting up a new project, you will need to choose a project template to begin with, as shown in the following table:

Project type	Template	Project folder
Client or server app to be run standalone	Command line	bin
Web or Polymer app	Web application or web application using Polymer, or project	web
Chrome app	Chrome-packaged application	web
Library	Package	lib

The `bin` folder contains a startup script with a `main()` function. It can also contain shell scripts, for example, a script to start a server. In the `web` folder, you will typically have `index.html` and a `main.dart` file, or in general, the `app.html` and `app.dart` files. Other resource files such as CSS, JavaScript files, and images can be contained in their own folders `css`, `js`, and `images`. A Polymer project will typically contain a `web\component` subfolder. Don't place any scripts with `main()` in a `lib` folder.

Then, you will want to enhance the structure of the project as follows:

Folder	Project type	Files
at top-level	All	README.md, LICENSE, CHANGELOG, and AUTHORS files
doc	All	getting_started.md, todo.txt
example	All	Example scripts showing how to use the app
lib	Web	Folders for src, view, and model
src	Server	Folders for src and model
test	All	Unit test scripts

How it works...

The templates from Dart Editor provide you with a basic structure, but usually you'll want to add some folders as specified previously to provide a recognizable and professional structure where you can easily find what you want to look at, for example, which tests are included with the project. As we will see in the next recipe, in order to use application libraries, they have to be placed in the lib folder.

There's more...

The README file (readme.md, which is in the markdown syntax; refer to <http://en.wikipedia.org/wiki/Markdown>) and the CHANGELOG file are shown in the pub repository on the page of your package, so their content is important.

Optional folders are as follows:

- ▶ mock: This contains classes to simulate certain behaviors of your app in testing environments, for example, a text file instead of a real database
- ▶ tool: This contains tooling scripts needed in the project such as a build script, test runners, and so on
- ▶ benchmark: When the performance is critical, this folder can contain examples to test it

An alternative structure for an app with both client and server components can be placed on top of the previous structure:

- ▶ client
- ▶ server
- ▶ core (or shared)

The <https://www.dartlang.org/tools/pub/package-layout.html> link on the Dart site contains some additional information.

See also

- ▶ You might also want to read the *Publishing and deploying your app* recipe in this chapter

Using a library from within your app

As indicated in the previous recipe, every kind of app can contain a `lib` folder, which at the very least contains the model classes. These model classes are very important because they form the backbone of your project, so they must be accessible in your entire application. You can do this by placing them at the top in a `lib/model` folder, or even better in the `lib/model`. This central position will also make them stand out and easy to find for other readers of your code.

How to do it...

Take a look at the structure of the `bank_terminal` project. The model classes `Person` and `BankAccount` are placed in the `lib\model` folder. Give your project a name in the `pubspec.yaml` file:

```
name: bank_terminal
```

Then, use the same name for the library script you created in the `lib` folder `bank_terminal.dart`, which contains the following code:

```
library bank_terminal;

import 'dart:convert';

part 'model/bank_account.dart';
part 'model/person.dart';
```



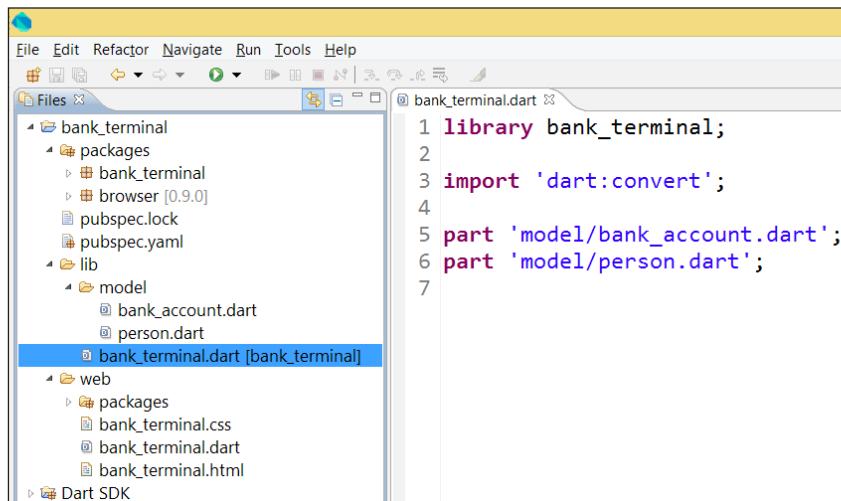
The library has the same name as your app! This is not required for the Dart script itself, but it is a common and advised practice.



Now when a `pub get` or `pub upgrade` command is performed, a `bank_terminal` folder appears in `packages`. You can then make this library available for use in your other Dart scripts by importing it as any hosted package you would have downloaded from the pub. For example, in `web\bank_terminal.dart` we have the following code:

```
import 'package:bank_terminal/bank_terminal.dart';
```

In this case, the model classes are made available. The project structure is shown in the following screenshot:



Using a library in your app

How it works...

The pub tool was designed to work this way to make it easy to use internal libraries for your app. Every script that declares a library in the `lib` folder (or its subfolders) will be picked up by the `pub get` or `pub upgrade` commands. The result is that the library with all its code is considered a separate "internal" package, and thus placed in the `packages` folder together with other packages your app depends on. This makes for a clean code model, and it also makes it easier for the pub to deploy your code.

There's more...

The pub tool can be extended to several subfolders of lib, each containing their own library file (for example, a script `model.dart` that starts with the `library model` in the `model` folder) and then one top-level library file `project_name.dart`, containing the following code as its first lines:

```
library project_name;
import 'model/model.dart';    // importing library model
import 'view/view.dart';      // importing library view

...
```

This way, the different libraries from `model`, `view`, and so on are imported into one big library, which is then imported by the app as follows:

```
import 'package:project_name/project_name.dart';
```

Microtesting your code with assert

Writing tests for your app is necessary, but it is not productive to spend much time on trivial tests. An often underestimated Dart keyword is `assert`, which can be used to test conditions in your code.

How to do it...

Look at the code file `microtest.dart`, where `microtest` is an internal package as seen in the previous recipe:

```
import 'package:microtest/microtest.dart';

void main() {
  Person p1 = new Person("Jim Greenfield", 178, 86.0);
  print('${p1.name} weighs ${p1.weight}' );
  // lots of other code and method calls
  // p1 = null;
  // working again with p1:
  assert(p1 is Person);
  p1.weight = 100.0;
  print('${p1.name} now weighs ${p1.weight}' );
}
```

We import the `microtest` library, which contains the definition of the `Person` class. In `main()`, we create a `Person` object `p1`, go through lots of code, and then want to work with `p1` again, possibly in a different method of another class. How do we know that `p1` still references a `Person` object? In the previous snippet, it is obvious, but it can be more difficult. If `p1` was, for example, dereferenced, without `assert` we would get the exception **NoSuchMethodError: method not found: 'weight='**.

However, if we use the `assert` statement, we get a much clearer message: **AssertionError: Failed assertion: line 9 pos 9: 'p1 is Person' is not true**. You can test it by uncommenting the line `p1 = null`.

How it works...

The `assert` parameter is a logical condition or any expression (such as calling a function returning a Boolean value) that resolves to false or true. If its value is false, the normal execution is stopped by throwing an `AssertionError`.

Use `assert` to test any non-obvious conditions in your code; it can replace a lot of simple unit tests or unit tests that can be difficult to set up. Rest assured `assert` only works in the checked mode; it does not affect the performance of your deployed app because it is ignored in the production mode.

There's more...

Testing with `assert` is often very useful when entering a method to test conditions on parameters (preconditions), and on leaving a method testing the return value (postconditions). You can also call a `test` function (which has to return a Boolean value) from `assert` such as `assert(testfunction()) ;`.

Unit testing a Polymer web app

A project should contain a number of automated tests that can be run after every code change to ensure that the previous functionality still works. Dart's `unittest` framework is the best tool for the job, and the Dart website has some excellent articles to get you started. However, testing Polymer web applications is a lot trickier because of the way Polymer works, as it hides HTML in shadow DOM and also because it works in an asynchronous fashion, independent of the testing code.

Getting ready

We will create some tests in the `ClickCounter` example (the standard web application template using the `polymer` library). You can find the code in the `polymer1` app. We include the `unittest` library in the `pubspec.yaml` file, and create a `test` folder.

How to do it...

In the `test` folder, we create a `test_polymer1.html` page; a web page that loads the Polymer component required to test this functionality. The following is the minimum content required for the component:

```
<head>
  <title>test_polymer1</title>
  <link rel="import" href="../web/polymer1.html">
  <script type="application/dart"    src="test_polymer1.dart"></
script>
</head>

test_polymer1.dart contains the test script:
import 'package:unittest/unittest.dart';
import 'dart:html';
import 'package:polymer/polymer.dart';
import '../web/clickcounter.dart';

main() {
  initPolymer();

  var _el;

  setUp(() {
    _el = createElement('<click-counter>Click counter test</click-
counter>');
    document.body.append(_el);
  });

  tearDown(() {
    _el.remove();
  });

  // tests:
  test('shadowroot elements are created', (){
expect(querySelector('click-counter').children, isNotNull);
expect(querySelector('click-counter').shadowRoot.text, isNotNull);
  });
  test('initial text ok', (){
expect(querySelector('click-counter').shadowRoot.text.
contains('click count: 0'), isTrue);
  });
}
```

```
// test button with text Transaction:  
test('button with id click exists', (){  
var button = querySelector('click-counter').shadowRoot.  
querySelector('#click');  
    expect(button, isNotNull);  
});  
test('button click() increments counter', (){  
ButtonElement button = querySelector('click-counter').shadowRoot.  
querySelector('#click');  
    button.click();  
    button.click();  
    button.click();  
    // get counter value:  
    ClickCounter cc = querySelector('click-counter');  
    expect(cc.count, 3); // after 3 clicks  
});  
}  
  
createElement(String html) =>  
    new Element.html(html, treeSanitizer: new NullTreeSanitizer());  
  
class NullTreeSanitizer implements NodeTreeSanitizer {  
    void sanitizeTree(node) {}  
}
```

When the script is run, the following output appears:

unittest-suite-wait-for-done

PASS: shadowroot elements are created

PASS: initial text ok

PASS: button with id click exists

PASS: button click() increments counter

All 4 tests passed.

unittest-suite-success

How it works...

The test web page loads the Polymer component, and the test script loads the `unittest` and `Polymer` packages and component classes (`clickcounter.dart`). In `main()`, we load the `Polymer` package with `initPolymer()`; in `setup()`, we use a helper method `createElement()`, with an HTML string containing the `Polymer` tag `<clickcounter>` as argument to instantiate the Polymer component and add it to the page. This was done in order to avoid the default HTML sanitization `createElement()`, which uses a null sanitizer instead of the built-in sanitizer (refer to *Chapter 5, Handling Web Applications*, for more information on this topic). Then, we start testing, for example:

- ▶ `expect(querySelector('click-counter').children, isNotNull);` so that the Polymer component tree is created
- ▶ `var button = querySelector('click-counter').shadowRoot.querySelector('#click');` `expect(button, isNotNull);` so that the button with the ID 'click' is created
- ▶ `expect(querySelector('click-counter').shadowRoot.text.contains('click count: 0'), isTrue);` so that the text initially displayed is 'click count: 0'.

Notice how we have to dig into `shadowRoot` of the Polymer component to get this information as follows:

- ▶ Verify that after clicking the button three times invoked by `button.click()`, our `count` property has the value 3:
`ClickCounter cc = querySelector('click-counter');`
`expect(cc.count, 3);`

See also

- ▶ To discover more information about the Dart `unittest` library, refer to the book *Learning Dart*, Ivo Balbaert, Dzenan Ridjanovic, Packt Publishing, or the excellent articles at <https://www.dartlang.org/articles/writing-unit-tests-for-pub-packages/> and <https://www.dartlang.org/articles/dart-unit-tests/>. These should help give you further background knowledge and information on how to extend the use of Dart `Unittest` in your own work.

Adding logging to your app

Every production app needs a logger functionality that allows you to output log messages at varying levels of severity (information/warning/debug) to the (web browser's debug) console or a file. This recipe will enable you to do just that quickly and easily.

Getting ready

Use the logging package developed by the Dart team available from pub for this purpose. Add it to your `pubspec.yaml` file, and add the code line `import 'package:logging/logging.dart';` to your code. See it in action in `bank_terminal_polymer`. We add the import to the code of the Polymer component and model class `BankAccount`.

How to do it...

1. In `web\bank_account.dart`, we have at the top level the following code:

```
import 'package:logging/logging.dart';
final Logger log = new Logger('Bank Account');
```

2. We change the constructor to the following code:

```
BankAccount.created() : super.created() {
    setupLogger();
    log.info('Bank Account component is created');
}

setupLogger() is the place where you can define the format of your
logs, the following code presents a minimal format:
setupLogger() {
    // Set up logger.
    Logger.root.level = Level.ALL;
    Logger.root.onRecord.listen((LogRecord rec) {
        print('${rec.level.name}: ${rec.time}: ${rec.message}');
    });
}
```

3. In `checkAmount()`, we add the following warning message:

```
checkAmount(String in_amount) {
    try {
        amount = double.parse(in_amount);
    } on FormatException catch(ex) {
        log.warning("Amount $in_amount is not a double!");
        return false;
    }
    return true;
}
```

4. In the model class in lib\bank_account.dart file, we add an "info" message when the BankAccount object is created: log.info('Bank Account is created'), and in the transact method, we add "severe message" when the balance becomes negative:

```
transact(double amount) {  
    balance += amount;  
    if (amount < 0 && (-amount) > balance) {  
        log.severe("Balance will go negative!");  
    }  
    date_modified = new DateTime.now();  
}
```

5. If we then run the app, input an amount 50q, and then an amount -5000, which will make our balance negative. This means we will get the following console output:

```
INFO: 2014-04-28 11:27:33.525: Bank Account component is created  
FINE: 2014-04-28 11:27:33.551: [Instance of '_Binding']: bindProperties: [value]  
to [bank-account].[Symbol("bac")]  
FINE: 2014-04-28 11:27:33.557: [bank-account] cancelUnbindAll  
FINE: 2014-04-28 11:27:33.561: [bank-app] cancelUnbindAll  
INFO: 2014-04-28 11:27:33.561: Bank Account is created  
FINE: 2014-04-28 11:27:39.172: >>> [bank-account]: dispatch enter  
INFO: 2014-04-28 11:27:39.176: <<< [bank-account]: dispatch enter  
WARNING: 2014-04-28 11:27:44.089: Amount 50qs is not a double!  
SEVERE: 2014-04-28 11:29:02.778: Balance will go negative!  
INFO: 2014-04-28 11:29:02.778: <<< [bank-account]: dispatch transact
```

How it works...

The object of the Logging class must first be configured; otherwise, nothing happens. This is done in `setupLogger()`, which does the following things:

- ▶ It sets the level of the messages (choose between SHOUT, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST, ALL, or OFF, or predefined your own).
- ▶ It sets up an event handler to listen for the `onRecord` stream. This processes objects of type `LogRecord`, which have access to the name, time, message, and stacktrace. Then, you code what you want to do with this event, print it to the console, write it in a file, send it in a mail, and so on.

There's more...

The following remarks tell you what to do in some special cases. To quickly display errors in a web app, you can add the following code in your code:

```
window.console.error('Something bad occurred');
```

If you want to log something asynchronously, use this snippet:

```
Future futr = doAsync();
futr.then((result) {
  log.fine('This result came back: $result');
  processResult(result);
})
.catchError((e, stackTrace) => log.severe('Something went wrong - ',
e, stackTrace));
```

Documenting your app

A project needs to be documented so that future developers can change and expand it. From release 1.2 onwards, the docgen tool is provided as an API Documentation Generator, which generates and serves documentation for the Dart SDK and packages, as well as for your applications. This recipe is designed to show you how to best document the application you have built.

Getting ready

Run `pub get` on your project before running the `docgen` tool. We will illustrate this with the `bank_terminal_polymer` app.

How to do it...

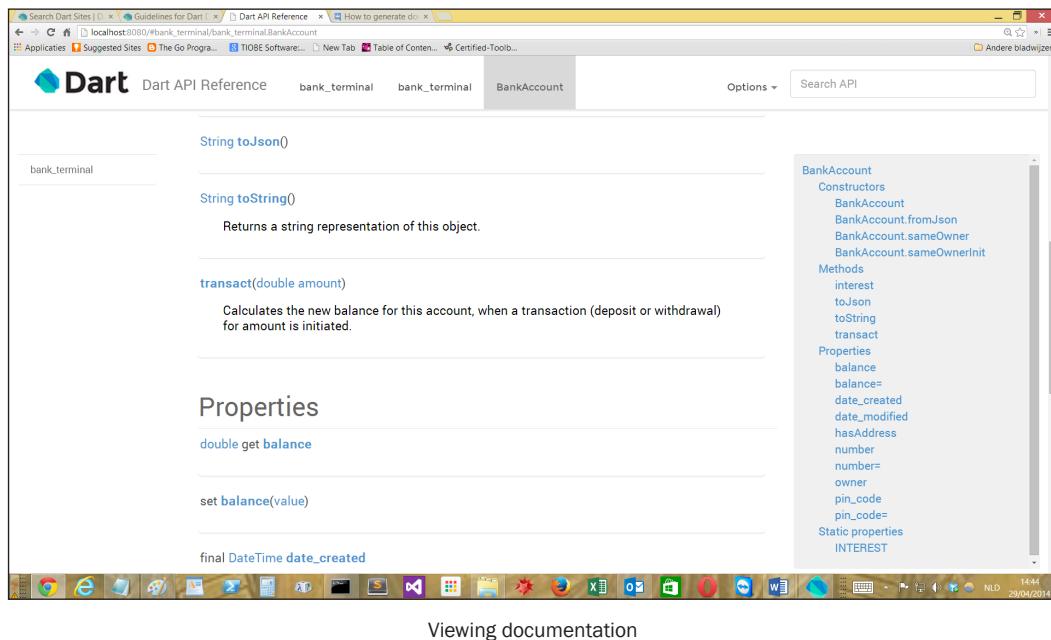
1. In a command-line session at the top level of your app, run:

```
docgen .
```

2. When the documentation is generated, issue the following command:

```
docgen --serve .
```

This installs the `dartdoc-viewer` tool and starts up a local web server for your docs, which you can access via the URL `http://localhost:8080`. The following is a screenshot from the `BankAccount` class documentation (you can change the view with the **Options** menu):



Viewing documentation

How it works...

The command creates a `docs` folder in your app, and writes JSON files with the API information in them for your app's `lib` folder, as well as for every imported package and the Dart SDK. Dartdoc comments (`/** ... */`) are included.

There's more...

To make the docs available from an intranet web server or from an Internet web server, perform the following steps:

1. Copy `dartdoc-viewer` (compiled to JavaScript) onto the server.
2. Copy the generated files to a `docs` directory under the main URL.

Also, private declarations are ignored unless the flag `--include-private` is used. Various other options are documented at <https://www.dartlang.org/tools/docgen/#options>.

A command that specifies the output folder and does not generate docs for the SDK and packages will be as follows:

```
docgen --out out/doc/api --no-include-sdk --no-include-dependent-packages  
--compile lib/app.dart
```

Profiling and benchmarking your app

One of the key success factors of Dart is its performance. The Dart VM has a built-in optimizer to increase the execution speed. This optimizer needs a certain amount of execution time before it has enough information to do its work. This is no problem for an app in production, but when testing, you must ensure this condition is met. In this recipe, we are going to focus on how to best benchmark your application.

Getting ready

The `benchmark_harness` package from the Dart team is built specifically for this purpose. Add it to the `pubspec.yaml` file and import it in your code. We will illustrate this with the `template_benchmark` app. Also, make sure that all the issues detected by the checked mode are solved (because these could have an effect on the execution speed) and that the app is run in the production mode (refer to the *Setting up the checked and production modes* recipe in *Chapter 1, Working with Dart Tools*).

How to do it...

Perform the following steps to get the benchmark harness working:

1. Import the benchmark library:

```
import 'package:benchmark_harness/benchmark_harness.dart';
```

2. Define a class `TemplateBenchmark` with its own `main()` and `run()` methods. It is a subclass of class `BenchmarkBase` in the `benchmark` library. The benchmark is started from `run()`:

```
class TemplateBenchmark extends BenchmarkBase {  
  const TemplateBenchmark() : super("Template");  
  
  static void main() {  
    new TemplateBenchmark().report();  
  }  
  
  void run() {  
    fib(20);  
  }  
}
```

```
// recursive algorithms:  
int fib(int i) {  
    if (i < 2) return i;  
    return fib(i-1) + fib(i-2);  
}  
  
// int fib(n) => n<2 ? n : fib(n-2) + fib(n-1);  
  
// iterative algorithm:  
// int fib(int i){  
//     int a = 0; int b = 1;  
//     for (int n=a; n < b; n++) {  
//         a = a + b; b = a;  
//     }  
//     return a;  
// }  
void setup() {}  
void teardown() {}  
}
```

3. The following code starts the whole benchmark machinery:

```
main() {  
    TemplateBenchmark.main();  
}
```

4. If we benchmark the Fibonacci algorithm for the two recursive implementations (with `if` and with the ternary operators) and the iterative algorithm, we get the following results:

Template(RunTime): 482.392667631452 us.

Template(RunTime): 498.00796812749 us.

Template(RunTime): 0.2441818187589752 us.

As we expected, the iterative algorithm performs orders of magnitude better than working recursively.

How it works...

Create a new benchmark by extending the `BenchmarkBase` class (here, `TemplateBenchmark`). The benchmarked code is called from the `run()` method within this subclass. The `setup()` and `teardown()` function code are run before and after the benchmark to prepare for the benchmark or clean up after it. These are not taken into account for the benchmark itself. The top-level `main()` function runs the benchmark by calling `main()` from the subclass of `BenchmarkBase`.

See also

- ▶ Refer to the new Observatory tool to profile Dart apps at <https://www.dartlang.org/tools/observatory/>

Publishing and deploying your app

There comes a point in time where you consider your app to be production ready, and you are eager to hand it over to your clients or users. If it is a web app in the Dart world at this moment in time, this means compiling to JavaScript. Luckily, the pub tool will take care of this stage in your app's life, so that your app can be deployed successfully.

Getting ready

This is pretty straightforward. To prepare, you need to run and test your application in both the checked and production modes.

How to do it...

Run the `pub build` command (see the *Compiling your app to JavaScript* recipe in Chapter 1, *Working with Dart Tools*), either from the command line or in Dart Editor. This creates the `build` folder with the subfolder `bin` or `web`, respectively, for a command-line or web application. The `build` folder contains complete deliverable files. The files generated in there can be deployed like any static content. Upload the JavaScript files together with the web pages and resources to any production web server.

How it works...

The `pub build` command is Dart's optimized command to create the deployment assets. It performs tree shaking so that only the code that is necessary during execution is retained, that is, functions, classes, and libraries that are not called are excluded from the produced `.js` file. The minification process further reduces the size of the file by replacing the names of variables, functions, and so on with shorter names and moving the code around to use a few lines.

There's more...

Some application hosting sites to run your app in the cloud are as follows:

- ▶ Heroku is a Platform as a Service for cloud-hosted web apps. It doesn't yet officially support the Dart runtime, but it can already be used to do just that. Refer to <http://blog.sethladd.com/2012/08/running-dart-in-cloud-with-heroku.html> and the Dart server code lab at <https://www.dartlang.org/codelabs/deploy> for more information.
- ▶ DartVoid at <http://www.dartvoid.com/> using the Vane middleware framework; it comes with support for MongoDB.
- ▶ Google itself offers the possibility of running a Dart server on Google Compute Engine (refer to <http://alexpaluzzi.com/tag/dartlang/>) and on Google's other cloud virtual machines in the future.

Once your app is ready to be shared with other developers, it can also be published to the pub repository (`pub.dartlang.org`). This is done with the `pub publish` command; this will verify the contents of your `pubspec.yaml` configuration file and the app's folder structure. For more detailed information, refer to <https://www.dartlang.org/tools/pub/publishing.html>.

Using different settings in the checked and production modes

Often the development and deployment environments are different. For example, the app has to connect to a different database or a different mail server in either environment to use a mocked service in development and the real service in production, or something like that. How can we use different setups in both modes, or achieve a kind of precompiler directive-like functionality?

How to do it...

Perform the following steps to use different settings:

1. Add a transformers section to `pubspec.yaml` with an environment line that specifies a map of the settings, names and values, as follows (see the code in `dev_prod_settings`):

```
transformers: # or dev_transformers
- $dart2js:
  environment: {PROD: "true", DB: "MongoPROD"}
```

2. You can, for example, get the value of the DB setting from `const String.fromEnvironment('DB')`, as you can see in the following code:

```
import 'dart:html';

void main() {
  print('PROD: ${const String.fromEnvironment('PROD')}');
  bool prod = const String.fromEnvironment('PROD') == 'true';
  if (prod) {
    // do production things
    window.alert("I am in Production!");
    connectDB(const String.fromEnvironment('DB'));
  }
  else { // do developer / test things }
  log('In production, I do not exist');
}

log(String msg) {
  if (const String.fromEnvironment('DEBUG') != null) {
    print('debug: $msg');
  }
}

connectDB(String con) {
  // open a database connection
}
```

3. When run in the Dart VM (and in the checked mode), the console gives `PROD: null` as an output when run as JavaScript in Chrome; an alert dialog appears and the console in the **Developer Tools** shows **PROD: true**.

How it works...

The import initializer option requires a manual code change in order to switch the environment. The transformers option uses environment declarations that are provided by the surrounding system compiling or running the Dart program. This is better because it only requires changing the configuration file `pubspec.yaml`. However, at the moment, it is only defined for `dart2js`, in order to deploy to JavaScript.

Make sure that the environment is indented; otherwise, you may get the error "transformers" must have a single key: the transformer identifier.

3

Working with Data Types

In this chapter, we will cover the following recipes:

- ▶ Concatenating strings
- ▶ Using regular expressions
- ▶ Strings and Unicode
- ▶ Using complex numbers
- ▶ Creating an enum
- ▶ Flattening a list
- ▶ Generating a random number within a range
- ▶ Retrieving a random element from a list
- ▶ Working with dates and times
- ▶ Improving performance in numerical computations
- ▶ Using SIMD for enhanced performance

Introduction

This chapter is about working with the different data types Dart has to offer. The basic data types available are `var` (stores any object); `num` (stores any number type); `int`, `double`, `String`, `bool`, `List` (arrays); and `Map` (associative arrays). All of these data types are declared in the `dart:core` library. We will talk about strings, random numbers, complex numbers, dates and times, enums, and lists. We will cover a lot of tricks to help you out in specific circumstances. To get a quick overview of all the data types in Dart, refer to <https://www.dartlang.org/docs/dart-up-and-running/contents/ch02.html#built-in-types>.

Concatenating strings

Concatenation can be done in a variety of ways in Dart (refer to the `concat_trim_strings` file, and download it from www.packtpub.com/support).

How to do it...

Strings can be concatenated as follows:

```
String s1 = "Dart", s2 = "Cook", s3 = "Book";
var res = "Dart" " Cook" "Book";           (1)
res = "Dart" " Cook"
      "Book";                  (2)
res = s1 + " " + s2 + s3;           (3)
res = "$s1 $s2$s3";            (4)
res = [s1, " ", s2, s3].join();       (5)

var sb = new StringBuffer();          (6)
sb.writeAll([s1, " ", s2, s3]);
res = sb.toString();
print(res); // Dart CookBook
```

How it works...

Adjacent string literals are taken together as one string as shown in line (1), even if they are on different lines as shown in line (2). The `+` operator does the same thing (3), as well as string interpolation (4), which is the preferred way. Still there is another way to add `join()` to `List<String>` as shown in line (5). The most efficient way, especially if you want to apply the `+` operator in a `for` loop, is to work with `StringBuffer` as shown in line (6); the concatenation only happens when `toString()` is called.



So if you have to glue a large number of strings together, use `StringBuffer`. Avoid concatenation using `+`. This will save you memory and will execute the file much faster.

There's more...

The `writeAll` method can take an optional separator argument, as in `sb.writeAll([s1, " ", s2, s3], '-')`, resulting in Dart - - Cook - Book.

Using regular expressions

Regular expressions are an indispensable tool in every programming language to search for matching patterns in strings. Dart has the `RegExp` class from `dart:core`, which uses the same syntax and semantics as JavaScript.

How to do it...

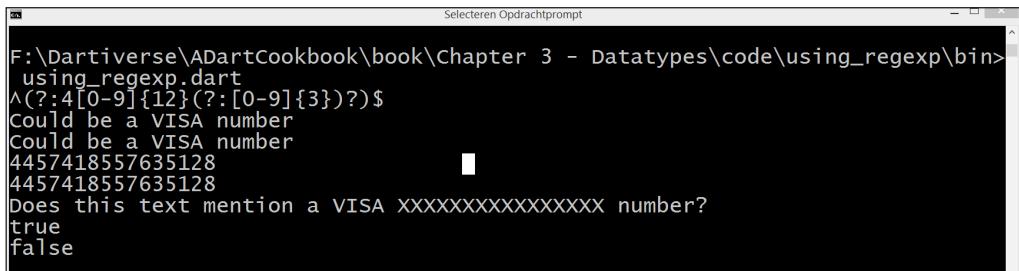
We use `RegExp` in the following code (see `using_regexp.dart`) to quickly determine whether a credit card number seems valid:

```
var visa = new RegExp(r"^(?:(?:[0-9]{12}(?:[0-9]{3})?)?)$");
var visa_in_text = new RegExp(r"\b(?:[0-9]{12}(?:[0-9]{3})?)\b");
var input = "4457418557635128";
var text = "Does this text mention a VISA 4457418557635128 number?";

void main() {
    print(visa.pattern);
    // is there a visa pattern match in input?
    if (visa.hasMatch(input)) {
        print("Could be a VISA number");
    }
    // does string input contain pattern visa?
    if (input.contains(visa)) {
        print("Could be a VISA number");
    }
    // find all matches:
    var matches = visa_in_text.allMatches(text);
    for (var m in matches) {
        print(m.group(0));
    }
}
```

```
visa_in_text.allMatches(text).forEach((m) => print(m[0]));
// let's hide the number:
print(text.replaceAll(visa_in_text, 'XXXXXXXXXXXXXXXXXX'));
print(visa.isCaseSensitive);
print(visa.isMultiLine);
}
```

The previous code gives the following output:



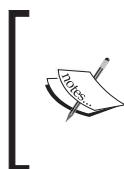
```
F:\Dartiverse\ADartCookbook\book\Chapter 3 - Datatypes\code\using-regexp\bin>
using_regex.dart
^(?:4[0-9]{12}(?:[0-9]{3})?)$ Could be a VISA number
Could be a VISA number
4457418557635128
4457418557635128 Does this text mention a VISA XXXXXXXXXXXXXXXX number?
true
false
```

How it works...

Credit card numbers are just a sequence of 13 to 16 digits, with one to four specific digits at the start that identify the card company. A regular expression is specified as a raw string `r"..."`, where ... is the pattern. The `hasMatch` code tells you whether there is a match or not, and `allMatches` produces a collection you can walk through. In most cases, the convenience method for a single match, `firstMatch` is what you need. The `allMatches.length` part gives you the number of matches. The `for` loop can also be written more functionally as `visa_in_text.allMatches(text).forEach((m) => print(m[0]));`.

There's more...

To further verify the card number before calling the credit card verification service, you should code the Luhn algorithm (http://en.wikipedia.org/wiki/Luhn_algorithm). The class `RegExp` implements Perl-style regular expressions. However, it lacks a number of advanced features such as named capturing groups or conditionals.



Refer to <https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:core.RegExp> for all the details on Dart. For more information on the regular expression syntax itself, refer to <http://www.regular-expressions.info/refquick.html>.

Strings and Unicode

Dart Strings are immutable sequences of UTF-16 code units. UTF-16 combines surrogate pairs, and if you decode these, you get Unicode code points. Unicode terminology is terse, but Dart does a good job of exposing the different parts.

How to do it...

We will see the different methods to perform action on strings with special characters in `unicode.dart`:

```
String country = "Egypt";
String city = "Zürich";
String japanese = "日本語"; // nihongo meaning 'Japanese'

void main() {
    print('Unicode escapes: \uFE18'); // the ☐ symbol
    print(country[0]); // E
    print(country.codeUnitAt(0)); // 69
    print(country.codeUnits); // [69, 103, 121, 112, 116]
    print(country.runes.toList()); // [69, 103, 121, 112, 116]
    print(new String.fromCharCode(69)); // E
    print(new String.fromCharCodes([69, 103, 121, 112, 116])); // Egypt
    print(city[1]); // ü
    print(city.codeUnitAt(1)); // 252
    print(city.codeUnits); // [90, 252, 114, 105, 99, 104]
    print(city.runes.toList()); // [90, 252, 114, 105, 99, 104]
    print(new String.fromCharCode(252)); // ü
    print(new String.fromCharCodes([90, 252, 114, 105, 99, 104])); // Zürich
    print(japanese[0]); // 日
    print(japanese.codeUnitAt(0)); // 26085
    print(japanese.codeUnits); // [26085, 26412, 35486]
    print(japanese.runes.toList()); // [26085, 26412, 35486]
    print(new String.fromCharCode(35486)); // 語
    print(new String.fromCharCodes([26085, 26412, 35486]));
// 日本語
}
```

How it works...

Within a string, Unicode characters can be escaped by using \u. The index operator [] on a string gives you the string representation of the UTF-16 code unit. These are also accessible as integers representing code points (also called runes) through the `codeUnitAt()` or `codeUnits` methods. The static member `charCode(s)` can take UTF-16 code units or runes. They work in the following way; if the char-code value is 16 bits (a single UTF-16 code unit), it is copied literally. Otherwise, it is of length 2 and the code units form a surrogate pair.

There's more...

The `dart:convert` code contains a UTF-8 encoder/decoder that transforms between strings and bytes. The `utf8encoding.dart` file shows how you can use these methods, as shown in the following code:

```
import 'dart:convert' show UTF8;

String str = "AcciÃ³n"; // Spanish for 'Action'

void main() {
  List<int> encoded = UTF8.encode(str);
  print(encoded); // [65, 99, 99, 105, 195, 179, 110]
  // The UTF8 code units are reinterpreted as
  // Latin-1 code points (a subset of Unicode code points).
  String latin1String = new String.fromCharCodes(encoded);
  print(latin1String); // AcciÃ³n
  print(latin1String.codeUnits);
  // [65, 99, 99, 105, 195, 179, 110]
  var string = UTF8.decode(encoded);
  print(string); // AcciÃ³n
}
```

Using complex numbers

Dart has no built-in type for complex numbers, but it is easy to build your own. The `complex_numbers` library (based on similar libraries by Tiago de Jesus and Adam Singer) provides constructors, utility methods, and four arithmetic operations both defined as operators and static methods.

How to do it...

We now define a `ComplexNumber` class, containing all utility methods for normal usage:

```
library complex_numbers;

import 'dart:math' as math;

class ComplexNumber {
    num _real;
    num _imag;

    // 1- Here we define different ways to build a complex number:
    // constructors:
    ComplexNumber([this._real = 0, this._imag = 0]);
    ComplexNumber.im(num imag) : this(0, imag);
    ComplexNumber.re(num real) : this(real, 0);

    // 2- The normal utility methods to get and set the real and
    // imaginary part, to get the absolute value and the angle, to      //
    compare two complex numbers:
    num get real => _real;
    set real(num value) => _real = value;

    num get imag => _imag;
    set imag(num value) => _imag = value;

    num get abs => math.sqrt(real * real + imag * imag);

    num get angle => math.atan2(imag, real);

    bool operator ==(other) {
        if (!(other is ComplexNumber)) {
            return false;
        }
        return this.real == other.real && this.imag == other.imag;
    }

    String toString() {
        if (_imag >= 0) {
            return '${_real} + ${_imag}i';
        }
        return '${_real} - ${-_imag.abs()}i';
    }
}
```

```
// 3- operator overloading:  
// The basic operations for adding, multiplying, subtraction and      //  
division are defined as overloading of the operators +, *, - and /  
ComplexNumber operator +(ComplexNumber x) {  
    return new ComplexNumber(_real + x.real, _imag + x.imag);  
}  
  
ComplexNumber operator -(var x) {  
    if (x is ComplexNumber) {  
        return new ComplexNumber(this.real - x.real, this.imag - x.imag);  
    } else if (x is num) {  
        _real -= x;  
        return this;  
    }  
    throw 'Not a number';  
}  
  
ComplexNumber operator *(var x) {  
    if (x is ComplexNumber) {  
        num realAux = (this.real * x.real - this.imag * x.imag);  
        num imagAux = (this.imag * x.real + this.real * x.imag);  
  
        return new ComplexNumber(realAux, imagAux);  
    } else if (x is num) {  
        return new ComplexNumber(this.real * x, this.imag * x);  
    }  
    throw 'Not a number';  
}  
  
ComplexNumber operator /(var x) {  
    if (x is ComplexNumber) {  
        num realAux = (this.real * x.real + this.imag * x.imag) / (x.real *  
x.real + x.imag * x.imag);  
        num imagAux = (this.imag * x.real - this.real * x.imag) / (x.real *  
x.real + x.imag * x.imag);  
        return new ComplexNumber(realAux, imagAux);  
    } else if (x is num) {  
        return new ComplexNumber(this.real / x, this.imag / x);  
    }  
    throw 'Not a number';  
}  
  
// 4- Here we define the same operations as methods:  
static ComplexNumber add(ComplexNumber c1, ComplexNumber c2) {
```

```
num rr = c1.real + c2.real;
num ii = c1.imag + c2.imag;
return new ComplexNumber(rr, ii);
}

static ComplexNumber subtract(ComplexNumber c1, ComplexNumber c2)
{
    num rr = c1.real - c2.real;
    num ii = c1.imag - c2.imag;
    return new ComplexNumber(rr, ii);
}

static ComplexNumber multiply(ComplexNumber c1, ComplexNumber c2)
{
    num rr = c1.real * c2.real - c1.imag * c2.imag;
    num ii = c1.real * c2.imag + c1.imag * c2.real;
    return new ComplexNumber(rr, ii);
}

static ComplexNumber divide(ComplexNumber c1, ComplexNumber c2)
{
    num real = (c1.real * c2.real + c1.imag * c2.imag) / (c2.real *
    c2.real + c2.imag * c2.imag);
    num imag = (c1.imag * c2.real - c1.real * c2.imag) / (c2.real *
    c2.real + c2.imag * c2.imag);
    return new ComplexNumber(real, imag);
}
```

How it works...

The ComplexNumber class is built using standard Dart functionalities:

- ▶ Private getters for real and imaginary parts to return their values and setters to change them
- ▶ A constructor with two optional arguments and two named constructors for a complex number, respectively, without real or imaginary parts
- ▶ Some utility methods such as `toString()` and overloading of `==`
- ▶ Operator overloading for `+`, `-`, `*`, and `/`
- ▶ The same operations implemented as static methods taking two complex numbers

There's more...

Keep an eye on the pub package `math-expressions` by Frederik Leonhardt, as evaluation of expressions with complex numbers is one of its goals.

Creating an enum

Enum does not exist in Dart as a built-in type. Enums provide additional type checking and thus, help enhance code maintainability. So what alternative do we have? Look at the code in project `enum`, where we want to differentiate the degree of an issue reported to us (we distinguish between the following levels: TRIVIAL, REGULAR, IMPORTANT, and CRITICAL).

How to do it...

The first way to achieve the creating an enum functionality is shown in `enum1.dart`:

```
class IssueDegree {
    final _value;
    const IssueDegree(this._value);
    toString() => 'Enum.${_value}';

    static const TRIVIAL = const IssueDegree('TRIVIAL');
    static const REGULAR = const IssueDegree('REGULAR');
    static const IMPORTANT = const IssueDegree('IMPORTANT');
    static const CRITICAL = const IssueDegree('CRITICAL');
}

void main() {
    var issueLevel = IssueDegree.IMPORTANT;
    // Warning and NoSuchMethodError for IssueLevel2:
    // There is no such getter ALARM in IssueDegree
    // var issueLevel2 = IssueDegree.ALARM;

    switch (issueLevel) {
        case IssueDegree.TRIVIAL:
            print("Ok, I'll sort it out during lunch");
            break;
        case IssueDegree.REGULAR:
            print("We'll assign it to Ellen, our programmer");
            break;
        case IssueDegree.IMPORTANT:
```

```
        print("Let's discuss it in a meeting tomorrow morning");
        break;
    case IssueDegree.CRITICAL:
        print('Warn the Boss!');
        break;
    }
}
```

This snippet prints **Let's discuss it in a meeting tomorrow morning**.

An alternative way, shown in `enum2.dart`, is to define the enum behavior in an abstract class and then to implement that, as shown in the following code:

```
import 'enum_abstract_class.dart';

class IssueDegree<String> extends Enum<String> {

    const IssueDegree(String val) : super (val);

    static const IssueDegree TRIVIAL = const IssueDegree('TRIV');
    static const IssueDegree REGULAR = const IssueDegree('REG');
    static const IssueDegree IMPORTANT = const IssueDegree('IMP');
    static const IssueDegree CRITICAL = const IssueDegree('CRIT');

    main() {
        assert(IssueDegree.REGULAR is IssueDegree);
        // switch code
    }
}
```

The switch code of the first example also works for this implementation. To simplify the code, the `const` values can also be defined outside the class, as in `enum3.dart`. Then, it is no longer needed to precede them with the enum class name, as shown in the following code:

```
import 'enum_abstract_class.dart';

const IssueDegree TRIVIAL = const IssueDegree('TRIV');
const IssueDegree REGULAR = const IssueDegree('REG');
const IssueDegree IMPORTANT = const IssueDegree('IMP');
const IssueDegree CRITICAL = const IssueDegree('CRIT');

class IssueDegree<String> extends Enum<String> {
    const IssueDegree(String val) : super (val);
}

main() {
```

```
assert(REGULAR is IssueDegree);

var issueLevel = IMPORTANT;
switch (issueLevel) {
  case TRIVIAL:
    print("Ok, I'll sort it out during lunch");
    break;
  // rest of the code
}
```

How it works...

The first option uses an `enum` class with a `const` constructor to set a private `_value`; the class contains the different values as constants. The constants can only be defined inside the class, and you get autocompletion (in Dart Editor or other editors with the Dart plugin) for them for free! In this way, you can use this enum-like class in a switch, and both `dartanalyzer` and the runtime point out the error to you if a non-existent value is used. The `enum_class.dart` file provides the template code for this case; make sure you create the constant values, as shown in the following code:

```
class Enum {
  final _value;
  const Enum(this._value);
  toString() => 'Enum.${_value}';

  static const VAL1 = const Enum('VAL1');
  static const VAL2 = const Enum('VAL2');
  static const VAL3 = const Enum('VAL3');
  static const VAL4 = const Enum('VAL4');
  static const VAL5 = const Enum('VAL5');
}
```

The second way uses an abstract class `Enum` (defined in `enum_abstract_class.dart`) that takes a generic parameter `<T>`, as shown in the following code:

```
abstract class Enum<T> {
  final T _value;
  const Enum(this._value);
  T get value => _value;
}
```

Making the values top-level constants simplifies the code.

There's more...

The Ecma TC52 Dart Standards Committee has investigated a proposal for enums that will be discussed in September 2014 (refer to <http://www.infoq.com/news/2014/07/ecma-dart-google>), so providing built-in support for enums probably will be implemented in a future Dart version.

Flattening a list

A list can contain other lists as elements. This is effectively a two-dimensional list or array. Flattening means making a single list with all sublist items contained in it. Take a look at the different possibilities in `flatten_list.dart`.

How to do it...

We show three ways to flatten a list in the following code:

```
List lst = [[1.5, 3.14, 45.3], ['m', 'pi', '7'], [true, false, true]];
// flattening lst must give the following resulting List flat:
// [1.5, 3.14, 45.3, m, pi, 7, true, false, true]

void main() {
    // 1- using forEach and addAll:
    var flat = [];
    lst.forEach((e) => flat.addAll(e));
    print(flat);
    // 2- using Iterable.expand:
    flat = lst.expand((i) => i).toList();
    // 3- more nesting levels, work recursively:
    lst = [[1.5, 3.14, 45.3], ['m', 'pi', '7'], "Dart", [true, false,
true]];
    print(flatten(lst));
}
```

How it works...

The simplest method uses a combination of `forEach` and `addAll`. The second method uses the fact that `List` implements `Iterable`, and so has the `expand` method. The `expand` method is used here with an identity function as its argument; every element is returned without applying a function.

Using `expand` does not work if the list contains `ints` (or `Strings`, `doubles`, and so on) as single list elements, or if there are multiple levels of nesting. In that case, we will have to work recursively, as implemented in the `flatten` method:

```
Iterable flatten(Iterable iterable)
=> iterable.expand((e) => e is List ? flatten(e) : [e]);
```

There's more...

Why would you want to flatten a list of lists? There may be application needs to do this, for example, when you use two-dimensional lists or matrices in the game logic, but an obvious reason is that working with a list of lists is much more expensive performance wise.

Generating a random number within a range

You may have often wondered how to generate a random number from within a certain range. This is exactly what we will look at in this recipe; we will obtain a random number that resides in an interval between a minimum (`min`) and maximum (`max`) value.

How to do it...

This is simple; look at how it is done in `random_range.dart`:

```
import 'dart:math';

var now = new DateTime.now();
Random rnd = new Random();
Random rnd2 = new Random(now.millisecondsSinceEpoch);

void main() {
    int min = 13, max = 42;
    int r = min + rnd.nextInt(max - min);
    print("$r is in the range of $min and $max"); // e.g. 31
    // used as a function nextInt:
    print("${nextInt(min, max)}"); // for example: 17

    int r2 = min + rnd2.nextInt(max - min);
    print("$r2 is in the range of $min and $max"); // e.g. 33
}
```

How it works...

The Random class in `dart:math` has a method `nextInt(int max)`, which returns a random positive integer between 0 and max (not included). There is no built-in function for our question but it is very easy, as shown in the previous example. If you need this often, use a function `nextInter` for it, as shown in the following code:

```
int nextInter(int min, int max) {  
    Random rnd = new Random();  
    return min + rnd.nextInt(max - min);  
}
```

The variable `rnd2` shows another constructor of `Random`, which takes an integer as a seed for the pseudo-random calculation of `nextInt`. Using a seed makes for better randomness, and should be used if you need many random values.

Getting a random element from a list

For certain applications such as games, it is necessary to have a means to retrieve a random element from a collection in Dart. This recipe will show you a simple way to do this.

How to do it...

This is easy to do; refer to the `random_list.dart` file:

```
import 'dart:math';  
  
Random rnd = new Random();  
var lst = ['Bill', 'Joe', 'Jennifer', 'Louis', 'Samantha'];  
  
void main() {  
    var element = lst[rnd.nextInt(lst.length)];  
    print(element); // e.g. 'Louis'  
    element = randomListItem(lst);  
    print(element); // e.g. 'Samantha'  
}
```

How it works...

We generate a random index number based on the list length and use it to retrieve a random element from the list. If you need this often, use the one-line function `randomListItem` for it, as shown in the following code:

```
randomListItem(List lst) => lst[rnd.nextInt(lst.length)];
```

See also

- ▶ Consult the previous recipe for more information about the use of Random

Working with dates and times

Proper date-time handling is needed in almost every data context. What does Dart give us to ease working with dates and times? Dart has the excellent built-in classes `DateTime` and `Duration` in `dart:core`. As a few of its many uses, you can do the following:

- ▶ Compare and calculate with date times
- ▶ Get every part of a date-time
- ▶ Work with different time zones
- ▶ Measure timespans with `Stopwatch`

However, the `DateTime` class does not provide internationalization; for this purpose, you need to use the `intl` package from the Dart team.

How to do it...

The following are some useful techniques (try them out in `date_time.dart`):

- ▶ Formatting dates (from `DateTime` to a string) to standard formats, but also to any format using the package `intl`, as shown in the following code:

```
import 'package:intl/intl.dart';
import 'package:intl/date_symbol_data_local.dart';

print(now.toIso8601String()); // 2014-05-08T14:03:21.238
print(now.toLocal()); // 2014-05-08 14:03:21.238
print(now.toString()); // 2014-05-08 14:03:21.238
print(now.toUtc()); // 2014-05-08 12:03:21.238Z
// using intl to format:
var formatter = new DateFormat('yyyy-MM-dd');
String formatted = formatter.format(now);
print(formatted); // 2014-05-08
print(new DateFormat("EEEE").format(now)); // Thursday
print(new DateFormat("yMMMMEEEEd").format(now)); // Thursday, May 8, 2014
print(new DateFormat("y-MM-E-d").format(now)); // 2014-05-Thu-8
print(new DateFormat("jms").format(now)); // 2:19:08 PM
```

```
    print(new DateFormat('dd/MMM/y HH:mm:ss').format(now));      //  
08/May/2014 14:39:07  
    // locale data:  
    initializeDateFormatting("fr_FR", null).then(formatDates);  
    // ...  
}  
  
formatDates (var d) {  
    print(new DateFormat("EEEE", 'fr_FR').format(now));      //  
jeudi  
    print(new DateFormat("yMMMMEEEEd", 'fr_FR').format(now)); //  
jeudi 8 mai 2014  
    print(new DateFormat("y-MM-E-d", 'fr_FR').format(now));   //  
2014-05-jeu.-8  
}
```

- ▶ Parsing dates (from a string to a `DateTime`) when the given string is not in one of the acceptable date formats; an exception is thrown and caught as shown in the following code:

```
try {  
    DateTime dt = DateTime.parse("2014-05-08T15+02:00");  
} on FormatException catch(e) {  
    print('FormatException: $e');  
}
```

- ▶ Working with timezone information:

```
print(now.toLocal()); // time in local timezone  
print(now.toUtc()); // time in Coordinated Universal Time  
print(now.timeZoneName); // Romance (zomertijd)  
print(now.timeZoneOffset); // 2:00:00.000000
```

- ▶ Finding the last day of the month can be done as follows:

```
var date = new DateTime(2014, 6, 0);  
print(date.day); // 31  
// more general:  
var lastDayDateTime = (now.month < 12) ?  
    DateTime(now.year, now.month + 1, 0) : new  
    DateTime(now.year + 1, 1, 0);  
print(lastDayDateTime.day); // 31
```

How it works...

Formatting a `DateTime` instance can be done with a few `to...` methods from `dart:core`, such as `toLocal()` to get the time in the local time zones (as defined by your machine). However, `intl` gives you much more flexibility; make a `DateFormat` object with the specific format string as an argument, and then call the `format` method to get a formatted string back.

For example, `d` gives the day number, `E` gives the weekday in an abbreviated form, `EE` gives the full day's name, `M` gives the month number, `y` gives the year, `H` gives the hour (0-24), `j` gives the hour (0-12) with AM or PM, `m` gives the minute, and so on. Many format combinations are possible (and you can build your own).



For a complete overview of format combinations, refer to
<https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/intl/DateFormat>.

If you want formatting for a particular locale (such as `be_NL`, `cs_CZ`, `de_DE`, and so on), you must first load the specific locale data by importing `date_symbol_data_local.dart` and then calling `initializeDateFormatting`. The first parameter is the specific locale. If you give it the value `null`, all the available locale data is loaded, as shown in the following code:

```
initializeDateFormatting("fr_FR", null).then(formatDates);
```

Next, when constructing the `DateFormat` object, the locale (such as '`fr_FR`') has to be given as the second parameter:

```
new DateFormat("yMMMMEEEEd", 'fr_FR')
```

The `formatDates` method shows the following code in action:

```
formatDates (var d) {  
  print(new DateFormat("EEEE", 'fr_FR').format(now)); // jeudi  
  print(new DateFormat("yMMMMEEEEd", 'fr_FR').format(now)); // jeudi  
  8 mai 2014  
  print(new DateFormat("y-MM-E-d", 'fr_FR').format(now)); // 2014-  
  05-jeu.-8  
}
```

Parsing a well-formed string into a `DateTime` string is done with the static `DateTime.parse` method, which takes the string and produces `DateTime`. The input format must conform to an ISO 8601 format. If the input format cannot be parsed, `FormatException` is thrown, so use try/catch to handle this.

The `toUTC` option gives you the time in **Coordinated Universal Time (UTC)** format, which is for all practical purposes identical to **Greenwich Mean Time (GMT)**. Use `timeZoneName` to get an abbreviated name of the time zone for the `DateTime` object. The difference between UTC and the time zone of a `DateTime` object is calculated by calling `timeZoneOffset`.

To find the last day of the month, giving a day value of zero for the next month returns the previous month's last day.

There's more...

Be aware that a `DateTime` object is always in the local time zone, unless explicitly created in the UTC time zone with the `DateTime.utc` constructor; this can be done as follows:

```
DateTime moonLanding = new DateTime.utc(1969, DateTime.JULY, 20);
```

In this constructor, only the year is required, all the other date and time parts are optional.

If you have other Datetime questions, look up the API docs at <https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart-core.DateTime>.

Improving performance in numerical computations

Unlike Java and C#, who have dedicated 8, 16, 32, and 64 bit signed and unsigned integer types, and 32-bit and 64-bit floats, Dart does not have bounded integer types or 32-bit floating point number types; it only has two numeric types, `int` (an arbitrarily sized integer) and `double` (conforming to the IEEE-754 spec), and their super type `num`. This was done to make the language more dynamic and easier to learn and use. However, the Dart VM does a good job of inferring the range of integers, and optimizes whenever possible. Here, we provide a number of discussions and tips to give your code the highest performance possible when it involves numerical computing.

How to do it...

The VM uses three integer types internally and switches between them behind the scenes as numbers grow and shrink in size. They are as follows:

- ▶ `smi` (small integer): You can think of this integer type as being 32 bit on a 32-bit machine and 64 bit on a 64-bit machine
- ▶ `mint` (medium integer): This integer type is always 64 bit
- ▶ `bigint` (big integer): The machine's RAM is the limit for this integer type

You can't use these classes in the Dart code; the VM automatically promotes an integer from smi to mint, and to bigint when necessary. However, you can get better performance if you ensure that your integer values stay in smaller ranges. For example, refer to the `integer_promotion.dart` file:

```
import 'dart:math';

void main() {
    int points = 42; // starts as a smi
    print(points); // 42
    points = pow(points, 10); // becomes a mint
    print(points); // 17080198121677824
    points = pow(points, 3); // becomes a bigint
    print(points); // 49828603059827911309997192674768286219621357322
24
}
```

[ Don't use the bigint range unless you really have to because this will result in a performance hit. If possible, stay in the smi range; if not, use the mint range.]

Working with lists of numbers, numbers can be stored in an object `List()`, or in a generic type `List<int>` or `List<double>`. While the latter is better than the first, it is best to use Dart typed lists, which is available in the `dart:typed_data` package such as `UInt8List`, `Int64List`, or `Float32List`. So instead of using `List<double>`, which is always slower, use `Float32List` or `Float64List`. Be careful when using web apps; not every browser supports `typed_data` yet. For example, Internet Explorer 10 supports it, but version 9 does not.

[ Use typed lists when computing: calculations will run more efficiently.]

Taking into account these considerations, it is advisable to benchmark a few versions of your app with different number of type usages (for example, comparing `Float32List` with `Float64List`, or even with a normal list), in order to see which one performs best in the specific context of your app.

How it works...

Advice mostly arises from the way the Dart VM can optimize working with the different numerical types. Because JavaScript can't use the exact same optimizations, dart2js sometimes works differently; refer to the next section for special advice in this case.

Small machine integers (smis) fit in a register and can be loaded and stored directly in a field instead of being fetched from memory because they never require memory allocation. That's why they are fast. However, their range depends on the CPU architecture, so assume a 31-bit range unless you require more bits. Operations with bigints cannot be optimized by the VM. Working with doubles is very efficient because they are unboxed, that is, they don't need to be put on the heap.

Dart-typed lists can only store numbers and not null values. They behave as an array of bounded integer and float values, each entry containing its value, and the VM optimizes their usage. Typed lists are most of the time much more compact, providing better memory and CPU cache usage; for example, if you need only 8 bits of precision, use `Int8List`. They are also faster to process when **Garbage Collection (GC)** occurs because they never store object references. So they don't have to be scanned by the GC.

There's more...

From a language design point of view, using the `int` and `num` types is the best; the Dart VM experts would probably want us to work only with `double` because that is where most optimizations can be done.

Working with JavaScript

In JavaScript, every number is represented as `double` (it only knows the IEEE-754 type).

 If your app will be used in the compiled JavaScript form, it is better to use only the `num` type. Also, use `is num` instead of testing `is int` or `is double` because the Dart VM and dart2js behave differently here too. If you insist on using the `int` type, stay within the smi range.
In any case, use Dart-typed lists as they map trivially to JavaScript-typed arrays. However, avoid `Int64List` or `Uint64List` because dart2js does not support 64-bit integers due to lack of support of typed data in Internet Explorer 9 (causing a runtime exception if used).

Parsing numbers

Avoid extensive use of `double.parse(String s)` or even `int.parse(String s)` as at the time of writing their performance was not optimized.

See also

- ▶ Consult <http://dartogreniyorun.blogspot.be/2013/05/performance-optimization-and-dart.html>, where the use of typed data results in a performance increase by two times
- ▶ You might also want to consult the *Benchmarking your app* recipe in *Chapter 2, Structuring, Testing, and Deploying an Application*

Using SIMD for enhanced performance

A lot of modern CPUs and GPUs provide **Single Instruction Multiple Data (SIMD)** support. Four 32-bit data values (integers or floats) can be processed in parallel with the help of 128-bit special registers. This provides a potential speedup of 400 percent for image processing, 3D graphics, audio processing, and other numeric computation algorithms. Also, machine-learning algorithms (such as for automatic speech recognition) that use a **Gauss Mixture Model (GMM)** benefit from SIMD.

How to do it...

Dart lets you work with this feature by using the special SIMD x types from the `typed_data` library. It offers the following four types:

- ▶ `Int32x4`, which represents four 32-bit integer values
- ▶ `Float32x4`, which represents four single-precision floating point values
- ▶ List structures to contain the 32-bit integer values, such as `Int32x4List`
- ▶ `Float32x4List`, list structure to contain the 32-bit floating point values

Let's see some examples of SIMD operations in `simd.dart`; the different types are highlighted in the following snippet:

```
import 'dart:typed_data';

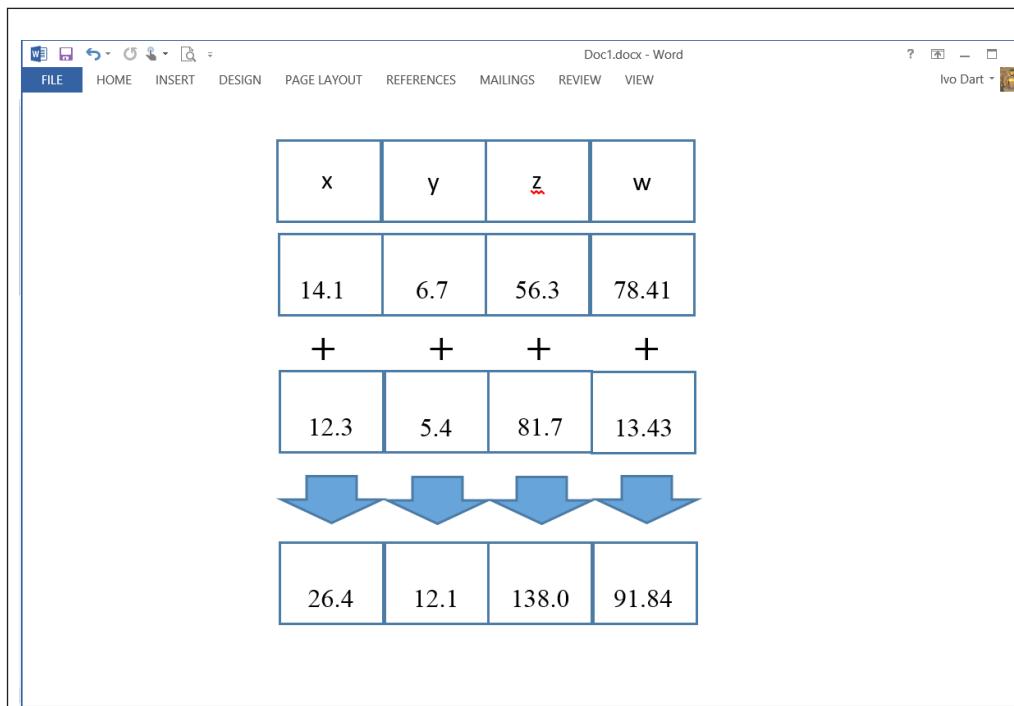
void main() {
    var a = new Float32x4(14.1, 6.7, 56.3, 78.41);
    var b = new Float32x4(12.3, 5.4, 81.7, 13.43);
    Float32x4 sum = new Float32x4.zero(); //
    print(sum); // [0.000000, 0.000000, 0.000000, 0.000000]
    sum = a + b;
    print(sum); // [26.400002, 12.100000, 138.000000, 91.840004]
    print(sum.z); // 138.0
    // b.y = 3.14; // --> NoSuchMethodError
    b = b.withY(3.14);
    print(b); // [12.300000, 3.140000, 81.699997, 13.430000]
    b = b.shuffle(Float32x4.WYXZ);
    print(b); // [13.430000, 3.140000, 12.300000, 81.699997]
    // a < b; // There is no such operator in Float32x4
    Int32x4 mask = a.greaterThan(b); // Create selection mask.
    Float32x4 c = mask.select(a, b); // Select.
    print(c); // [14.100000, 6.700000, 56.299999, 81.699997]
    // selectively applying an operation:
    Float32x4 v = new Float32x4(22.0, 33.0, 44.0, 55.0);
    // mask = [0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0x0]
    mask = new Int32x4.bool(true, true, false, true);
    // r = [4.0, 9.0, 16.0, 25.0].
    Float32x4 r = v - v;
    v = mask.select(r, v);
    print(v); // [0.000000, 0.000000, 44.000000, 0.000000]
}
```



If your application needs a list of `Float32` objects and you can deploy them on an SIMD platform, then be sure to use `Float32x4List` instead of `List<Float32x4>` to get better performance.

How it works...

The `Float32x4` object offers the standard set of arithmetic operations and more. A `Float32x4` object is in fact an immutable object with operations that create new immutable `Float32x4` objects. The `Int32x4` object is more limited, being useful for comparison, branching, and selection. In code that is optimized for these types, the values are mapped directly to SIMD registers, and operations on them compile into a single SIMD instruction with no overhead. You can think of an SIMD value as a horizontal compartment being subdivided into four lanes, respectively called **x**, **y**, **z**, and **w**, as shown in the following screenshot:



The SIMD architecture

An operation on two SIMD values happens on all the lanes simultaneously. With `.x` and other values, you can read the values of the individual lanes, but attention, this is slow. Because an SIMD value is immutable, a `b.y = value` statement is illegal. However, the `withX` methods let you do this, but again this is slow. Reordering values in one SIMD is done with a number of `shuffle` methods, where the lane order (WYXZ) indicates the new order. An SIMD instance contains four numbers, so comparisons such as `<` or `>=` cannot be defined. If you want to make `c` equal to the higher values of `a` and `b`, first you have to create a mask with the `greaterThan` operation, and then perform a `select` operation on it. An analogous masking technique is used if you want to perform an operation on some of the lanes only.

At this time, you can get this performance acceleration on all IA32/X64 platforms, and on ARM only if the processor supports NEON technology, and its implementation is pending for JavaScript. Thanks to the work of John McCutchan, Dart was the first web technology to use SIMD processing.

See also

- ▶ For more information on SIMD, refer to <http://en.wikipedia.org/wiki/SIMD>
- ▶ Dart can even beat the Java VM when using SIMD; for more information refer to <http://dartogreniyorum.blogspot.be/2013/05/dart-beats-java-in-numerical-computing.html>

4

Object Orientation

In this chapter, we will cover the following recipes:

- ▶ Testing and converting types
- ▶ Comparing two objects
- ▶ Using a factory constructor
- ▶ Building a singleton
- ▶ Using reflection
- ▶ Using mixins
- ▶ Using annotations
- ▶ Using the call method
- ▶ Using noSuchMethod
- ▶ Creating toJSON and fromJSON methods in your class
- ▶ Creating common classes for client and server apps

Introduction

Dart, although optionally typed, is an object-oriented language like Ruby, so it is an object. This is in contrast to JavaScript, where object orientation is not inherent in the language and was added later in different ad hoc ways. We can expect huge benefits from working with a real object-oriented language to build our client web applications. So, let's delve a bit deeper into the object-oriented nature of Dart, and find some new techniques and insights.

Testing and converting types

Everything in Dart is an object and has a type; an instance of a class descended from an object through a single inheritance chain, even null is of type Null. Type annotating a variable is not required in Dart, for example, contrary to Java. In this case, the variable is declared with var and is of type dynamic. Values sometimes have to be converted from one type to another. In order to avoid runtime type errors when the conversion fails, we can test if the value is of the type we want to cast it to before the conversion. For the code examples, refer to types.dart.

How to do it...

We can test and convert types with the help of the following steps:

1. We can test and show the type of an object, as shown in the following code:

```
void main() {
    var p = new Person();
    p.name = "Joe";
    if (p is Person) {
        print('p is called ${p.name}');
        print('p is of type ${p.runtimeType}');
    } else {
        // p has value null and is of type Null
        print('p has value $p and is of type ${p.runtimeType}');
    }
}

class Person {
    String name;
}
```

The preceding code prints **p is of type Person**.

2. Conversion between types of variables:

- ❑ To convert everything to a string, use the \$ operator in string interpolation or the `toString()` method
- ❑ To convert a string str to int, use `int.parse(str)`
- ❑ To convert a string str to double, use `double.parse(str)`
- ❑ To handle a possible `FormatException` in the previous two conversions, use `try / catch`
- ❑ To convert int to double, use `toDouble()`

- ❑ To convert `double` to `int`, use `toInt()`, but truncation of the decimal part will occur
- ❑ To convert a string to `DateTime`, use `DateTime.parse(str)` with `try / catch`
- ❑ To handle a possible `FormatException` when `str` is not a recognizable date and time format
- ❑ To convert `DateTime` to a string, use `toLocal()`, `toUtc()`, or a `DateFormat` object (refer to the *Working with dates and times* recipe in *Chapter 3, Working with Data Types*)
- ❑ To convert `bool` to `int`, use the `int toInt(bool val) => val ? 1 : 0;` function

How it works...

To test if an object `p` is of a certain type, use the `is` (or its negation `is!=`) operator. Because types are optional in Dart, you can only ask for `runtimeType` of an object, which is of type `Type`. The `p is Object` message is always true, and `null is T` (with `T` being a type) is only true if `T == null`.

The `toString()` method is a method of `Object` and returns **Instance of Type** by default, where **Type** is the class name of the object. Override this method to have more specific behaviors in your classes. In Dart, `bool` and `int` are two different types, but you can mimic C-like behavior by using a ternary operator in `toInt(bool)`.



Conversion between types is not always safe; use `try/catch` if a `FormatException` occurs.

There's more...

There is also a shorter version, the `is` test if `(p is Person) { p.name = 'Bob'; }` can be shortened to `(p as Person).firstName = 'Bob';` with the `as` operator being a typecast of `p` to `Person` is done. However, you can only use this form when you are certain that `p` can be of that type; when `p` is `null` or not a `Person` Dart will throw an exception.

See also

- ▶ Refer to the *Working with dates and times* recipe in *Chapter 3, Working with Data Types* for date and time conversions

Comparing two objects

How can you determine whether two objects are equal or not? Basically, this is defined by objects (refer to the *How it works...* section of this recipe). Obviously, two equal objects will be of the same class (so the same type) and have the same value(s).

How to do it...

In the `comparing_objects` program, we define a class `Person`, override `==` and `hashCode`, and test the equality of some objects, as shown in the following code:

```
void main() {
    var p1 = new Person("Jane Wilkins", "485-56-7861", DateTime.
        parse("1973-05-08"));
    var p2 = new Person("Barack Obama", "432-94-1282", DateTime.
        parse("1961-08-04"));
    var p3 = p1;
    var p4 = new Person("Jane Wilkins", "485-56-7861", DateTime.
        parse("1973-05-08"));

    // with == and hashCode from Object:
    // (comment out == and hashCode in class Person)
    print(p2==p1); //false: p1 and p2 are different
    print(p3==p1); //true: p3 and p1 are the same object
    print(p4==p1); //false: p4 and p1 are different objects
    print(identical(p1, p3)); //true
    print(identical(p1, p4)); //false
    print(p1.hashCode); // 998736967
    print(p2.hashCode); // 676682609
    print(p3.hashCode); // 998736967
    // with specific == and hashCode for class Person:
    print(p2==p1); //false: p1 and p2 are different
    print(p3==p1); //true: p3 and p1 are the same object
    print(p4==p1); //true: p4 and p1 are the same Person
    print(identical(p1, p3)); //true
    print(identical(p1, p4)); //false
    print(p1.hashCode); // 105660000000
    print(p2.hashCode); // -265428000000
    print(p3.hashCode); // 105660000000
}

class Person {
    String name;
```

```
String ssn; // social security number
DateTime birthdate;

Person(this.name, this.ssn, this.birthdate);
toString() => 'I am $name, born on $birthdate';
operator ==(Person other) => this.ssn == other.ssn;
int get hashCode => birthdate.millisecondsSinceEpoch;
}
```

How it works...

The `Object` superclass has a `bool ==(other)` method that defines the equality of objects. This returns `true` only when `this` and `other` are the same object, that is, they reference the same object in the (heap) memory. This is the default behavior, unless you override it in your classes. The `!=` operator is the negation of this. Another way to test this is the top-level `identical` function from `dart:core` `bool identical(Object a, Object b)`.

Every object has an integer `hashCode` returned by the getter `int get hashCode`. Two objects that are equal have the same `hashCode`, but this can differ between two runs of a program. Any subclass can have its own version to define equality between its objects.



If you override `==` in a class, you should override `hashCode` as well for consistency.

There's more...

If you want to use instances of a class as keys in maps, then you have to overload the `==` operator and the `hashCode` method.

See also

- ▶ Refer to the *Working with dates and times* recipe in *Chapter 3, Working with Data Types* for date and time conversions

Using a factory constructor

Dart gives us many flexible and succinct ways to build objects through constructors:

- ▶ With optional arguments, such as in the `Person` constructor where `salary` is optional:

```
class Person{  
    String name;  
    num salary  
    Person(this.name, {this.salary});  
}
```

- ▶ With named constructors (a bonus for readable self-documenting code), for example, where a `BankAccount` for the same owner as `acc` is created:

```
BankAccount.sameOwner(BankAccount acc): owner = acc.owner;
```

- ▶ With `const` constructors, as shown in the following code:

```
class ImmutableSquare {  
    final num length;  
    static final ImmutableSquare ONE = const ImmutableSquare(1);  
    const ImmutableSquare(this.length);  
}
```

However, modern modular software applications require more flexible ways to build and return objects, often extracted into a factory design pattern (for more information, refer to [http://en.wikipedia.org/wiki/Factory_\(object-oriented_programming\)](http://en.wikipedia.org/wiki/Factory_(object-oriented_programming))). Dart has this pattern built right into the language with factory constructors.

How to do it...

In the `factory` program, we explore some usage examples as follows:

1. Returning an object from a cache, as shown in the following code:

```
main() {  
    var sv = new Service('Credit Card Validation');  
    sv.serve('Validate card number');  
    Person p = new Person("S. Hawking");  
    print(p); // S. Hawking  
}  
  
class Service {  
    final String name;  
    bool mute = false;
```

```
// _cache is library-private, thanks to the _ in front of its
name.
static final Map<String, Service> _cache = <String, Service>{};

Service._internal(this.name);

factory Service(String name) {
    if (_cache.containsKey(name)) {
        return _cache[name];
    } else {
        final serv = new Service._internal(name);
        _cache[name] = serv;
        return serv;
    }
}

void serve(String msg) {
    if (!mute) {
        print(msg); // Validate card number
    }
}
```

2. Creating an object from a subtype, as shown in the following code:

```
class Person {
    factory Person(name) => new Teacher(name);
}

class Teacher implements Person {
    String name;
    Teacher(this.name);
    toString() => name;
}
```

Using an abstract class, `abstract_factory1-3` shows you how to use a factory constructor with it:

- The factory constructor can be used with an abstract class, as shown in the following code:

```
void main() {
    // factory as a default implementation of an abstract class:
    Cat cat = new Animal();
    var catSound = cat.makeNoise();
```

```
        print(catSound); // Meow
    }

abstract class Animal {
    String makeNoise();
    factory Animal() => new Cat();
}

class Cat implements Animal { String makeNoise() => 'Meow'; }
class Dog implements Animal { String makeNoise() => 'Woef'; }
```

- ▶ Another example of using the factory constructor with an abstract class is shown in the following code:

```
import 'dart:math';

void main() {
    Cat an = new Animal();
    print(an.makeNoise());
}

abstract class Animal {
    // simulates computation:
    factory Animal() {
        var random = new Random();
        if (random.nextBool())
            return new Cat();
        else
            return new Dog();
    }
}

class Cat implements Animal { String makeNoise() => 'Meow'; }
class Dog implements Animal { String makeNoise() => 'Woef'; }
```

- ▶ The next example also illustrates how to use a factory constructor with an abstract class:

```
void main() {
    Cat cat = new Animal("cat");
    Dog dog = new Animal("dog");
    print(cat.makeNoise());
}

abstract class Animal {
```

```
String makeNoise();  
factory Animal(String type) {  
    switch(type) {  
        case "cat":  
            return new Cat();  
        case "dog":  
            return new Dog();  
        default:  
            throw "The '$type' is not an animal";  
    }  
}  
  
class Cat implements Animal { String makeNoise() => 'Meow'; }  
class Dog implements Animal { String makeNoise() => 'Woef'; }
```

How it works...

In the first example, we had a number of services gathered in `_cache`, but each service can be created only once through a private `_internal` constructor; every named service is unique.

In the second example, we showed that a class with a factory constructor cannot be directly extended; instead, the subtype must implement the class. In the third example, we showed three variants of using an abstract class with a factory constructor.

Why and when would you want to use a factory constructor? Sometimes, we don't want a constructor to always make a new object of the class it is in. The following are some use cases for a factory constructor; the code examples show you a usage example for all of them:

- ▶ To return an object from a cache in order to reuse it
- ▶ To create an object from a subtype of the class the constructor is in
- ▶ To limit the instances to one unique object (the singleton pattern)
- ▶ Even an abstract class can contain a factory constructor to return a default implementation of a concrete class (this is the only way an abstract class can have a constructor)

The factory constructor is invoked just as any other constructor by `new`. The consumer of the class doesn't know the constructor is really a factory, so you can refactor regular constructors into factory constructors to enhance flexibility without forcing clients to change their code. The factory could also involve much more preparation and computation, and the consumer of the class may not be aware of it; the consumer may just create a new instance. A factory invoking a private constructor is also a common pattern, as we saw in the first example.



The keyword `this` cannot be used inside a factory constructor because the constructor has no access to it.

There's more...

The factory constructor is also extensively used in standard libraries, for example, the DOM type `CustomEvent` only has factory constructors. This is because the browser has to produce these instances; the Dart object is just a wrapper. Another use case in Dart can be that you want to abstract an implementation of a certain feature. Some browsers support it natively, while others don't. You can then look into the factory to see whether the browser can handle it and then choose the right implementation according to the capabilities of the browser.

Building a singleton

In some cases, you only need one unique instance of a class because this is simply enough for the app you're working with, or perhaps to save resources. This recipe shows you how to do this in Dart.

How to do it...

The singleton example shows how to do this (substitute your singleton class name for `Immortal`). Use a factory constructor to implement the singleton pattern, as shown in the following code:

```
class Immortal {  
  static final Immortal theOne = new Immortal._internal('Connor  
MacLeod');  
  String name;  
  factory Immortal(name) => theOne;  
  // private, named constructor  
  Immortal._internal(this.name); }  
  
main() {  
  var im1 = new Immortal('Juan Ramirez');  
  var im2 = new Immortal('The Kurgan');  
  print(im1.name);           // Connor MacLeod  
  print(im2.name);           // Connor MacLeod  
  print(Immortal.theOne.name); // Connor MacLeod  
  assert(identical(im1, im2));  
}
```

All `Immortal` instances are the same object.

How it works...

The `Immortal` class contains an object of its own type, which instantiates itself by calling the private `_internal` constructor. Because it will be unique, we declare it as static. The factory constructor always returns this instance; only one instance of the `Singleton` class (named `Immortal` here) can ever exist in the executing isolate. It has to be a factory constructor because only this type can return a value. The code can even be shortened, shown as follows:

```
class Singleton {  
    factory Singleton() => const Singleton._internal_();  
    const Singleton._internal_();  
}
```

Using reflection

The Dart mirror-based reflection API (contained in the `dart:mirrors` library) provides a powerful set of tools to reflect on code. This means that it is possible to introspect the complete structure of a program and discover all the properties of all the objects. In this way, methods can be invoked reflectively. It will even become possible to dynamically evaluate code that was not yet specified literally in the source code. An example of this would be calling a method whose name was provided as an argument because it is looked up in the database table.

Getting ready

The part of your code that uses reflection should have the following import code:

```
import 'dart:mirrors';
```

How to do it...

To perform reflection we perform the following actions:

- ▶ In the reflection project, we use a class `Embrace` to reflect upon:

```
void main() {  
    var embr = new Embrace(5);  
    print(embr); // Embraceometer reads 5  
    embr.strength += 5;  
    print(embr.toJson()); // {strength: 10}  
    var embr2 = new Embrace(10);  
    var bigHug = embr + embr2;  
    // Start reflection code:  
}
```

- ▶ Use of MirrorSystem, as shown in the following code:

```
final MirrorSystem ms = currentMirrorSystem();
// Iterating through libraries
ms
    .libraries
    .forEach((Uri name, LibraryMirror libMirror) {
        print('$name $libMirror');
    });
}
```

- ▶ Use of InstanceMirror and ClassMirror, as shown in the following code:

```
InstanceMirror im = reflect(embr);
InstanceMirror im2 = im.invoke(#toJson, []);
print(im2.reflectee); // {strength: 10}
ClassMirror cm = reflectClass(Embrace);
ClassMirror cm2 = im.type;
printAllDeclarationsOf(cm);
InstanceMirror im3 = cm.newInstance(#light, []);
print(im3.reflectee.strength);
im3.reflectee.withAffection();
}

printAllDeclarationsOf(ClassMirror cm) {
    for (var k in cm.declarations.keys)  print(MirrorSystem.
getName(k));
print(MirrorSystem.getName(m.simpleName));
}

class Embrace  {
    num _strength;
    num get strength => _strength;
    set strength(num value) => _strength=value;
    Embrace(this._strength);
    Embrace.light(): _strength=3;
    Embrace.strangle(): _strength=100;
    Embrace operator +(Embrace other) => new
    Embrace(strength + other.strength);
    String toString() => "Embraceometer reads $strength";
    Map toJson() => {'strength': '$_strength'};

    withAffection()  {
        for (var no=0; no <= 3; no++)  {
            for (var s=0; s <=5; s++) { strength = s; }
        }
    }
}
```

- ▶ Running the previous program produces the following output:

Embraceometer reads 5

{strength: 10}

dart:core LibraryMirror on 'dart.core'

dart:mirrors LibraryMirror on 'dart.mirrors'

dart:nativewrappers LibraryMirror on ''

dart:typed_data LibraryMirror on 'dart.typed_data'

dart:async LibraryMirror on 'dart.async'

dart:convert LibraryMirror on 'dart.convert'

dart:collection LibraryMirror on 'dart.collection'

dart:_internal LibraryMirror on 'dart._internal@0x1f109d24'

dart:isolate LibraryMirror on 'dart.isolate'

dart:math LibraryMirror on 'dart.math'

dart:builtin LibraryMirror on 'builtin'

dart:io LibraryMirror on 'dart.io'

file:///F:/Dartiverse/ADartCookbook/book/Chapter 4 - Object orientation/code/reflection/bin/reflection.dart LibraryMirror on ''

{strength: 10}

_strength

strength

strength=

+

toString

toJson

withAffection

Embrace

Embrace.light

Embrace.strangle

3

How it works...

The `currentMirrorSystem` class returns a `MirrorSystem` object on the current isolate; the `libraries` getter gives you the list of libraries in the scope of the current code.

The `InstanceMirror` subclass is a representation of an instance of an object and `ClassMirror` is the representation of the class definition.

Use the top-level `reflect` method on an object to get `InstanceMirror`. This allows you to dynamically invoke code on the object producing another `InstanceMirror`; using its `reflectee` property gives you access to the actual instance.

Note that the `invoke` method takes as its first argument a symbol (recognizable from its `#` prefix) for the method name. Symbols were introduced in Dart because they survive minification.

The top-level `reflectClass` method on a class results in `ClassMirror`; the same type of object is given by calling `type` on `InstanceMirror` of that class. The `ClassMirror` class has a `declarations` getter that returns a map from the names of the declarations to the mirrors on them. Static methods can be called on `ClassMirror`.

For every type of object in Dart, there exists a corresponding mirror object. So we have `VariableMirror`, `MethodMirror`, `ClassMirror`, `LibraryMirror`, and so on. Invoking `newInstance` on a `ClassMirror` class with the name of a constructor as a symbol produces `InstanceMirror`; you can then call methods on the real object via `reflectee`.

There's more...

There are some things we should be aware of when using reflection:

- ▶ The mirror API is still evolving, so expect some additions and adjustments in the future. The implementation is most complete for code running in the Dart VM.
- ▶ Mirroring in dart2js lags a bit behind. The processes of minifying and tree shaking your app performed by dart2js will generally not detect the reflected code. So the use of reflection at runtime might fail, resulting in `NoSuchMethod()` errors. To prevent this from happening, use the `Mirrors` annotation, as shown in the following code, which helps the dart2js compiler to generate a smaller code:

```
@MirrorsUsed(override: '*')
import 'dart:mirrors';
```
- ▶ One of the restrictions is the reflections across isolates. At the time of writing this book, reflection only works if the reflection code and the object being reflected are running in the same isolate.

- ▶ Suppose you have an undocumented method that returns a Future value and you want to know the properties and methods of that object without digging into the source code. Run the following code snippet:

```
import 'dart:mirrors';

undocumentedMethod().then((unknown) {
    var r = reflect(unknown).type; // ClassMirror
    var m = r.declarations;
    for (var k in m.declarations.keys) print(MirrorSystem.getName(k));
});
```

See also

- ▶ When you want to use reflection in the code, which has to be minified and tree shaken, read the *Shrinking the size of your app* recipe in Chapter 1, *Working with Dart Tools*

Using mixins

In Dart, just like in Ruby, your classes can use mixins to assign a certain behavior to your class. Say an object must be able to store itself, so its class mixes in a class called `Persistable` that defines `save()` and `load()` methods. From then on, the original class can freely use the mixed-in methods. The mechanism is not used for specialized subclassing or is-a relationships, so it doesn't use inheritance. This is good because Dart uses a single inheritance, so you want to choose your unique direct superclass with care.

How to do it...

- ▶ Look at the `mixins` project; the `Embrace` class from the previous recipe needs to persist itself, so it mixes with the abstract class `Persistable`, thereby injecting the `save` and `load` behavior. Then, we can apply the `save()` method to the `embr` object, thereby executing the code of the mixin as follows:

```
void main() {
    var embr = new Embrace(5);

    print(embr.save(embr.strength));
    print(embr is Movement); // true
    print(embr is Persistable); // true
}
```

- ▶ Mixing in `Persistable`, as shown in the following code:

```
class Embrace extends Movement with Persistable {  
    // code omitted, see previous recipe  
}  
  
class Movement {  
    String name;  
    Movement();  
}
```

- ▶ Defining `Persistable`, as shown in the following code:

```
abstract class Persistable {  
    save(var s) {  
        // saving in data store  
        return "You are saved with strength $s!";  
    }  
    load() => "You are loaded!";  
}
```

The previous program produces the output **You are saved with strength 5!**

How it works...

The abstract class `Persistable` is mixed in with the keyword `with`. The class `Embrace` is a subclass of class `Movement`, `Embrace` is `Movement`; if you don't have a direct superclass, as shown in the following code:

```
class Embrace extends Object with Persistable
```

So it means that in order to use a mixin, you will always need to extend a class. Objects of the class `Embrace` are also of the mixed-in type `Persistable`.

There's more...

The class that is mixed in is usually an abstract class, but it doesn't have to be. You can also mix in several classes to give a taste of multiple inheritance, as shown in the following code:

```
class Developer extends Person with Intellectual, Addicted
```

The mixed-in class has to obey some restrictions as follows:

- ▶ It must not declare a constructor
- ▶ Its superclass is an object
- ▶ It may not contain calls to the superclass

The mixin concept very much resembles the implementation of an interface mechanism, which exists in many other languages. However, it is much more powerful because the mixed in class(es) can contain real code that can be executed, whereas interfaces can't contain code; only definitions of methods. So that's why using mixins is a good practice.



Always examine your inheritance relationship to see if it can be better described by a mixin than with a superclass.

In Dart, you can also use the `implements` keyword; a class can implement one or more other classes. What's the difference between implementing and mixing in? Implementing means that the class provides its own code for the public methods from the class it implements, while mixing in means that the class can use the code from the mixin class itself.

Using annotations

Dart shares with other languages such as Java and C# the ability to attach (or annotate) variables, classes, functions, methods, and other Dart program structures with metadata words preceded by an @ sign. This is done to give more information about the structure, or indicate that it has a special characteristic or behavior. Examples are `@override`, `@deprecated`, and `@observable` (used in Polymer), so they are liberally used by the Dart team. Also, `Angular.dart` uses them abundantly. Moreover, you can also define your own annotations.

How to do it...

In the project annotations, we gave our `Embrace` class the metadata `@ToFix`. The `strangle` method is denoted by `@deprecated`, and we indicate with `@override` in `Embrace` that we want to override the method `consumedCalories` inherited from `Movement`, as shown in the following code:

```
const Anno = "Meta";

void main() {
    var embr = new Embrace(5);
    print(embr);
    var str = new Embrace. strang();
}

@Anno
@ToFix("Improve the algorithms", "Bill Gates")
class Embrace {
    // code omitted, see previous recipes
```

```
@deprecated  
Embrace.strangle(): _strength = 100;  
  
@override consumedCalories() { } // warning!  
  
}  
  
class Movement {  
  String Name;  
  Movement();  
  consumedCalories() {  
    // calculation of calories  
  }  
}  
  
class ToFix {  
  final String note, author, date;  
  const ToFix(this.note, this.author, {this.date});  
}
```

How it works...

The `@deprecated` instance is used to indicate something that you no longer want users of your library to use, and that will probably stop working in a future version. When the analyzer in Dart Editor sees this annotation, it marks the code component that follows (and everywhere it is used) with a strike-through line. Moreover, it will give a warning: **'...' is deprecated**. The `@override` instance is also a good (but not necessary) indication that you want to override an inherited behavior. Here, the editor also uses this instance to point to possible bugs; if you had written `@override consumedcalories()`, then you would get the warning Method does not override an inherited method; so typos are eliminated.

To make your own annotations, you must make sure that it is defined as a constant expression that starts with an identifier, such as `const Anno` in the example, which could be used as `@Anno`. More specifically, it must be a reference to a compile-time constant variable or call to a constant constructor. We used the class `ToFix` to give our class the annotation `@ToFix("Improve the algorithms")`.

There's more...

Annotations are defined in Dart through the class `Annotation` in the `analyzer` library. With the reflection mirror library (see the *Using reflection* recipe), it is possible to extract metadata at runtime and use its values to influence program execution.

See also

- ▶ See the annotations used in `Angular.dart` in *Chapter 11, Angular Dart Recipes*

Using the call method

This is a hidden gem in Dart. It enables you to give a parameter to an object, thereby invoking the `call` method from the object's class.

How to do it...

See its usage in the `call` project, as shown in the following code:

```
var u = "Julia";
```

```
void main() {  
    var embr = new Embrace(5);
```

The `call` method can be used in the following ways:

1. Invoke `call`, as shown in the following code:

```
embr(u); // callable method!  
var m = new Mult();  
print(m(3, 4));  
}
```

```
class Embrace {  
    // see code in recipe: Using reflection
```

2. Define the `call` method, as shown in the following code:

```
call(var user) { print("$user is called, and hugged with  
strength $strength!"); }  
}
```

```
class Mult{  
    call(int a, int b) => a * b;  
}
```

We get the following output on the screen:

Julia is called, and hugged with strength 5!

12

How it works...

We pass the value `u` to the `embr` object in the `embr(u)` call, which invokes the `call` method. This method defines what the instances of your class do when invoked as functions via the `()` syntax. The advantage of using this in a normal class is perhaps not that clear. It is more useful when making a class that in fact wants to emulate a function, such as the `Mult` class. An object of this class can take two integers and return their product. This example is trivial and not worth writing a special class for it, but there are cases where this ability can be put to use.

There's more...

The `Mult` class contains only one function. We could also have written it as a `Multi` function, as shown in the following code:

```
int Multi(int a, int b) => a * b;
```

We could have even invoked it with `print(Multi(3, 4)); // 12.`

All functions in Dart are of type `Function`, but we can be more specific by defining a `typedef`. A `typedef` is a way to give a name to a function's signature, that is, the type of its arguments and return type. The `typedef IntOp` generalizes the type of `Multi`, as shown in the following code:

```
typedef int IntOp(int a, int b);  
Then we can write for example, when f is defined as a Function:  
assert(Multi is IntOp);  
f = Multi;  
print(f(3, 4)); // 12  
assert(f is IntOp);
```

Using noSuchMethod

When a method is called on an object, and this method does not exist in its class, or any of its superclasses in the inheritance tree, then `noSuchMethod()` from `Object` is called. The default behavior of `noSuchMethod` is to throw a `NoSuchMethodError`, **method not found: 'methodname'**. However, Dart can do more; as in some other dynamic languages, every class can implement `noSuchMethod` to make its behavior more adaptive and flexible. This is because of the fact that Dart is dynamically typed, so it is possible to call a method that does not exist in a dynamic variable. In Java, you get a compile time error for this. In Dart too, an error is thrown but at runtime. By using `noSuchMethod()`, we can circumvent this and put it to our use.

How to do it...

See noSuchMethod in action in the nosuchmethod project:

```
void main() {
    var embr = new Embrace(5);
    print(embr.missing("42", "Julia")); // is a missing method!
}

@proxy
class Embrace {
    // see code in previous recipes
    @override
    noSuchMethod(Invocation msg) => "got ${msg.memberName} "
        "with arguments ${msg.positionalArguments}";
}
```

This script gives the output **got Symbol("missing") with arguments [42, Julia]**.

How it works...

When `missing` is called and not found, `noSuchMethod` is found and executed instead. The exact signature of the method is `noSuchMethod(Invocation msg)`. When it is invoked, an object `msg` of type `Invocation` is passed to it, which contains the names of the method and its arguments. If `noSuchMethod` returns a value, that value becomes the result of the original `Invocation`. `Invocation` also has Boolean getters `isMethod`, `isAccessor`, `isGetter`, and `isSetter` to find out whether the called method was a normal method, getter, or setter. With this information being passed, we could do something more useful than just print it, as shown in the following code:

```
noSuchMethod(Invocation msg) =>
    msg.memberName == #meth1 ? Function.apply(meth2,
        msg.positionalArguments,
        msg.namedArguments)
    : super.noSuchMethod(msg);
```

Here, we check whether the called method was `meth1` (notice that `memberName` is of type `Symbol`) and if so, we call the `meth2` method by passing the supplied arguments. Calling `meth2` is done through the static method `apply()` in the `Function` class, which allows functions to be called in a generic fashion. Use the `@override` annotation to indicate that you are intentionally overriding a member. Use the `@proxy` annotation on the class header itself to avoid analyzer warnings if you use `noSuchMethod()` to implement every possible getter, setter, and method for a class.

There's more...

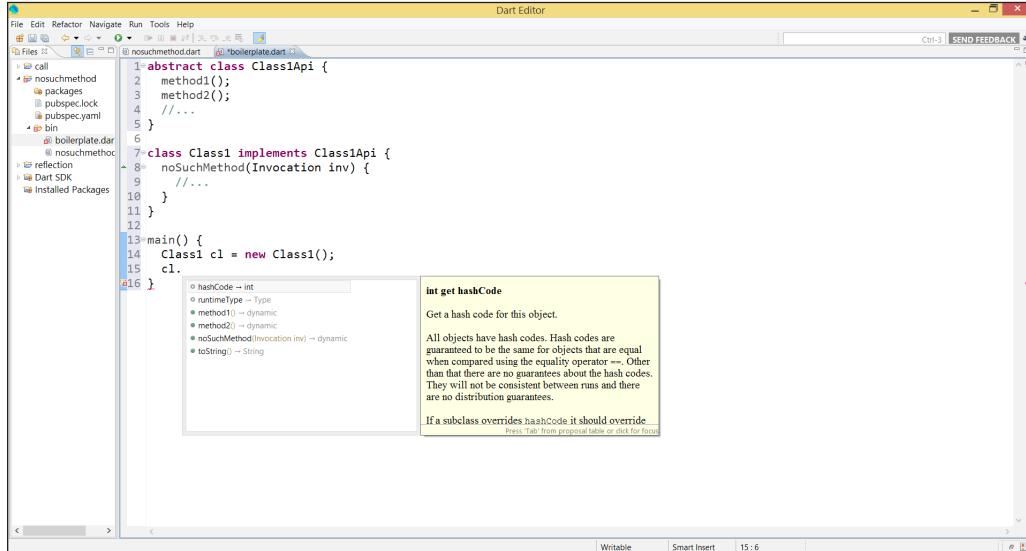
Another good reason to use `noSuchMethod` is to reduce boilerplate code when writing a lot of similar methods. To make sure that code completion still works, use the following structure:

```
abstract class Class1Api {
    method1();
    method2();
    //...
}

class Class1 implements Class1Api {
    noSuchMethod(Invocation inv) {
        //...
    }
}

main() {
    Class1 cl = new Class1();
    // cl. // remove comment and type cl. to see method1()
    // and so on in code completion
}
```

All the methods that will implement in `noSuchMethod` are summed up in the abstract class `Class1Api`, and then the code completion list will present them. See the code in the `boilerplate.dart` file and the following screenshot:



See also

- ▶ See the *Using annotations* recipe in this chapter for more information on `@override` and `@proxy`

Making `toJson` and `fromJson` methods in your class

JavaScript Object Notation (JSON) is probably the most widely used data format in web applications, so it is a common requirement for a class to be able to serialize its objects to JSON strings, or reconstruct objects from JSON strings.

Getting ready

JSON is lightweight (not as verbose as XML) and text-based, so it is easily readable by humans. It starts from the notion that the state (or content) of an object is in fact like a map; the keys are the field names, and their values are the concrete data stored in the fields. For example, (see project `json/job.dart`), say we have a class `Job` defined, as shown in the following code:

```
class Job {  
    String type;  
    int salary;  
    String company;  
    Job(this.type, this.salary, this.company);  
}
```

Next, we construct a job object with the following code:

```
var job = new Job("Software Developer", 7500, "Julia Computing LLC")  
;
```

Then, it can be represented as the following JSON string:

```
'{  
    "type": "Software Developer",  
    "salary": 7500,  
    "company": "Julia Computing LLC"  
'
```

The values can themselves be lists or maps or lists of maps. For more information about JSON, refer to <http://en.wikipedia.org/wiki/JSON>.

How to do it...

- ▶ The following is the code to give our class JSON functionality:

```
import 'dart:convert';

class Job {
    String type;
    int salary;
    String company;
    Job(this.type, this.salary, this.company);
```

- ▶ The following is the code to encode or serialize data:

```
String toJson() {
    var jsm = new Map<String, Object>();
    jsm["type"] = type;
    jsm["salary"] = salary;
    jsm["company"] = company;
    var jss = JSON.encode(jsm);
    return jss;
}
```

- ▶ The following is the code to decode or deserialize data:

```
Job.fromJson(String jsonStr) {
    Map jsm = JSON.decode(jsonStr);
    this.type = jsm["type"];
    this.salary = jsm["salary"];
    this.company = jsm["company"];
}
void main() {
    var job = new Job("Software Developer", 7500, "Julia Computing
LLC");
    var jsonStr = job.toJson();
    print(jsonStr);
    var job2 = new Job.fromJson(jsonStr);
    assert(job2 is Job);
    assert(job2.toJson() == jsonStr);
}
```

The output of `jsonStr` is **{"type": "Software Developer", "salary": 7500, "company": "Julia Computing LLC"}**.

The assert statements confirm that the decoded object is of type `Job` and is equal to the JSON string we started from.

How it works...

A JSON string can be stored in a file or database, or sent over the network to a server. So our class needs to be able to:

- ▶ Write its objects out in JSON format, which is also called serializing or encoding; we'll conveniently call this method `toJson()`
- ▶ Read a JSON string and construct an object (or many objects) from it; this is called deserializing or decoding, and we'll make a method `fromJson()` to do just this

Part of the work is done by functions in the imported `dart:convert` library, which produces and consumes JSON data, respectively:

- ▶ `JSON.encode()`: This serializes a Dart object into a JSON string, ready to be stored or sent over a network
- ▶ `JSON.decode()`: This builds Dart objects from a string containing JSON data, which is just read from storage or received over the network

These functions on `JSON` (which is an object of the class `JSONCodec`) can process data from the types `null`, `num`, `bool`, `String`, `List`, and `Map` automatically and also from a combination of these (the keys of the map need to be strings). In our `Job` class, the data is processed, as explained in the following points:

- ▶ `toJson()` makes the map from the object and then calls `JSON.encode()` on it to return a JSON string
- ▶ `fromJson()` (conveniently implemented as a named constructor) takes the JSON string, calls `JSON.decode()` to return a map, and builds the object

There's more...

If your data is not that complicated, you can build the JSON string in the code yourself, possibly as a getter, as shown in the following snippet:

```
String get toJson => '{
  "type": "$type",
  "salary": "$salary",
  "company": "$company"
}';
```



If you have to construct JSON strings literally in your code, make sure to always use double quotes to indicate strings; Dart single quotes cannot be used here.

When an object contains other objects (composition or association, such as a `BankAccount` object containing a `Person` object for the owner), the `fromJson()` and `toJson()` methods from the outer object will call the corresponding methods with the same name for all the contained objects.

If you want a more sophisticated solution that supports the encoding and decoding of arbitrary objects, look for the `jsonx` package on pub, by Man Hoang. This library can decode a JSON string into a strongly typed object, which gets type checking and code completion support, or encodes an arbitrary object into a JSON string. When working with JSON, it is preferred to validate it; this can be done online at <http://jsonlint.com/>.

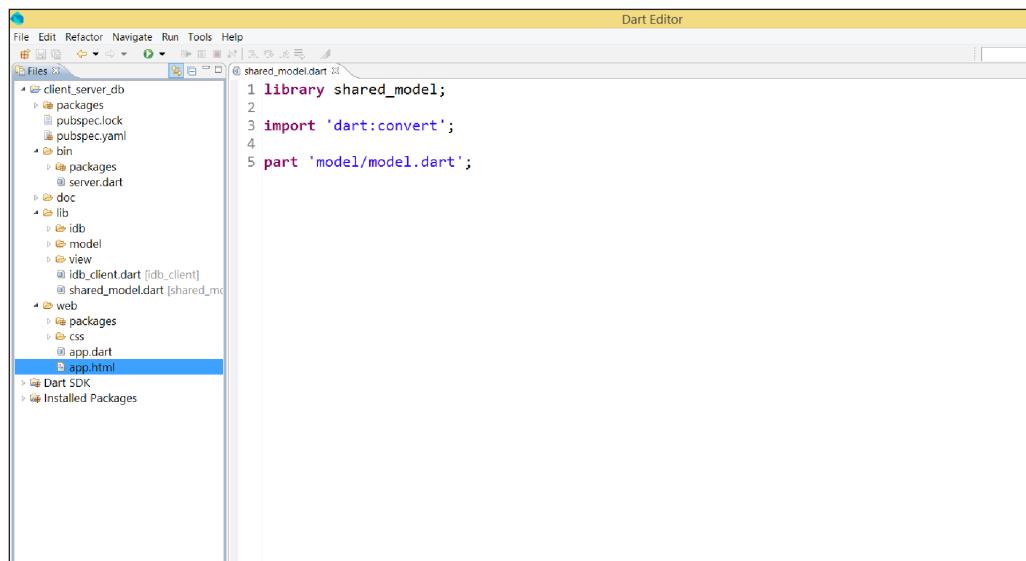
Creating common classes for client and server apps

In distributed apps such as the client-server pattern, you often want to work with the same model classes on both ends. Why? Because client input needs to be validated at the client side and for this, we need the model in the client app. Data has to pass through the model before being stored so that if we want to store the data in a client database (`indexed_db`) as well as in a server data store, we need the model on both the sides.

How to do it...

You can see how common classes are created for client and server apps in the `client_server_db` app. This is a to-do application; the client is started from `web/app.html` and the server is started from `bin/server.dart`. The client stores the to-do data in `indexed_db`, while the server stores the data in memory. Both client and server need the model classes.

The project structure is shown in the following screenshot:



Common library

How it works...

If you want to be able to import your common model library from another package, the files must be under the `lib` directory. In this example, we see from `pubspec.yaml` that the name of the app is `client_server`. The model (with the `Task` and `Tasks` classes) has its own folder `lib/model`, which contains the model classes defined in the `shared_model` library within `lib/shared_model.dart`. The client-app `web/app.dart` uses `indexed_db`; it therefore imports a library `idb_client` to work with `indexed_db`, as shown in the following code:

```
import 'package:client_server/idb_client.dart';
```

The `idb_client` library, defined in `lib/idb_client.dart`, imports the `shared_model` library, as shown in the following code:

```
library idb_client;

import 'package:client_server/shared_model.dart';
import 'dart:async';
import 'dart:html';
import 'dart:indexed_db';
import 'dart:convert';

part 'idb/idb.dart';
part 'view/view.dart';
```

In this way, the client app knows about the model. The server app also knows about the model by importing it (see `bin/server.dart`):

```
import 'dart:io';
import 'dart:convert';
import 'package:client_server/shared_model.dart';
// rest of code
```

There's more...

Here's another common use case for shared models; say the class has a method that performs HTTP requests; on the client, you will use `HttpRequest` from the `dart:html` library, while on the server, you will use the one from `dart:io` instead. For security reasons, both `dart:io` and `dart:html` cannot be imported in the same library, so it's a common practice to define the shared class as an abstract class, and then delegate the concrete implementation of the `HttpRequest` method to both the client and server classes, which extend the shared abstract class.

See also

- ▶ Refer to the *Structuring an application* and *Making and using a library* recipes in *Chapter 2, Structuring, Testing, and Deploying an Application* for your app

5

Handling Web Applications

In this chapter, we will cover the following recipes:

- ▶ Responsive design
- ▶ Sanitizing HTML
- ▶ Using a browser's local storage
- ▶ Using an application cache to work offline
- ▶ Preventing an onSubmit event from reloading the page
- ▶ Dynamically inserting rows in an HTML table
- ▶ Using CORS headers
- ▶ Using keyboard events
- ▶ Enabling drag-and-drop
- ▶ Enabling touch events
- ▶ Creating a Chrome app
- ▶ Structuring a game project
- ▶ Using WebGL in your app
- ▶ Authorizing OAuth2 to Google services
- ▶ Talking with JavaScript
- ▶ Using JavaScript libraries

Introduction

Web applications are what Dart was made for, so it comes as no surprise that we have a lot of questions to deal with in this area. Dart here as a client language presents itself as an alternative to JavaScript (to which it compiles), but also to CoffeeScript and TypeScript. Because the language is a higher-level one and more robust, Dart enables developers to reach a higher rate of productivity. Its structure and tooling makes possible the building of complex software systems with large teams. When running in its virtual machine, Dart delivers a very shortened app startup time, and higher performance during execution. All these enhancements make Dart a prime choice to develop browser apps. You'll find topics in this chapter that deal with safety, browser storage, all kinds of interactive events, WebGL, and of course, working together with JavaScript.

Responsive design

Nowadays, users have to interact with computer screens of all different sizes, from smartphones and tablets to laptops, desktop monitors, and TVs. To design a web application in such a way that page layouts adapt intelligently to the user's screen width resolutions is called responsive design; for example, an advanced four-column layout 1292 pixels wide, on a 1025-pixel-wide screen, that autosimplifies into two columns when viewed on a tablet or smartphone. Its significance is now broadened to encompass web applications that respond to the user's environment intelligently, but also to make the web app adapt to the user's behavior. If you do only one thing to make your app's responsive design aware, apply what you read in this topic.

How to do it...

Add the following `<meta>` tag (the so-called `viewport` tag) to the `<head>` section of your HTML pages:

```
<meta name="viewport"  
      content="width=device-width, initial-scale=1.0">
```

How it works...

This will set you up for cross-device layout peace of mind. `viewport` is another word for screen width and this tag was originally devised by Apple. Setting `content` to `"width=device-width"` will query your device for its standard width and set your layout width accordingly. To be extra certain that your layout will be displayed as you intended it, you can also set the zoom level with `content="initial-scale=1"`. This will make sure that upon opening the page, your layout will be displayed properly at a 1:1 scale; no zooming will be applied. You can even prevent any zooming by adding a third attribute value `"maximum-scale=1"`. However, you must make sure that everything is readable for everybody; using this would probably hinder people with visual problems.

See also

- ▶ If you want to start learning more about responsive design, a nice tutorial is available at <http://www.adamkaplan.me/grid/>

Sanitizing HTML

We've all heard of (or perhaps even experienced) **cross-site scripting (XSS)** attacks, where evil minded attackers try to inject client-side script or SQL statements into web pages. This could be done to gain access to session cookies or database data, or to get elevated access-privileges to sensitive page content. To verify an HTML document and produce a new HTML document that preserves only whatever tags are designated safe is called sanitizing the HTML.

How to do it...

Look at the web project sanitization. Run the following script and see how the text content and default sanitization works:

1. See how the default sanitization works using the following code:

```
var elem1 = new Element.html('<div class="foo">content</div>');
document.body.children.add(elem1);
var elem2 = new Element.html('<script class="foo">evil content</
script><p>ok?</p>');
document.body.children.add(elem2);
```

The text content and `ok?` from `elem1` and `elem2` are displayed, but the console gives the message **Removing disallowed element <SCRIPT>**. So a script is removed before it can do harm.

2. Sanitize using `HtmlEscape`, which is mainly used with user-generated content:

```
import 'dart:convert' show HtmlEscape;
```

In `main()`, use the following code:

```
var unsafe = '<script class="foo">evil      content</script><p>ok?</p>';
var sanitizer = const HtmlEscape();
print(sanitizer.convert(unsafe));
```

This prints the following output to the console:

```
&lt;script class="foo">evil      content&lt;#x2F;script&gt;&lt;p&gt;ok?&lt;#x2F;p&gt;
```

3. Sanitize using node validation. The following code forbids the use of a `<p>` tag in `node1`; only `<a>` tags are allowed:

```
var html_string = '<p class="note">a note aside</p>';
var node1 = new Element.html(
    html_string,
    validator: new NodeValidatorBuilder()
        ..allowElement('a', attributes: ['href'])
);
```

The console prints the following output:

Removing disallowed element <p>

Breaking on exception: Bad state: No elements

4. A `NullTreeSanitizer` for no validation is used as follows:

```
final allHtml = const NullTreeSanitizer();
class NullTreeSanitizer implements NodeTreeSanitizer {
    const NullTreeSanitizer();
    void sanitizeTree(Node node) {}
}
```

It can also be used as follows:

```
var elem3 = new Element.html('<p>a text</p>');
elem3.setInnerHtml(html_string, treeSanitizer: allHtml);
```

How it works...

First, we have very good news: Dart automatically sanitizes all methods through which HTML elements are constructed, such as `new Element.html()`, `Element.innerHTML()`, and a few others. With them, you can build HTML hardcoded, but also through string interpolation, which entails more risks. The default sanitization removes all scriptable elements and attributes.

If you want to escape all characters in a string so that they are transformed into HTML special characters (such as ;#x2F for a /), use the class `HTMLEscape` from `dart:convert` as shown in the second step. The default behavior is to escape apostrophes, greater than/less than, quotes, and slashes. If your application is using untrusted HTML to put in variables, it is strongly advised to use a validation scheme, which only covers the syntax you expect users to feed into your app. This is possible because `Element.html()` has the following optional arguments:

```
Element.html(String html, {NodeValidator validator, NodeTreeSanitizer treeSanitizer})
```

In step 3, only `<a>` was an allowed tag. By adding more `allowElement` rules in cascade, you can allow more tags. Using `allowHtml5()` permits all HTML5 tags.

If you want to remove all control in some cases (perhaps you are dealing with known safe HTML and need to bypass sanitization for performance reasons), you can add the class `NullTreeSanitizer` to your code, which has no control at all and defines an object `allHtml`, as shown in step 4. Then, use `setInnerHtml()` with an optional named attribute `treeSanitizer` set to `allHtml`.

Using a browser's local storage

Local storage (also called the Web Storage API) is widely supported in modern browsers. It enables the application's data to be persisted locally (on the client side) as a map-like structure: a dictionary of key-value string pairs, in fact using JSON strings to store and retrieve data. It provides our application with an offline mode of functioning when the server is not available to store the data in a database. Local storage does not expire, but every application can only access its own data up to a certain limit depending on the browser. In addition, of course, different browsers can't access each other's stores.

How to do it...

Look at the following example, the local_storage.dart file:

```
import 'dart:html';

Storage local = window.localStorage;

void main() {
    var job1 = new Job(1, "Web Developer", 6500, "Dart Unlimited") ;
```

Perform the following steps to use the browser's local storage:

1. Write to a local storage with the key Job:1 using the following code:

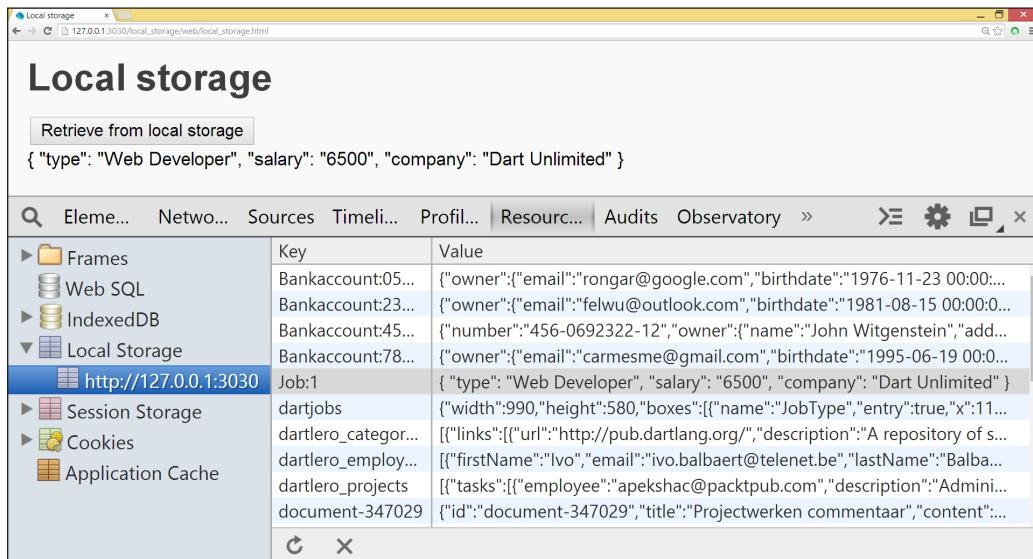
```
local["Job:${job1.id}"] = job1.toJson;
ButtonElement bel = querySelector('#readls');
bel.onClick.listen(readShowData);
}
```

2. A click on the button checks to see whether the key Job:1 can be found in the local storage, and, if so, reads the data in. This is then shown in the data <div>:

```
readShowData(Event e) {
    var key = 'Job:1';
    if(local.containsKey(key)) {
        // read data from local storage:
        String job = local[key];
        querySelector('#data').appendText(job);
    }
}

class Job {
    int id;
    String type;
    int salary;
    String company;
    Job(this.id, this.type, this.salary, this.company);
    String get toJson => '{ "type": "$type", "salary": "$salary",
"company": "$company" } ';
```

The following screenshot depicts how data is stored in and retrieved from local storage:



How it works...

You can store data with a certain key in the local storage from the Window class as follows using `window.localStorage[key] = data;` (both key and data are Strings).

You can retrieve it with `var data = window.localStorage[key];`

In our code, we used the abbreviation `Storage local = window.localStorage;` because `local` is a map. You can check the existence of this piece of data in the local storage with `containsKey(key)`; in Chrome (also in other browsers via **Developer Tools**). You can verify this by navigating to **Extra | Tools | Resources | Local Storage** (as shown in the previous screenshot) `window.localStorage` also has a `length` property; you can query whether it contains something with `isEmpty`, and you can loop through all stored values using the following code:

```
for(var key in window.localStorage.keys) {  
    String value = window.localStorage[key];  
    // more code  
}
```

There's more...

Local storage can be disabled (by user action, or via an installed plugin or extension), so we must alert the user when this needs to be enabled; we can do this by catching the exception that occurs in this case:

```
try {
    window.localStorage[key] = data;
} on Exception catch (ex) {
    window.alert("Data not stored: Local storage is disabled!");
}
```

Local storage is a simple key-value store and does have good cross-browser coverage. However, it can only store strings and is a blocking (synchronous) API; this means that it can temporarily pause your web page from responding while it is doing its job storing or reading large amounts of data such as images. Moreover, it has a space limit of 5 MB (this varies with browsers); you can't detect when you are nearing this limit and you can't ask for more space. When the limit is reached, an error occurs so that the user can be informed.



These properties make local storage only useful as a temporary data storage tool; this means it is better than cookies, but not suited for a reliable, database kind of storage.



Web storage also has another way of storing data called `sessionStorage` used in the same way, but this limits the persistence of the data to only the current browser session. So, data is lost when the browser is closed or another application is started in the same browser window.

See also

- ▶ For more information on the JSON format, refer to the *Making toJSON and fromJSON methods* recipe in *Chapter 4, Object Orientation in your class*
- ▶ A better alternative to simple local storage is IndexedDB; see the *Storing data locally in IndexedDB* recipe in *Chapter 9, Working with Databases*

Using application cache to work offline

When, for some reason, our users don't have web access or the website is down for maintenance (or even broken), our web-based applications should also work offline. The browser cache is not robust enough to be able to do this, so HTML5 has given us the mechanism of `ApplicationCache`. This cache tells the browser which files should be made available offline. The effect is that the application loads and works correctly, even when the user is offline. The files to be held in the cache are specified in a manifest file, which has a `.wma` or `.appcache` extension.

How to do it...

Look at the appcache application; it has a manifest file called `appcache.mf`.

1. The manifest file can be specified in every web page that has to be cached. This is done with the manifest attribute of the `<html>` tag:

```
<html manifest="appcache.mf">
```

If a page has to be cached and doesn't have the manifest attribute, it must be specified in the CACHE section of the manifest file. The manifest file has the following (minimum) content:

```
CACHE MANIFEST
# 2012-09-28:v3

CACHE:
Cached1.html
appcache.css
appcache.dart
http://dart.googlecode.com/svn/branches/bleeding_edge/dart/client/
dart.js

NETWORK:
*

FALLBACK:
/ offline.html
```

2. Run `Cached1.html`. This displays the **This page is cached, and works offline!** text. Change the text to **This page has been changed!** and reload the browser. You don't see the changed text because the page is created from the application cache.
3. When the manifest file is changed (change version v1 to v2), the cache becomes invalid and the new version of the page is loaded with the **This page has been changed!** text.
4. The Dart script of the page, `appcache.dart`, should contain the following minimal code to access the cache:

```
main() {
    new AppCache(window.applicationCache);
}

class AppCache {
    ApplicationCache appCache;

    AppCache(this.appCache) {
```

```
    appCache.onUpdateReady.listen((e) => updateReady());
    appCache.onError.listen(onCacheError);
}

void updateReady() {
    if (appCache.status == ApplicationCache.UPDATEREADY) {
        // The browser downloaded a new app cache. Alert the user:
        appCache.swapCache();
        window.alert('A new version of this site is available.
Please reload.');
    }
}

void onCacheError(Event e) {
    print('Cache error: ${e}');
    // Implement more complete error reporting to developers
}
}
```

How it works...

The CACHE section in the manifest file enumerates all the entries that have to be cached. The NETWORK: and * options mean that to use all other resources, the user has to be online. Fallback specifies that offline.html will be displayed if the user is offline and a resource is inaccessible. A page is cached when either of the following is true:

- ▶ Its HTML tag has a manifest attribute pointing to the manifest file
- ▶ The page is specified in the CACHE section of the manifest file

The browser is notified when the manifest file is changed, and the user will be forced to refresh its cached resources. Adding a timestamp and/or a version number such as # 2014-05-18:v1 works fine. Changing the date or the version invalidates the cache, and the updated pages are again loaded from the server.

To access the browser's app cache from your code, use the `window.applicationCache` object. Make an object of the class `AppCache`, and alert the user when the application cache has become invalid (the status is **UPDATEREADY**) by defining an `onUpdateReady` listener.

There's more...

The other known states of the application cache are UNCACHED, IDLE, CHECKING, DOWNLOADING, and OBSOLETE. To log all these cache events, you could add the following listeners to the appCache constructor:

```
appCache.onCached.listen(onCacheEvent) ;  
appCache.onChecking.listen(onCacheEvent) ;  
appCache.onDownloading.listen(onCacheEvent) ;  
appCache.onNoUpdate.listen(onCacheEvent) ;  
appCache.onObsolete.listen(onCacheEvent) ;  
appCache.onProgress.listen(onCacheEvent) ;
```

Provide an onCacheEvent handler using the following code:

```
void onCacheEvent(Event e) {  
    print('Cache event: ${e}') ;  
}
```

Preventing an onSubmit event from reloading the page

The default action for a submit button on a web page that contains an HTML form is to post all the form data to the server on which the application runs. What if we don't want this to happen?

How to do it...

Experiment with the `submit` application by performing the following steps:

1. Our web page `submit.html` contains the following code:

```
<form id="form1" action="http://www.dartlang.org" method="POST">  
  <label>Job:<input type="text" name="Job" size="75"></input>  
  </label>  
  <input type="submit" value="Job Search">  
</form>
```

Comment out all the code in `submit.dart`. Run the app, enter a job name, and click on the **Job Search** submit button; the Dart site appears.

2. When the following code is added to `submit.dart`, clicking on the **no** button for a longer duration makes the Dart site appear:

```
import 'dart:html';

void main() {
    querySelector('#form1').onSubmit.listen(submit);
}

submit(Event e) {
    e.preventDefault();
    // code to be executed when button is clicked
}
```

How it works...

In the first step, when the **submit** button is pressed, the browser sees that the method is POST. This method collects the data and names from the input fields and sends it to the URL specified in `action` to be executed, which only shows the Dart site in our case.

To prevent the form from posting the data, make an event handler for the `onSubmit` event of the form. In this handler code, `e.preventDefault()`; as the first statement will cancel the default submit action. However, the rest of the `submit` event handler (and even the same handler of a parent control, should there be one) is still executed on the client side.

Dynamically inserting rows in an HTML table

When displaying data coming from a database, you often don't know how many data records there will be. Our web page and the HTML table in it have to adapt dynamically. The following recipe describes how to do this.

How to do it...

Look at the `html_table` application. The web page contains two `<table>` tags:

```
<table id="data"></table>
<table id="jobdata"></table>
```

On running the app, you will be redirected to the following web page, which displays data in an HTML table:

The screenshot shows a web browser window with the title "Html table". Below the title, there is some text: "cell 0-0" followed by three colored boxes: green ("cell 0-1"), red ("cell 0-2"), and black ("cell 1-0"). Below this, there is a table with a light blue header row containing columns for "Jobtype", "Salary", and "Company". The data rows are: "Software Developer 7500 Julia Computing LLC", "Web Developer 6500 Dart Unlimited", and "Project Manager 10500 Project Consulting Inc.". The entire table is enclosed in a light blue border.

Jobtype	Salary	Company
Software Developer	7500	Julia Computing LLC
Web Developer	6500	Dart Unlimited
Project Manager	10500	Project Consulting Inc.

Displaying data in an HTML table

The data is shown by the code in the `html_table.dart` file.

1. To make the code more flexible, the necessary element objects are declared up front; we use the class `Job` to insert some real data:

```
TableElement table;
TableRowElement row;
TableCellElement cell;
List<Job> jobs;

class Job {
    String type;
    int salary;
    String company;
    Job(this.type, this.salary, this.company);
}
```

The first table is shown with the preceding code.

2. Find the created table using the following query:

```
table = querySelector('#data');
```

3. Insert a row at index 0, and assign that row to a variable:

```
row = table.insertRow(0);
```

4. Insert a cell at index 0, assign that cell to a variable, and provide it with content, as shown in the following code:

```
cell = row.insertCell(0);
cell.text = 'cell 0-0';
```

5. Insert more cells with a message cascading approach and style them using the following code:

```
row.insertCell(1)
..text = 'cell 0-1'
..style.background = 'lime';
row.insertCell(2)
..text = 'cell 0-2'
..style.background = 'red';
```

6. Insert a new row at the end of the table:

```
row = table.insertRow(-1);
row.insertCell(0).text = 'cell 1-0';
```

Here is the code to display data from a List, applying the same methods as above:

```
var job1 = new Job("Software Developer", 7500, "Julia Computing
LLC") ;
var job2 = new Job("Web Developer", 6500, "Dart Unlimited") ;
var job3 = new Job("Project Manager", 10500, "Project Consulting
Inc.") ;
jobs = new List<Job>();
jobs
..add(job1)
..add(job2)
..add(job3);
table = querySelector('#jobdata');
// insert table headers:
InsertHeaders();
// inserting data:
InsertData();

InsertHeaders() {
row = table.insertRow(-1);
cell = row.insertCell(0);
cell.text = "Jobtype";
cell.style.background = 'lightblue';
```

```
cell = row.insertCell(1);
cell.text = "Salary";
cell = row.insertCell(2);
cell.text = "Company";
cell.style.background = 'lightblue';
}

InsertData() {
  for (var job in jobs) {
    row = table.insertRow(-1);
    cell = row.insertCell(0);
    cell.text = job.type;
    cell = row.insertCell(1);
    cell.text = (job.salary).toString();
    cell = row.insertCell(2);
    cell.text = job.company;
  }
}
```

7. The preceding code gets the content from the indicated cell:

```
print(table.rows[1].cells[1].text); // prints 7500
```

How it works...

We use the methods `insertRow()` and `insertCell()` from `TableElement` and `TableRowElement`, respectively, and the properties of `TableCellElement`. Rows and columns are numbered from 0. As shown in the last line of the code, the content of specific cells can be retrieved by using an indexer `[]` on a specific row and cell.

See also...

- ▶ The API docs at <https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart-dom-html.TableElement> will show you more useful methods

Using CORS headers

In the web application security model, the same-origin policy is an important concept. The basic principle is that content provided by unrelated websites must be strictly separated on the client side; otherwise, confidentiality or data integrity might be compromised, perhaps through cross-site scripting attacks. In other words, web pages or scripts running on pages can only access scripts or pages from the same domain as they came from; no access to other sites is allowed. For example, `http://www.example.com/dir/page2.html` cannot access `http://en.example.com/dir/other.html`. However, in a number of cases, this is too strict, as in AJAX calls with `HttpRequest` we have to load data from another server (refer to *Chapter 7, Working with Web Servers*). To make this possible, the CORS mechanism (cross-origin resource sharing) was developed, which is supported by most modern web browsers. This recipe will enable you to easily achieve this by performing the following steps.

How to do it...

The following steps show how you can configure your web server to add CORS headers:

- When a web server sends CORS headers back in the response, the client is also allowed to send requests to servers in other domains. So in every request handling code, we must add a call to a method such as `addCorsHeaders` as shown in the following code:

```
void handleGet(HttpServletRequest request) {  
    HttpServletResponse res = request.response;  
    addCorsHeaders(res);  
    // other code to prepare the response  
    // ...  
    res.write(content);  
    res.close();  
}
```

- Now we need to define the `addCorsHeaders` method; it contains the following code:

```
void addCorsHeaders(HttpServletResponse response) {  
    response.headers.add('Access-Control-Allow-Origin', '*', '');  
    response.headers.add('Access-Control-Allow-Methods', 'POST,  
OPTIONS');  
    response.headers.add('Access-Control-Allow-Headers', 'Origin,  
X-Requested-With, Content-Type, Accept');  
}
```

How it works...

All CORS-related headers are prefixed with `Access-Control-`. The `Access-Control-Allow-Origin` option has to be included in all valid CORS responses. The value `*` means that access is allowed from all domains. In development, this can be useful so that you can run apps from Dart Editor, which uses a 3030 port by default for its internal server. However, for a production application, you should sum up the allowed origins as follows:

- ▶ `Access-Control-Allow-Origin` at `http://www.example-social-network.com`
- ▶ `Access-Control-Allow-Origin` at `http://www.snapshot.com`

So, effectively, we name all the websites to which access is allowed. In our code, the `POST` and `OPTIONS` requests are allowed from any origin. An `OPTIONS` request is sent to get permission from the server to post data, in case the client is running from a different origin. So before the `POST` request, a so-called preflighted `OPTIONS` request is sent to determine if the actual request is allowed.



In general, using CORS headers is not safe, and the allowed origins should be summed up. However, for development purposes, it is useful to allow them.



There's more...

Refer to <http://enable-cors.org/server.html> for more detailed information on using CORS on different platforms.

Using keyboard events

Handling keyboard events in normal web applications is not so common. However, if you are writing a web game, you'll almost certainly want to catch arrow key input, and the like. This recipe shows us how we can handle those events.

How to do it...

Look at the `keyboard` project. The web page is a copy of the page used in the *Preventing an `onSubmit` event* recipe, but now `submit` stays enabled. Suppose we also want to ensure that pressing the `Enter` key provokes `submit`, just like clicking on the `submit` button.

1. To that end, we add the following event listener to `main()`:

```
document.onKeyPress.listen(_onKeyPress);
```

2. The `_onKeyPress` method is as follows:

```
_onKeyPress(KeyboardEvent e) {  
  if (e.keyCode == KeyCode.ENTER) submit(e);  
}
```

Now, pressing *Enter* will cause the form to be submitted.

How it works...

The document object from `dart :html` has three events to work with key input: `onKeyPress`, `onKeyDown`, and `onKeyUp`. They all generate a stream of `KeyboardEvent` objects that capture user interaction with the keyboard. However, these events are also defined for `window`, the parent object of `document`, and any `Element` on the page, so you can use them very specifically on a certain `InputElement` or `<div>` region.

The `keyCode` of the event is an integer from a list of constants, which specifies a value for every key, such as `KeyCode.A`, `KeyCode.SPACE`, and `KeyCode.LEFT` for the left arrow key, and so on. This lends itself very easily to a `switch/case` construct, as follows:

```
_onKeyPress(KeyboardEvent e) {  
  e.preventDefault();  
  print('The charCode is ${e.charCode}');  
  if (e.keyCode == KeyCode.ENTER) submit(e);  
  if (e.ctrlKey) return;  
  
  switch(e.keyCode) {  
    case KeyCode.DOWN:  
      // move sprite down  
      break;  
    case KeyCode.UP:  
      // move sprite up  
      break;  
    case KeyCode.F1:  
      // show help  
      break;  
  }  
}
```

The event also has an integer character code getter named `charCode`. If the key was a special key, this can also be tested with `e.ctrlKey`, `e.altKey`, `e.metaKey`, `e.shiftKey`, and `e.altGraphKey`:

```
if (e.ctrlKey) return;
```

There's more...

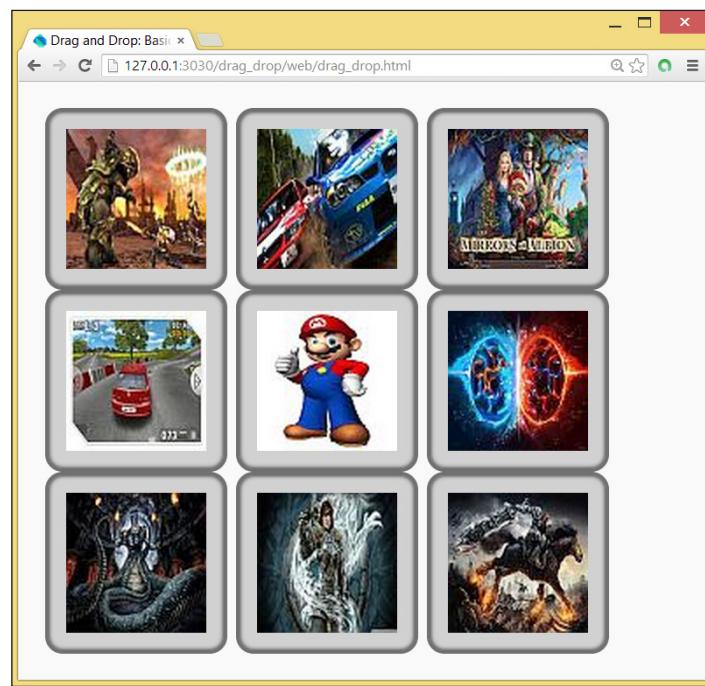
The Dart classes for keyboard handling try to minimize cross-browser differences (usually related to special keys), and to do a good job with as many international keyboard layouts as possible.

Enabling drag-and-drop

Imagine developing a board game and the player has to use the keyboard to move the pieces; this wouldn't be acceptable anymore. Before HTML5, drag-and-drop using the mouse was a feature that had to be implemented through an external library like Dojo or jQuery. However, HTML5 provides native browser support to make nearly every element on a web page draggable, thus allowing more user-friendly web apps. This recipe shows you how to implement drag-and-drop (abbreviated as DnD) with Dart.

How to do it...

Run the `drag_drop` project. The result is a board of images as shown in the following screenshot, where the images are draggable and you can swap an image with any other image, as shown in the following screenshot:



Drag-and-drop images

Perform the following steps to enable drag-and-drop

1. The elements that we want to drag-and-drop must get the `draggable` attribute in HTML, so in `drag_drop.html`, we indicate this for the `<div>` elements of the board:

```
<div id="tiles">
    <div id="tile1" class="pict" draggable="true"></div>
    <div id="tile2" class="pict" draggable="true"></div>
    ...
</div>
```

2. In the accompanying script `drag_drop.dart`, we make a class, `DragnDrop`, to integrate all the drag-and-drop event handling. All our `<div>` tiles have a CSS class `pict` and are hence captured in the `cols` collection. The `main()` function makes an object, `dnd`, and calls the `init()` method on it:

```
void main() {
    var dnd = new DragnDrop();
    dnd.init();
}

class DragnDrop {
    Element dragSource;
    Map tiles;
    var rand = new Random();
    var cols = document.querySelectorAll('.pict');
    // ...
}
```

3. The `init()` function constructs a map with images from the `img` folder in `generateMap()`. It sets up listeners for all the Dnd events of our tiles, and the images are set up as background images for the tiles via `style.setProperty` by calling `getRandomTile()`:

```
void init() {
    generateMap();
    for (var col in cols) {
        col
            ..onDragStart.listen(_onDragStart)
            ..onDragEnd.listen(_onDragEnd)
            ..onDragEnter.listen(_onDragEnter)
            ..onDragOver.listen(_onDragOver)
            ..onDragLeave.listen(_onDragLeave)
            ..onDrop.listen(_onDrop)
            ..style.setProperty("background-image", getRandomTile(),
        ""))
}
```

```
..style.setProperty("background-repeat", "no-repeat", "")
..style.setProperty("background-position", "center", "");
}
}

void generateMap() {
    tiles = new Map();

    for (var i = 1; i <= 9; i++) {
        tiles[i] = "url(img/tiles_0$i.jpg)";
    }
}

String getRandomTile() {
    var num = rand.nextInt(10);
    var imgUrl = tiles[num];
    while (imgUrl == null) {
        num = rand.nextInt(10);
        imgUrl = tiles[num];
    }
    tiles.remove(num);
    return imgUrl;
}
```

4. The code for each of the Dnd event handlers is as follows:

```
void _onDragStart(MouseEvent event) {
    Element dragTarget = event.target;
    dragTarget.classes.add('moving');
    dragSource = dragTarget;
    event.dataTransfer.effectAllowed = 'move';
    // event.dataTransfer.setData('text/html', dragTarget.
innerHTML);
}

void _onDragEnd(MouseEvent event) {
    Element dragTarget = event.target;
    dragTarget.classes.remove('moving');
    for (var col in cols) {
        col.classes.remove('over');
    }
}

void _onDragEnter(MouseEvent event) {
```

```
Element dropTarget = event.target;
dropTarget.classes.add('over');
}

void _onDragOver(MouseEvent event) {
    // This is necessary to allow us to drop.
    event.preventDefault();
    event.dataTransfer.dropEffect = 'move';
}

void _onDragLeave(MouseEvent event) {
    Element dropTarget = event.target;
    dropTarget.classes.remove('over');
}

void _onDrop(MouseEvent event) {
// Stop the browser from redirecting and bubbling up the      //
event:
    event.stopPropagation();
    // Don't do anything if dropping onto the same tile we're
dragging.
    Element dropTarget = event.target;
    if (dragSource != dropTarget) {
        var swap_image = dropTarget.style.backgroundImage;
        dropTarget.style.backgroundImage = dragSource.style.
backgroundImage;
        dragSource.style.backgroundImage = swap_image;
    }
}
```

How it works...

Drag-and-drop makes an element on a page draggable by working with the following event life cycle: Drag, DragStart, DragEnd, DragEnter, DragLeave, DragOver, and finally Drop. They generate a stream of `MouseEvents` caught by the `onDragEvent` event handlers (where the event is Start, End, and so on). The `event.target` option changes for each type of event, depending on where we are in the drag-and-drop event model.

The basic requirement for an element to move is that it has the attribute `draggable="true"`. In drag-and-drop, we can distinguish three objects:

- ▶ The source element is where the drag originates (this can be an image, list, link, file object, or block of HTML code)

- ▶ The data payload is what we're trying to drop
- ▶ The target element is an area to catch the drop or the drop zone, which accepts the data the user is trying to drop

Nearly everything can be drag enabled, including images, files, links, or other DOM nodes. However, not all elements can be targets, for example images, but our example shows that it is easy to work around that.

The visual effects that accompany drag-and-drop are implemented through CSS. Every tile has the CSS class `pict`, which contains the attribute `cursor: move`, which gives users a visual indicator that something is moveable.

When a drag action is initiated, `_onDragStart` is executed, adding the CSS class `moving` to our tiles. This class can be found in `drag_drop.css`:

```
.pict.moving {  
    opacity: 0.25;  
    -webkit-transform: scale(0.8);  
    -moz-transform: scale(0.8);  
    -ms-transform: scale(0.8);  
    transform: scale(0.8);  
}
```

We see that the opacity is reduced to 0.25 and size is scaled with factor 0.8, exactly what we see when a drag operation is started. In this example, we don't need it, but if the dragging involves HTML text, `_onDragStart` should contain the following code line:

```
event.dataTransfer.setData('text/html', dragTarget.innerHTML);
```

The preceding code indicates what data will be transferred during the drag-and-drop process. The `dataTransfer` property is where it all happens; it stores the piece of data sent in a drag action. The `dataTransfer` option is set in the `DragStart` event and read/handled in the `Drop` event. Calling `e.dataTransfer.setData(format, data)` will set the object's content to the MIME type and the data payload will be passed as arguments.

The `_onDragEnter` function is executed when we hover over another tile; this adds the CSS class `over`, making the borderlines dashed:

```
.column.over {  
    border: 2px dashed #000;  
}
```

Finally, the `_onDrop` event swaps the tile images. If HTML text has to be swapped, the following lines should be added:

```
dragSource.innerHTML = dropTarget.innerHTML;  
dropTarget.innerHTML = event.dataTransfer.getData('text/html');
```

The article at <http://www.html5rocks.com/en/tutorials/dnd/basics/> by Eric Bidelman is a good resource, but it uses JavaScript to explain the DnD event model. To make it easier to work with DnD in Dart, Marco Jakob developed the library `dart-html5-dnd`, available on pub. For more information on this nice package, refer to <http://code.makery.ch/dart/html5-drag-and-drop/>. DnD functionality is also implemented in the `dart:svg` library.

See also

- ▶ See the *Enabling touch events* recipe in this chapter to learn how to implement drag-and-drop using touch

Enabling touch events

Drag-and-drop is very handy on mobile devices where we don't have a mouse connected; we only have our fingers to interact with the screen. This recipe will show you how to add interactivity via touch events to your web app. The way to do this is very similar to the previous recipe. We will reuse the drag-and-drop example.

How to do it...

Let's look at the `touch` project as explained in the following steps:

1. Prevent zooming with the following `<meta>` tag in `touch.html`:

```
<meta name="viewport"  
      content="width=device-width, initial-scale=1.0, user-  
      scalable=no">
```

We use the same `<div>` structure, each with `class = "draggable"`, as shown in the previous recipe.

2. In `touch.dart`, we make a class, `Touch`, to contain the code specific to the touch events. An object, `touch`, is instantiated and the `init()` method is called, which prepares the board (each tile gets a different background image) and binds the touch events to event handlers:

```
void main() {  
    var touch = new Touch();  
    touch.init();  
}  
  
class Touch {
```

```
Element dragSource;
Map tiles;
var rand = new Random();
var cols = document.querySelectorAll('.pict');

void init() {
    generateMap();
    for (var col in cols) {

        col
            ..onTouchStart.listen(_onTouchStart)
            ..onTouchEnd.listen(_onTouchEnd)
            ..onTouchMove.listen(_onTouchMove)
            ..style.setProperty("background-image", getRandomTile(), "")
            ..style.setProperty("background-repeat", "no-repeat", "")
            ..style.setProperty("background-position", "center", "");
    }
}
```

3. The following is the code for the TouchStart event:

```
void _onTouchStart(TouchEvent event) {
    event.preventDefault(); //stop scrolling by default
    Element dragTarget = event.target; //capture drag target
    //add style to element to indicate its moving
    dragTarget.classes.add('moving');
    dragSource = dragTarget;
}
```

4. The following is the code for TouchMove:

```
void _onTouchMove(TouchEvent event) {
    event.preventDefault();
    Element dropTarget = event.target;
    dragSource.classes.add('moving');
    // Get the current x,y position of the first finger touch      //
    and find the element it is over
    dropTarget = document.elementFromPoint(event.touches[0].page.x, event.touches[0].page.y);
    // If the finger is over an element indicate that in the UI
    if (dropTarget != null) {
        dropTarget.classes.add('over');
    }
}
```

5. The following is the code for TouchEnd:

```
void _onTouchEnd(TouchEvent event) {  
    event.stopPropagation();  
    event.preventDefault();  
    // Don't do anything if dropping onto the same tile we're      //  
    dragging.  
    Element dropTarget = event.target;  
    if (dragSource != dropTarget) {  
        var swap_image = dropTarget.style.backgroundImage;  
        dropTarget.style.backgroundImage = dragSource.style.  
backgroundImage;  
        dragSource.style.backgroundImage = swap_image;  
    }  
}
```

How it works...

The `meta` tag is necessary on mobile devices, and `user-scalable=no` will prevent the device from zooming in/out of the web page. We want DnD in our example and the finger gestures should not be mingled with zooming.

The three main touch events that we will have to handle in our code are `touchStart`, `touchEnd`, and `touchMove`. They are defined in `Element`, so touch can be used for nearly everything on the page. They generate a stream of `TouchEvent`s, and they contain the following three (read only) `Touch` objects:

- ▶ `touches`: These are all the current contact points with the touch surface, that is, fingers on the screen
- ▶ `changedTouches`: These are points of contact whose states changed between the previous touch event and this one
- ▶ `targetTouches`: These are all the current contact points with the surface and also all touches started on the same element which are the target of the event

This list are of type `TouchList`. A `Touch` object has, among other properties, a position given by `page.x` and `page.y`, and a `target` element.

The three `TouchEvent` handlers start with `e.preventDefault()`; this stops the browsers default scrolling behavior. We don't want scrolling, we want dragging here. The element that is dragged is styled with `dragTarget.classes.add('moving')`. The area where the drop could take place is continually monitored in the `TouchMove` handler with the following code:

```
dropTarget = document.elementFromPoint(event.touches[0].page.x, event.  
touches[0].page.y);
```

The element in that position is assigned the CSS class `over`. In the `TouchEnd` handler, the background images of the source and target element are switched.

There's more...

If you want a scenario where the element that is touched and dragged performs a certain movement, you can capture its coordinates in the `TouchStart` handler with the following code:

```
if (event.touches.length > 0) {  
    touchStartX = event.touches[0].page.x;  
}
```

In `TouchMove`, you move the element, probably as a function of the difference between `newtouchX` and `touchStartX`:

```
if (touchStartX != null && event.touches.length > 0) {  
    int newTouchX = event.touches[0].page.x;  
    if (newTouchX > touchStartX) {  
        moveElement(newTouchX - touchStartX);  
    }  
}
```

[ Alternatively, if working on a canvas element, you could start drawing by making use of the touch coordinates. See the Dart site for the complete example code for such scenarios at https://www.dartlang.org/samples/#touch_events.]

See also

- ▶ See the *Responsive Design* recipe in this chapter for more information on the `<meta>` tag used

Creating a Chrome app

You can build native-like apps with the Dart web technology through Chrome apps. Such apps can be large for some browsers and appear to be part of the surrounding operating system; they run offline by default, and you can make them available from the Chrome Web Store. This recipe will show you how to start working with this type of app.

How to do it...

Perform the following steps to create a Chrome app. The code for this recipe can be found in the project `chrome_pack`:

1. Start with creating a new Dart project and choose the **Chrome packaged application** template.
2. The `Pub Get` command starts automatically, and installs the Chrome package from the pub.
3. The first thing you will notice is the presence of a new file `manifest.json`, with the following initial content:

```
{  
  "name": "Chrome pack",  
  "version": "1",  
  "manifest_version": 2,  
  "icons": {"128": "dart_icon.png"},  
  "app": {  
    "background": {  
      "scripts": ["background.js"]  
    }  
  }  
}
```

We don't have to change anything here, but you can add a "description" tag if you like.

4. The `background.js` file indicates which page the application should start with:

```
chrome.app.runtime.onLaunched.addListener(function(launchData) {  
  chrome.app.window.create('chrome_pack.html', {  
    'id': '_mainWindow', 'bounds': {'width': 800, 'height': 600 }  
  });  
});
```

5. The `chrome_pack.html` file has a `<p>` tag with the ID `text_id` and references the script `chrome_pack.dart`. This script first has to import the Chrome package:

```
import 'dart:html';  
import 'package:chrome/chrome_app.dart' as chrome;  
  
int boundsChange = 100;  
var txtp = querySelector("#text_id");  
int n = 0;
```

6. In `main()`, we get information about the platform the app is being executed in; in our case, this displays `{arch: x86-32, nacl_arch:x86-64, os: win}` and we register `rewriteText` as a Click event handler:

```
void main() {  
    chrome.runtime.getPlatformInfo().then((var m) {  
        txtpt.text = m.toString();  
    });  
    txtpt.onClick.listen(rewriteText);  
}
```

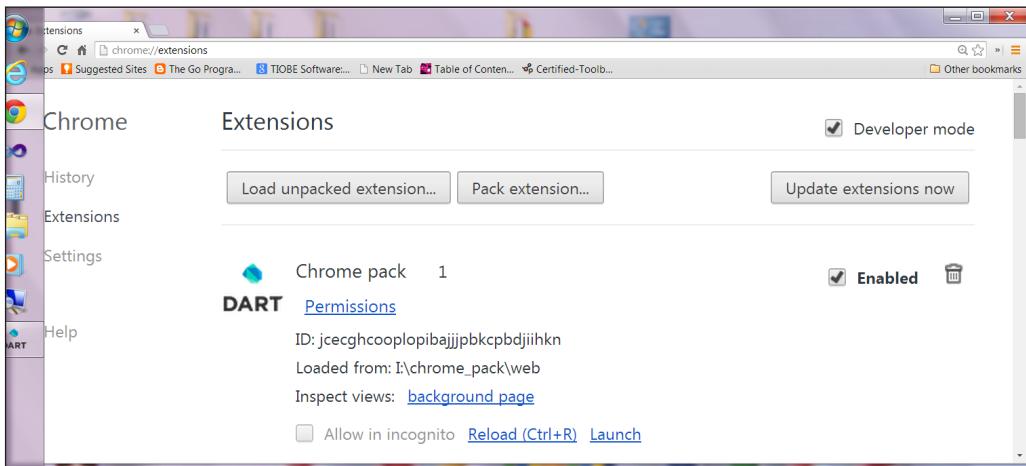
7. At each click event, we show a message with the number of times clicked in the `<p>` tag and we resize the window:

```
void rewriteText(MouseEvent e) {  
    txtpt.text = "Hey, you clicked ${n++} times on a Chrome packaged  
app!";  
    resizeWindow(e);  
}  
  
void resizeWindow(MouseEvent e) {  
    chrome.ContentBounds bounds = chrome.app.window.current().  
getBounds();  
    bounds.width += boundsChange;  
    bounds.left -= boundsChange ~/ 2;  
    chrome.app.window.current().setBounds(bounds);  
    boundsChange *= -1;  
}
```

8. We cannot run this app from Dart Editor as it is; if you try it, you get the exception **Unsupported operation: 'chrome.runtime' is not available.**
9. Instead, we have to compile it to JavaScript. Open a command window and go to the `web` folder of your app. The easiest way to do this is to select the `web` folder in Dart Editor, click on the right mouse button and select **Copy File Path**. You can paste this path on your command line after `cd` and press `Enter`. Then, type the command `dart2js chrome_pack.dart -o chrome_pack.dart.js`.
10. The option `-o` allows you to give an output file name. Alternatively, from within Dart Editor, on the Dart file navigate to **Tools | Pub Build**.

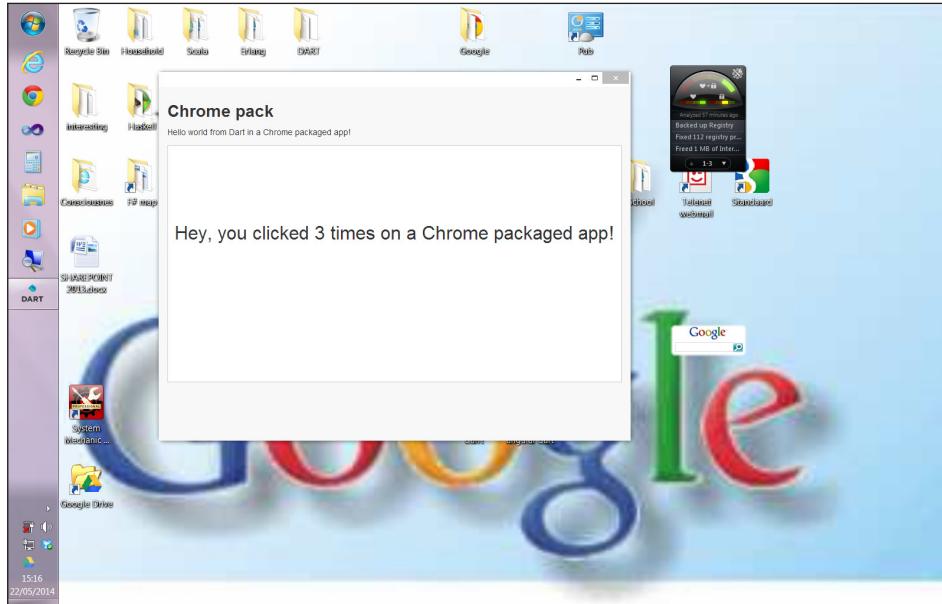
Handling Web Applications

11. Now, we have to configure Chrome to install our app as a packaged app. Start chrome://extensions in the Chrome browser and check the **Developer mode**. Then, click on the **Load unpacked extension** button and browse to the folder containing the manifest.json file, and the app will be installed in Chrome, as shown in the following screenshot:



Loading an app in Chrome

12. Click on the **Launch** link to start the app, as shown in the following screenshot:



A running Chrome-packaged app

13. When you want to deploy your app to the Chrome Webstore, just click on **Pack Extension**.
14. The following are the steps to be followed when you make changes to the code:
 - ❑ Save your changes
 - ❑ Regenerate the JavaScript using dart2js (either from within Dart Editor or from the command line)
 - ❑ Click on the **reload** button against your app's entry

How it works...

The `chrome` package was made by members of the Dart team to provide interoperability from Dart to the Chrome APIs.

The `manifest.json` file is the app's configuration file. It tells Chrome which background JavaScript file to run on starting up (`app` | `background` | `scripts` | `background.js`), which icons to use, which permissions the app has, and so on.

In the ninth step, `dart2js` compiles your Dart script to the JavaScript file indicated after `-o`. Configuring your app as a Chrome-packaged app, as shown from the tenth step to the twelfth, is straightforward.

There's more...

Chrome apps run inside a Chrome process separate from the browser, so they look like any other native application. Compared to pure web apps, they have greater access to the underlying hardware, such as the file system, USB or serial ports, socket-level protocols, and so on via the `chrome.*` API libraries.

You cannot use `window.localStorage` in this type of app, but you can use `chrome.storage` to get an API with similar features (and more); the following is a code snippet that sets a value for a key, and gets the value back:

```
chrome.storage.local.set({'key': 'val'})  
chrome.storage.local.get(['key'])
```

Because of the extensive use of Chrome-specific functionalities in these apps, it is generally not possible to reuse your Chrome-packaged app code in a normal web application.

See also

- ▶ The chrome package is documented at http://dart-gde.github.io/chrome.dart/index.html#chrome/chrome_app.
- ▶ Learn more about Chrome-packaged apps at https://developer.chrome.com/apps/about_apps
- ▶ For more information on dart2js, see the *Compiling your app to JavaScript* recipe in *Chapter 1, Working with Dart Tools*

Structuring a game project

Board games, for the greater part, have the same structure; what the user sees (the view) is a surface containing a grid of cells (model classes) that can be of different shapes. Along with a few utility classes to choose a color at random, this constitutes a solid base to build a board game. With his experience in building game projects, Dzenan Ridjanovic has extracted this game structure in the `boarding` project, which can be found at <https://github.com/dzenanr/boarding>.

How to do it...

Download the game project as a zip from the preceding URL, unzip it, and open it in Dart Editor. The code that is the starting part of a new board game can be found in the `lib` folder; the library that is `boarding` (in `boarding.dart`) imports the view classes `Surface` and `Shape`:

```
library boarding;

import 'dart:html';
import 'dart:math';

import 'package:boarding/boarding_model.dart';

part 'view/shape.dart';
part 'view/surface.dart';
```

The script `boarding_model.dart` declares the library `boarding_model`, which imports the model classes `Grid`, `Cell`, `Cells`, and some utility methods:

```
library boarding_model;

import 'dart:math';

part 'model/cell.dart';
```

```
part 'model/grid.dart';

part 'util/color.dart';
part 'util/random.dart';
```

By building upon these classes, you can build the specific board game you want. Concrete implementations can be found in the folder example that contains a memory game and a tic-tac-toe game (ttt). For example, the memory game start up script imports two libraries, and adds its own specific Memory class, which inherits from the Grid and Board classes that extend Surface:

```
library memory;

import 'dart:async';
import 'dart:html';
import 'package:boarding/boarding_model.dart';
import 'package:boarding/boarding.dart';

part 'model/memory.dart';
part 'view/board.dart';

playAgain(Event e) {
    window.location.reload();
}

main() {
    new Board(new Memory(4), querySelector('#canvas')).draw();
    querySelector('#play').onClick.listen(playAgain);
}
```

How it works...

The Surface class has a draw() method, which draws lines (if needed), and cells:

```
draw() {
    clear();
    if (withLines) lines();
    cells();
}
```

The classes Circle, Rectangle, Square, Line, and Tag (for a text) in shape.dart know how to draw themselves. The Grid class in the model folder knows how to construct itself with the objects of class Cell. The cell.dart file has the code for the class Cell, which can find out if two cells intersect, and it also has a collection of cells. Use this project as a starting point to build your own games!

There's more...

Another way of doing this is by using the `animationFrame` method from the `window` class. With this technique, we start `gameLoop` in the `main()` function and let it call itself recursively, as shown in the following code:

```
main() {
    // redraw
    window.animationFrame.then(gameLoop) ;
}

gameLoop(num delta) {
    // animation code;
    window.animationFrame.then(gameLoop) ;
}
```

See also

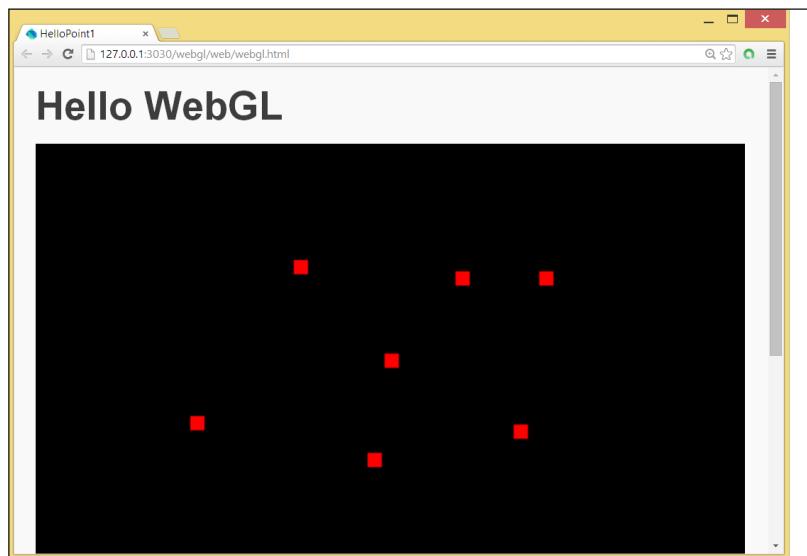
- ▶ Take a look at the pub package `game_loop` from John McCutchan (https://github.com/johnmccutchan/game_loop). Another popular package is StageXL developed by Bernhard Pichler (www.stagexl.org). It is intended for Flash/ActionScript developers who want to migrate their projects as well as their skills to HTML5; visual effects, animation, sound—it's all there.

Using WebGL in your app

The HTML5 Canvas API allows you to draw in only two dimensions. Using **Web Graphics Library (WebGL)**, you can show interactive 2D graphics and 3D graphics within any modern web browser (Internet Explorer 11 has only partial support for WebGL) without the use of plugins. WebGL elements are drawn on a canvas element, and can be combined with other HTML elements. Programs that use WebGL are a mixture of Dart (or JavaScript) code for control, and are of specific WebGL shader code. This shader code is executed on the computer's **Graphic Processing Unit (GPU)**, allowing GPU-accelerated usage of physics effects and image processing as part of the web page canvas, so we have real parallel processing here!

WebGL provides a low-level 3D API; mastering it needs more than a recipe, probably a course or book on its own. However, this recipe will provide you with the basics in the `webgl` project and we point to some links to get further information.

Look at the code of the project `webgl`. When executed, we see a rectangular red point on a black surface. When we click on the surface, new points are shown:



Drawing with WebGL

How to do it...

Start using WebGL by performing the following steps:

1. First, go to <http://get.webgl.org/> to determine if your browser and GPU support WebGL; you should see a spinning cube.
2. The `webgl.html` page simply defines a `<canvas>` tag on which we will draw, using the following code:

```
<canvas id="webgl" style="border: none;" width="500"  
height="500"></canvas>
```

3. To use WebGL in the code, import the `dart_webgl` package:

```
import 'dart:web_gl';
```

4. Now define the shader code for the drawing:

```
// Vertex shader program  
var VSHADER_SOURCE = '''attribute vec4 a_Position;\n    void main() {\n        gl_Position = a_Position;\n        gl_PointSize = 10.0;\n    }\n''';
```

```
// Fragment shader program
var FSHADER_SOURCE = '''void main() {\n    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n} // Set the point\ncolor\n}''';
```

To set the vertex coordinates of the point to a fixed value, use the following code:

```
gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
```

5. The `main()` function starts by getting a reference to the canvas and the 3D rendering context:

```
void main() {
    // Retrieve <canvas> element
    var canvas = querySelector("#webgl");
    if (canvas == null) {
        print('Failed to retrieve the <canvas> element');
    }
    // Get the rendering context for WebGL
    RenderingContext gl = canvas.getContext3d();
    if (gl == null) {
        print('Failed to get the rendering context for WebGL');
        return;
    }
}
```

6. Then, the shader code must be compiled and linked to the code:

```
// compiling the GPU code
Shader fragShader = gl.createShader(FRAGMENT_SHADER);
gl.shaderSource(fragShader, FSHADER_SOURCE);
gl.compileShader(fragShader);

Shader vertShader = gl.createShader(VERTEX_SHADER);
gl.shaderSource(vertShader, VSHADER_SOURCE);
gl.compileShader(vertShader);

Program program = gl.createProgram();
gl.attachShader(program, vertShader);
gl.attachShader(program, fragShader);
gl.linkProgram(program);

if (!gl.getProgramParameter(program, LINK_STATUS)) {
    print("Could not initialise shaders");
    return;
}
gl.useProgram(program);
```

7. We then get an index to the location in a program of the named attribute variable `a_Position`:

```
var a_Position = gl.getAttribLocation(program, 'a_Position');
if (a_Position < 0) {
    print('Failed to get the storage location of a_Position');
    return;
}
```

8. We register the event handler to be called when the mouse is clicked:

```
canvas.onMouseDown.listen((ev) => click(ev, gl, canvas, a_Position));
```

9. Specify the color to clear `<canvas>`. Clear the canvas and draw the point:

```
gl.clearColor(0.0, 0.0, 0.0, 1.0);
gl.clear(COLOR_BUFFER_BIT);
gl.drawArrays(POINTS, 0, 1);
}
```

10. The click handler uses an array `g_points` to remember the mouse click positions:

```
List<num> g_points = new List<num>();

void click(ev, RenderingContext gl, canvas, a_Position) {
    var x = ev.clientX; // x coordinate of a mouse pointer
    var y = ev.clientY; // y coordinate of a mouse pointer
    var rect = ev.target.getBoundingClientRect();

    x = ((x - rect.left) - canvas.width / 2) / (canvas.width / 2);
    y = (canvas.height / 2 - (y - rect.top)) / (canvas.height / 2);

    // Store the coordinates to g_points array
    g_points.add(x);
    g_points.add(y);

    gl.clear(COLOR_BUFFER_BIT);

    var len = g_points.length;
    for (var i = 0; i < len; i += 2) {
        // Pass the position of a point to a_Position variable
        gl.vertexAttrib3f(a_Position, g_points[i],
            g_points[i + 1], 0.0);
        gl.drawArrays(POINTS, 0, 1);
    }
}
```

How it works...

The shader code in the fourth step consists of a vertex shader program (to draw the shape boundaries) and fragment shader (for colors, texturing, and lighting) program; they are hard coded in strings assigned to the constants `VSHADER_SOURCE` and `FSHADER_SOURCE`.

The piece of code in the sixth step looks daunting, but don't worry, this code can simply be reused in other drawings.

Step 7 is necessary to make a connection between the `a_position` variable in the shader code, and the variable with the same name in Dart. Notice that the click event handler in steps 8 and 10 needs the GL context and canvas as second and third parameters.

There's more...

WebGL in itself has no built-in support to load a 3D scene defined in a regular 3D file format. The viewer code or a library such as `three.dart`, which is a port of `Three.js` (get it from the pub as `three` is necessary to display a 3D scene). To create content, use a regular content-creation tool and export the content to a viewer-readable format.

See also

- ▶ View and study the beautiful 3D solar system visualization example made by the Dart team at <https://github.com/dart-lang/dart-samples/tree/master/html5/web/webgl/solar3>
- ▶ The tutorial at <http://www.learnwebgl.com> was rewritten for Dart by John Thomas McDole; find it at <https://github.com/jtmcdoile/dart-webgl>
- ▶ The website to find out more about WebGL at http://www.khronos.org/webgl/wiki/Main_Page
- ▶ If you want to read more about Shaders, visit <http://aerotwist.com/tutorials/an-introduction-to-shaders-part-1/>

Authorizing OAuth2 to Google services

You certainly have already seen websites where you can log in using your Google, Facebook, or Twitter account, instead of having to enter your information all over again. This service is most probably powered by OAuth2, which means (the second version of) the open (web) standard for authorization. It provides secured access to the server side of your application for clients that have been given an access token by a third-party OAuth2 authorization server. The credentials are guaranteed to be verified by the token and are not given to you as the website owner. The Dart team and community have provided us with some nice packages to easily implement this functionality.

How to do it...

If you want to use OAuth2 authentication from Google in a client app, there is the `google_oauth2_client` library. Add `google_oauth2_client` to your `pubspec.yaml` dependencies and let pub in Dart Editor install it, or invoke `pub get` in the command line. Add the following to your script, `import 'package:google_oauth2_client/google_oauth2_browser.dart' ;`, to start working with it in code.

Perform the following steps to use OAuth2 authentication from Google:

1. You need to register at the Google API Console site and create a Client ID by performing the following steps:
 - ❑ Go to <https://console.developers.google.com/project> and create a project, for example, `oauth2-test`; it will be given a project ID.
 - ❑ From the **Project Dashboard**, go to **APIs & auth, Credentials**. Click on the **Create New Client ID** button. Choose **Web application** as the type.
 - ❑ In **Authorized JavaScript Origins**, insert the URL `http://127.0.0.1:3030`, used by the Dart Editor to launch web apps. Click on the **Create Client ID** button and a new screen appears with your client ID.
2. Add an `auth` variable initialized with the Oauth Client ID you just registered:

```
final auth = new GoogleOAuth2(  
    "xxxxxxxxxxxxxxxxxxxxxx.apps.googleusercontent.com", // insert Client  
    id  
    ["openid", "email"],  
    tokenLoaded:oauthReady);
```

3. Add a Log in with Google button in the `main()` code:

```
var logIn = new ButtonElement()  
    ..text = "Log in with Google"  
    ..onClick.listen((_) {  
        auth.login();  
    });  
  
document.body.children.add(logIn);
```

In the same way, you could also provide a logout facility:

```
var logOut = new ButtonElement()  
    ..text = "Log out"  
    ..onClick.listen((_) {  
        auth.logout();  
    });  
  
document.body.children.add(logOut);
```

4. To see what is returned from the authentication, we add the following code:

```
void oauthReady(Token token) {  
    print(token);  
}
```

5. Suppose we want to use that token to access our Google+ profile data and display the user's full name. Go to the project dashboard (refer to step 2), select **Boost your app with a Google API**, then **Enable an API**, and then enable the Google+ API. Go to the API access screen and create a **Public API Access Key**. You will have to insert this value in **plus.key** in step 8.
6. Now add `google_plus_v1_api` to the `pubspec.yaml` file and add the following import line:

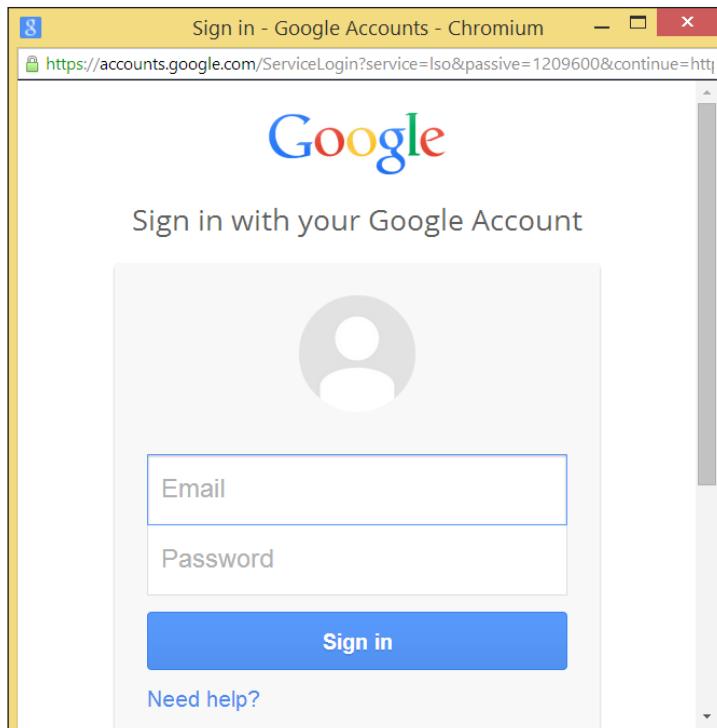
```
import "package:google_plus_v1_api/plus_v1_api_browser.dart"  
      as plusclient;
```

7. Add `plusclient.Plus.PLUS_ME_SCOPE` after the e-mail scope in the `auth` variable.
8. Now add the following code to `oauthReady`:

```
// get the users full name  
var plus = new plusclient.Plus(auth);  
// set the API key  
plus.key = "Axxxxxxxxxxxxxxxxxxxx-x-xxxxxxxxx-x";  
plus.oauth_token = auth.token.data;  
plus.people.get("me").then((person) {  
    // log the users full name to the console  
    print("Hello ${person.name.givenName} ${person.name.  
familyName}");  
});
```

How it works...

Steps 3 to 5 illustrate how to make use of the OAuth client. Clicking on the **login** button will display the Google login screen, where the user has to enter their e-mail address and password, as shown in the following screenshot:



Google account login screen

A screen appears to grant access to your application, based on the scopes you specified in the auth variable ["openid", "email"]. After that, the tokenLoaded event is fired, an OAuth token is returned, and its callback function oauthReady can print it out; it has the following format (sensitive data being replaced by x):

```
[Token type=Bearer,  
data=xxx.xxxx-xxxxxxxxxxxxxxxxxxxxxx,  
expired=false,  
expiry=2014-05-24 15:46:19.445,  
email=xxxxxxxxxxxxxx@gmail.com,  
userId=xxxxxxxxxxxxxxxxxxxxxxxxx]
```

This token will be sent to your web application on the server. To summarize, the user is authenticated to your application as an existing Google account through the OAuth2 protocol.

From step 7 onwards, we use our token to log in to Google+. In step 9, auth.token is given to the Google+ client, and this prints out givenName and familyName from the Google+ service, for example, **Hello John Doe**.

In general, if you want to access a certain URL `urlxyz` through OAuth2 and you already have an auth token, use the following code:

```
var request = new HttpRequest();
request.onLoad.listen(...);
request.open(method, urlxyz);
auth.authenticate(request).then((request) => request.send());
```

There's more...

The Dart team has made the `oauth2` client library, which allows you to obtain OAuth2 authorization from a non-Google server. With it, a user is authenticated for your app, without having to store passwords on your website. After the user has been authenticated, your application has an `oauth2` token for that user, which can be used to access other services. To start working with it in the code, add `oauth2` to your `pubspec.yaml` dependencies and let the pub in Dart Editor install it, or invoke `pub get` on the command line. Then, add the following to your script:

```
import 'package:oauth2/oauth2.dart';
```

The way to go about this is a bit more involved, but with what you have learned now, you will be able to grasp the example code given at <http://pub.dartlang.org/packages/oauth2>.

To authenticate via OAuth2 from Facebook, Windows Live, or Google in a server-side application, use the pub package by Christophe Hurpeau.

See also

- ▶ For more detailed information on OAuth2 and Google, see the article *Using OAuth2 to access Google APIs* at <https://developers.google.com/accounts/docs/OAuth2?csw=1>
- ▶ If you want to learn more about Oauth2, refer to <http://oauth.net/2/>

Talking with JavaScript

If we take into account the enormous amount of JavaScript code and libraries that exist, and are still being developed, it is very important that we have a simple way to use JavaScript code from within Dart applications, in particular to get access from Dart to the JavaScript code that is running in the same web page. The earliest attempts used `window.postMessage`, and then a package called `js` was built. Because of the huge importance of this topic, the Dart team now has provided us with a core library, `dart:js`, to interoperate with JavaScript. This provides better performance, reduces the size of the compiled JavaScript file, and makes it also easier to use. Once `dart:js` is ready, the package `js` has been rewritten to use `dart:js` under the covers.

How to do it...

Take a look at the project js_interop, as explained in the following steps:

1. To start using dart:js in our project, we have to import it in our code:

```
import 'dart:js';
```

2. In js_interop.html, we declare a Dart script, and the JavaScript program js_interop.dart will look into the code of interact.js:

```
<script type="application/dart" src="js_interop.dart"></script>
<script type="application/javascript"
src="interact.js"></script>
```

3. The interact.js file contains the following code: a variable jsvar, a class Person with the properties name and gender, and the methods greeting and sayHello:

```
var jsvar = "I want Dart";

function Person(name, gender) {
    this.name = name;
    this.gender = gender;
    this.greeting = function(otherPerson) {
        alert('I greet you ' + otherPerson.name);
    };
}

Person.prototype.sayHello = function () {
    alert ('hello, I am ' + this.name );
};
```

4. First, we get the contents of a JavaScript variable:

```
var dart = context['jsvar'];
print(dart); // I want Dart
```

5. Then, we make a Person object:

```
var pers1 = new JsObject(context['Person'], ['An',
'female']);
var pers2 = new JsObject(context['Person'], ['John',
'male']);
```

6. We access and set the properties using the following code:

```
print(pers1['name']); // An  
print(pers2['gender']); // male  
pers2['gender'] = 'female';  
print(pers2['gender']); // female
```

7. We call the methods on the Person object:

```
pers1.callMethod('sayHello', []);  
pers2.callMethod('greeting', [pers1]);
```

The preceding steps display alert windows with the messages **hello, I am An** and **I greet you An.**

8. Now we get the global object in JavaScript (normally a window) via context, and display an alert window with callMethod:

```
context.callMethod('alert', ['Hello from Dart!']);
```

9. Use jsify to create a JavaScript object and array:

```
var jsMap = new JsObject.jsify({'a': 1, 'b': 2});  
print(jsMap); // [object Object]  
var jsArray = new JsObject.jsify([1, 2, 3]);  
print(jsArray); // [1, 2, 3]
```

How it works...

The `dart:js` library provides Dart access to JavaScript objects in web applications, not in server applications. More specifically, it exposes wrapped or proxy versions of any JavaScript objects you access. This enables Dart to safely sandbox JavaScript away and prevents its problems from leaking into the Dart application. You can get and set properties and call JavaScript functions and methods on JavaScript objects, while conversions between Dart and JavaScript are taken care of as far as possible. At this moment, the bridge is not fully bidirectional; JavaScript has no access to Dart objects, but it can call Dart functions.



Inclusion of the `<script src="packages/browser/interop.js"></script>` code is no longer needed.

The main type of object is `JsObject` with which we can reach out to JavaScript objects; in other words, we create a Dart proxy object to the JavaScript object. To get the global object in JavaScript (which is mostly `window`), use the top-level getter function `context`; this is used in step 8. However, `context` is also used to get the values of JavaScript variables, as shown in step 4.

You can create JavaScript objects as shown in step 5. Use the `JsObject()` constructor. This takes the name of a JavaScript constructor function and the list of arguments that it needs as arguments. As shown in step 6, we can use the `[]` index operator to get the value of properties and `[] =` to set them; instead of a numerical index, we use the property name string as the key. The seventh and eighth step demonstrate that we can call a JavaScript method on an object with `callMethod`, taking the name of the method and the list of its arguments as parameters. Finally, in step 9, we see that `JsObject.jsify` turns a Dart map into a JavaScript object using the keys as properties; the same method also turns a Dart list into a JavaScript array.

There's more...

To be able to compare, we will now show the same code but rewritten with the `js` package. In `js_interop2.html`, we have the same JavaScript, but running together with the Dart script `js_interop2.dart`. We add the `js` package to our `pubspec.yaml` file as `js: any`, and let `pub get` do its magic. To make the package available to our Dart script, we add the following code to `js_interop2.dart`:

```
import 'package:js/js.dart' as js;
```

Rewriting the Dart code from `js_interop.dart` gets us the following output:

```
void main() {
  // getting a variable:
  var dart = js.context['jsvar'];
  print(dart); // I want Dart
  // making objects:
  var pers1 = new js.Proxy(js.context.Person, ['An', 'female']);
  var pers2 = new js.Proxy(js.context.Person, ['John', 'male']);
  // accessing and setting properties:
  print(pers1.name); // prints the whole object: [An, female]
  pers1.name = 'Melissa'; // change name property
  print(pers1.name); // Melissa
  // calling methods:
  pers1.sayHello.call(); // window: hello, I am Melissa
  pers2.greeting.call(pers1); // window: I greet you Melissa
  // getting the global object in JavaScript via context
  js.context.alert('Hello from Dart via JavaScript');
  // using jsify:
  var jsMap = js.map({'a': 1, 'b': 2});
  print(jsMap); // [object Object]
  var jsArray = js.array([1, 2, 3]);
  print(jsArray); // [1, 2, 3]
}
```

The syntax is a bit easier than `dart:js` but because the names in the `js` package cannot be minified since it uses `dart:mirrors` and `noSuchMethod`, using this library can result in a noticeable increase in code size when compiled to JavaScript. If this is a big disadvantage for you, use `dart:js` instead. We use the `js` package in the next recipe to talk to the Google Visualizations API.

See also

- ▶ See the *Using JavaScript libraries* recipes for more information on how to use JavaScript libraries
- ▶ A small library that makes it easy to call Dart from Javascript is available at https://github.com/jptrainor/js_bridge, it's a thin layer around `dart:js`.

Using JavaScript libraries

In this recipe, we use the `js` package as an interface from our Dart script to the Google Chart JavaScript API. This gives us many rich and highly customizable ways to graphically represent data in our Dart web apps and, because it is built with HTML5/SVG, it works cross-browser (even for older IE versions) and cross-platform (also for iOS and Android).

How to do it...

Take a look at the project `googlechart`:

1. In the `<body>` tag of the HTML file, place the following:

```
<div id="chart" style="width: 900px; height: 500px;"></div>
<script type="text/javascript"
       src="https://www.google.com/jsapi"></script>
<script type="application/dart"
       src="googlechart.dart"></script>
```

The `<div>` tag with the ID `chart` is where the chart will be drawn; the code to do this is contained in `googlechart.dart`.

2. The following is the data we want to represent in a chart:

```
var listData = [
    ['Year', 'Sales', 'Expenses'],
    ['2004', 1000, 400],
    ['2005', 1170, 460],
    ['2006', 660, 1120],
    ['2007', 1030, 540]
];
```

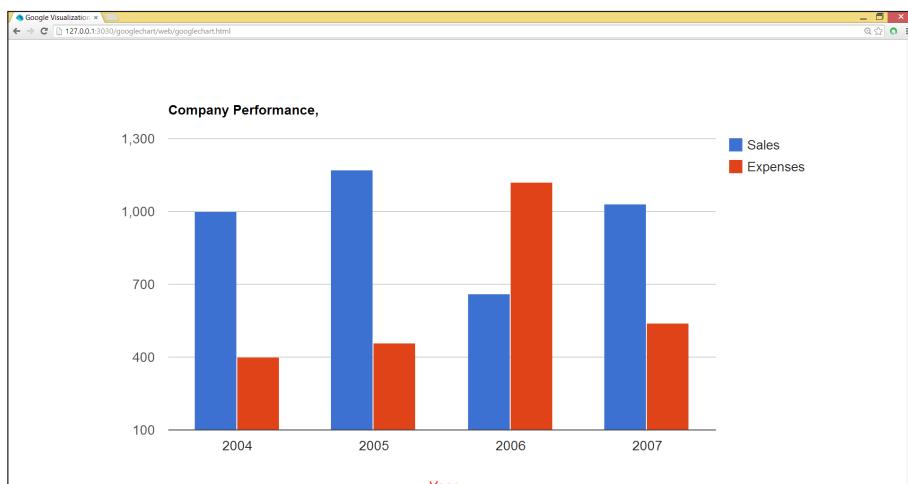
3. After importing the `js` package, we load the `corechart` package from the Google API and tell our code to execute the method `drawChart` when this is done:

```
import 'dart:html';
import 'package:js/js.dart' as js;
main() {
  js.context.google.load('visualization', '1', js.map(
    { 'packages': ['corechart'],
      'callback': drawChart,
    }));
}
```

4. Calling the `drawChart` method then loads the data and draws the chart:

```
void drawChart() {
  var gviz = js.context.google.visualization;
  var arrayData = js.array(listData);
  var tableData = gviz.arrayToDataTable(arrayData);
  var options = js.map({
    'title': 'Company Performance, ',
    'hAxis': {'title': 'Year', 'titleTextStyle': {'color': 'red'}}
  });
  var chart = new js.Proxy(gviz.ColumnChart,
    querySelector('#chart'));
  chart.draw(tableData, options);
}
```

Performing the previous steps gives the following output in the browser:



Using Google Charts with js

Also, notice that when hovering over the columns, a tooltip is shown with the exact data for that column.

How it works...

In the first step, the highlighted script tag refers to the online Google visualization libraries that have to be loaded dynamically before you can use them to draw charts. Step 2 defines the data; this could have been loaded from a file or database. In step 4, we get a reference `gviz` to the JavaScript Google Visualizations object via `js.context`. Then, the data is transformed into a JavaScript array with `js.array`; the chart options such as title, x axis, color, and so on are passed via `js.map`. Then, we construct a proxy chart to the JavaScript object, on which the `draw` method is invoked.

So basically, there are four steps to create a chart:

1. Load the `jsapi` library.
2. List the data.
3. Configure the options.
4. Create the chart.

See also

- ▶ Want to learn more about Google Charts? Then visit <https://google-developers.appspot.com/chart/interactive/docs/>, where you can find a wealth of examples. These are written in JavaScript, but with what you now know, you can easily translate them to Dart.

6

Working with Files and Streams

In this chapter, we will cover the following recipes:

- ▶ Reading and processing a file line by line
- ▶ Writing to a file
- ▶ Searching in a file
- ▶ Concatenating files
- ▶ Downloading a file
- ▶ Working with blobs
- ▶ Transforming streams

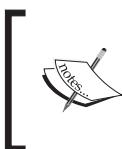
Introduction

Working with files is the bread and butter of every programming language when reaching out for data in the environment. The classes and methods dealing with this functionality can be found in the `dart:io` package, together with support for networking (sockets and HTTP). This package can only be used in Dart command-line applications, not in browser apps, so our code runs in a Dart VM.

When working with files, and I/O in general, there are two modes of operation:

- ▶ Synchronous operations, where code execution waits for the I/O result
- ▶ Asynchronous operations, where the code execution is not blocked and continues while I/O is taking place

Because the Dart VM is single threaded, a synchronous call blocks the application. So, for scalability reasons, the asynchronous way is the best practice using the `Future` and `Stream` classes from the `dart:async` package. Most methods on files come in pairs, the asynchronous and the synchronous versions, such as `copy` and `copySync`. Unless you really have to wait for the result, use the asynchronous way so that screens and apps do not appear to be blocked and can still respond.



If you need a recap, visit the tutorials at <https://www.dartlang.org/docs/tutorials/futures/> and <https://www.dartlang.org/docs/tutorialsstreams/>. In this chapter, we will focus on recipes to handle files.

Reading and processing a file line by line

Files containing data in the **comma separated values (csv)** format are structured so that one line contains data about one object, so we need a way to read and process the file line by line. As an example, we use the data file `winequality-red.csv`, that contains 1,599 sample measurements, 12 data columns, such as **pH** and **alcohol**, per sample, separated by a semicolon (;), of which you can see the top 20 in the following screenshot:

1	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
2	7.4	0.7		0.19	0.076	11	34	0.9978	3.51	0.56	9.4	5
3	7.8	0.88		0.26	0.098	25	67	0.9968	3.2	0.68	9.8	5
4	7.8	0.76	0.04	2.3	0.092	15	54	0.997	3.26	0.65	9.8	5
5	11.2	0.28	0.56	1.9	0.075	17	60	0.998	3.16	0.58	9.8	6
6	7.4	0.7		0.19	0.076	11	34	0.9978	3.51	0.56	9.4	5
7	7.4	0.66		0.18	0.075	13	40	0.9978	3.51	0.56	9.4	5
8	7.9	0.6	0.06	1.6	0.069	15	59	0.9964	3.3	0.46	9.4	5
9	7.3	0.65		0.12	0.065	15	21	0.9946	3.39	0.47		10
10	7.8	0.58	0.02		2.073	9	18	0.9968	3.36	0.57	9.5	7
11	7.5	0.5	0.36	6.1	0.071	17	102	0.9978	3.35	0.8	10.5	5
12	6.7	0.58	0.08	1.8	0.097	15	65	0.9959	3.28	0.54	9.2	5
13	7.5	0.5	0.36	6.1	0.071	17	102	0.9978	3.35	0.8	10.5	5
14	5.6	0.615		0.16	0.089	16	59	0.9943	3.58	0.52	9.9	5
15	7.8	0.61	0.29	1.6	0.114	9	29	0.9974	3.26	1.56	9.1	5
16	8.9	0.62	0.18	3.8	0.176	52	145	0.9986	3.16	0.88	9.2	5
17	8.9	0.62	0.19	3.9	0.17	51	148	0.9986	3.17	0.93	9.2	5
18	8.5	0.28	0.56	1.8	0.092	35	103	0.9969	3.3	0.75	10.5	7
19	8.1	0.56	0.28	1.7	0.368	16	56	0.9968	3.11	1.28	9.3	5
20	7.4	0.59	0.08	4.4	0.086	6	29	0.9974	3.38	0.5		9
												4

How to do it...

Examine the code of the command-line project processing_lines using the following methods:

1. Using the `readAsLines` method as shown in the following code:

```
import 'dart:io';
// for step 3:
import 'dart:async';
import 'dart:convert';

main() {
    File data= new File("../winequality-red.csv");
    data.readAsLines().then(processLines)
        .catchError((e) => handleError(e));
}

processLines(List<String> lines) {
    // process lines:
    for (var line in lines) {
        print(line);
    }
}

handleError(e) {
    print("An error $e occurred");
}
```

The previous code gives the following output:

```
"fixed acidity";"volatile acidity";"citric acid";"residual sugar";"chlorides";"free sulfur dioxide";"total sulfur dioxide";
7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5: 48 bytes
7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5: 48 bytes
7.8;0.76;0.04;2.3;0.092;15;54;0.9973;2.6;0.65;9.8;5: 51 bytes
11.2;0.28;0.56;1.9;0.075;17;60;0.998;3.16;0.58;9.8;6: 52 bytes
7.4;0.7;0;1.9;0.076;11;34;0.9978;3.51;0.56;9.4;5: 48 bytes
7.4;0.66;0;1.8;0.075;13;40;0.9978;3.51;0.56;9.4;5: 49 bytes
7.9;0.6;0.061;1.6;0.069;15;59;0.9964;3.3;0.46;9.4;5: 50 bytes
7.3;0.65;0;1.2;0.065;15;21;0.9946;3.39;0.47;10;7: 48 bytes
7.8;0.58;0.02;2;0.073;9;18;0.9968;3.36;0.57;9.5;7: 49 bytes
```

2. Extracting the data of each line to an object as shown in the following code:

```
processLines(List<String> lines) {
    // process lines:
    for (var line in lines) {
        print(line);
        // when not header line, split line on separator:
        if (!header) {
```

```
    List<String> fields = line.split(";");
    Wine wn = new Wine();
    wn.fixed_acidity = fields[0];
    wn.volatle_acidity = fields[1];
    // extracting remaining properties
    wn.alcohol = fields[10];
    wn.quality = fields[11];
    print(wn);
}
header = false;
}
}
class Wine {
    var fixed_acidity;
    var volatle_acidity;
    // other properties
    var alcohol;
    var quality;

    toString() => "This wine has $fixed_acidity fixed acidity, "
    "alcohol % of $alcohol and quality $quality.";
}
```

The preceding code gives the **This wine has 6.8 fixed acidity, alcohol % of 11.3 and quality 6** output.

3. Use the openRead method as an alternative as shown in the following code:

```
main() {
    File data= new File("../winequality-red.csv");
    // using openRead:
    Stream<List<int>> input = data.openRead();
    input
        .transform(UTF8.decoder) // Decode to UTF8.
        // Convert stream to individual lines.
        .transform(const LineSplitter())
        .listen((String line) { // Callback to process results.
            print('$line: ${line.length} bytes');
            // Further processing of line, e.g. as in processLines
        }, onDone: () {
            print('File is now closed.');
        }, onError: (e) {
            print(e.toString());
        });
}
```

The previous code gives the following output:

7.8;0.88;0;2.6;0.098;25;67;0.9968;3.2;0.68;9.8;5: 48 bytes

7.8;0.76;0.04;2.3;0.092;15;54;0.997;3.26;0.65;9.8;5: 51 bytes

How it works...

In step 1, we created a `File` object reference to our data file. On this object, we call the asynchronous `readAsLines` method that returns `Future` with a return value of the type `List<String>`. Each line is read as a string, and all lines form a list. When the file is read in its entirety and this value is returned, it is executed with the callback function `processLines` that effectively gets `List<String>` as its argument. In `processLines`, we can get at each line and transform or process it. If `readAsLines` returns with an error, `catchError` is fired, and `handleError` callback is executed (we could have shortened this line to `catchError(handleError) ;`).

For example, when the file is not found, we have the following message:

An error FileSystemException: Cannot open file, path = 'winequality-red.csv' (OS Error: The system cannot find the specified file. , errno = 2) occurred

In step 2, we split each data field. At that moment, we can create a `Wine` object for each line, start doing calculations with the data, and so on.

The `readAsLines` method takes an optional argument of the type `encoding`, such as `this:readAsLines(encoding: ASCII)`; instead of ASCII, you can use `LATIN1` or `UTF-8`.

In step 3, we see an alternative way to be used when the file is too large to fit in memory (the code line with `readAsLines` is now no longer needed). With `openRead`, the file is read in chunks as a stream of integers. In this stream, we use the `transform` method to convert to `UTF-8` and then to split it into separate lines. The `listen` event then activates a callback function for each line read. The `onDone` option defines a callback function when the last line of the file is read in; `onError` defines an error handler. For this to work, we need to import `dart:async` and `dart:convert`. If you find the syntax used in the `listen` callback a bit too clunky, you can always write it with named event handlers as in `processing_lines2.dart` as follows:

```
// previous lines left out
.listen(processLine, onDone: close, onError: handleError);
}

processLine(line) { print('$line: ${line.length} bytes'); }
close() { print('File is now closed.');?>
handleError(e) { print(e.toString()); }
```

The other possibilities are as follows:

- ▶ Use the method `readAsBytes` if you want to be able to process individual bytes in the file
- ▶ Use the method `readAsString` if you want to read the file in memory as one big string
- ▶ Use the method `readAsLinesSync` if you want to read the file in memory as one big string and wait until that is done

See also

- ▶ See the *Transforming streams* recipe in this chapter.

Writing to a file

In this recipe, we demonstrate the three most important ways to write to a file. You can find the code in the project `writing_files`.

How to do it...

The three ways to write to a file are discussed as follows:

1. First, we import the packages `io` and `convert` as shown in the following code:

```
import 'dart:io';
import 'dart:convert';

void main() {
```

2. We can write to a file using `writeAsString` as shown in the following code:

```
final filename = 'outString.txt';
new File(filename).writeAsString('Dart is an elegant
language').then((File file) {
// do something with the file.
});
```

3. We can write to a file using `writeAsBytes` as shown in the following code:

```
final string = '你好世界';
// Encode to UTF8.
var encodedData = UTF8.encode(string);
new
File('outUTF8.txt').writeAsBytes(encodedData).then((file)
=> file.readAsBytes()).then((data) {
// Decode to a string, and print.
```

```
    print(data);
    // [228, 189, 160, 229, 165, 189, 228, 184, 150, 231, 149, 140]
    print(UTF8.decode(data)); // prints '你好世界'.
});
```

4. We can write to a file using `openWrite` as shown in the following code:

```
var file = new File('out3.txt');
var sink = file.openWrite();
sink.write('File was written to at ${new DateTime.now()}\n');
// close the IOSink to free system resources!
sink.close();
}
```

How it works...

Step 1 uses the asynchronous `writeAsString` method to write one (big) string to a file, and this file is then automatically closed. In the callback function called by `then`, you could, for example, send the file over the network. Step 2 shows how to write raw bytes to a file with `writeAsBytes`. This is necessary when the file contains non-readable or Unicode characters.

There's more...

What do we do when we want to write to our file in chunks? Then, we use the `openWrite` method as shown in step 3. When called on a `File` object, this creates an `IOSink` object for that file, which you can write to with any of the following methods: `write`, `writeln`, `writeCharCode`, `writeAll`. In contrast to the `write` methods of the previous steps, the `IOSink` object must be explicitly closed when no longer needed. The `openWrite` method takes two optional arguments as shown in the following code:

```
file.openWrite(mode: FileMode.APPEND, encoding: ASCII);
```

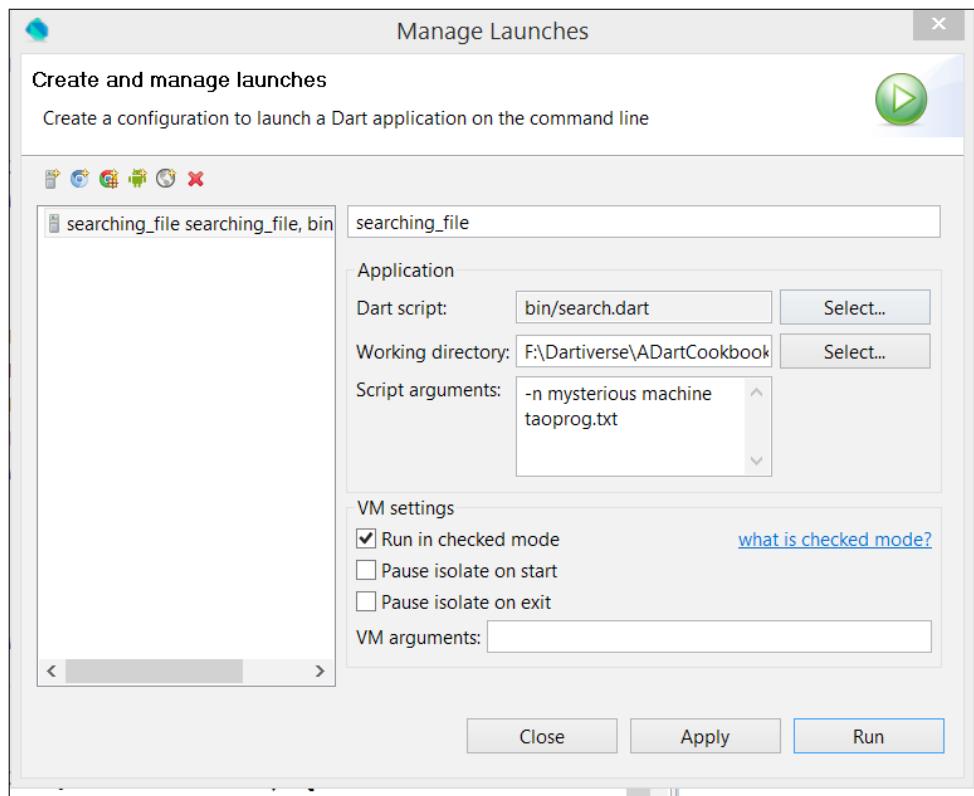
The default mode is `FileMode.WRITE`.

Searching in a file

In this recipe, we demonstrate how to search for certain words in a text file. You can find the code in the `search.dart` script in the project `searching_file`. As an example text file, we use `taoprog.txt`.

How to do it...

The program is launched from the command-line in the bin folder (or in Dart Editor with a Managed Launch with **Script arguments** -n search1 search2taoprog.txt) as shown in the following screenshot:



In dart search.dart -n search1 search2 taoprog.txt , where search1 and search2 are words to be searched for, there can be one or more search words. For example, let's search for *mysterious* and *machine*, in which case, the output is as follows:

```
Observatory listening on http://127.0.0.1:5623
15: Something mysterious is formed, born in the silent void. Waiting
32: The Tao gave birth to machine language. Machine language gave birth
90: The programmers of old were mysterious and profound. We cannot
107: Grand Master Turing once dreamed that he was a machine. When he
110: "I don't know whether I am Turing dreaming that I am a machine, or a
111: machine dreaming that I am Turing!"
229: seeks the simplest harmony between machine and ideas. This is why
306: Why do you expect it from a machine that humans have constructed?
```

The flag `-n` is optional; if included, we see a line number printed in front of the line.

The following is the code from the script:

```
import 'dart:io';
import 'package:args/args.dart';

const HOWTOUSE = 'usage: dart search.dart [-n] search-pattern file';
const LINENO = 'line-number';
ArgResults argResults;
var searchTerms = "";
File file;
```

Perform the following steps to search in a file:

1. We can search a file using the `args` package as shown in the following code:

```
void main(List<String> args) {
    final parser = new ArgParser()..addFlag(LINENO, negatable:
        false, abbr: 'n');
    argResults = parser.parse(args);
    if (argResults.rest.length < 2) {
        print(HOWTOUSE);
        exit(1);
    }
}
```

2. We can search a file by capturing the filename and the search terms as shown in the following code:

```
final strFile = argResults.rest.last;
File file = new File(strFile);
searchTerms = argResults.rest.sublist(0,
    argResults.rest.length - 1);
searchFile(file, searchTerms);
}
```

3. We can search a file by reading in the file and searching, as shown in the following code:

```
searchFile(File file, searchTerms) {
    file.readAsLines().then(searchLines).catchError(print);
}
searchLines(lines) {
    for (var i = 0; i < lines.length; i++) {
        for (var j = 0; j < searchTerms.length; j++) {
            if (lines[i].contains(searchTerms[j])) {
                printMatch(lines[i], i);
            }
        }
    }
}
```

```
        }
    }
}
```

4. We can search a file by printing out the match line found as shown in the following code:

```
void printMatch(String line, int i) {
    StringBuffer sb = new StringBuffer();
    if (argResults[LINENO]) sb.write('${i + 1}: ');
    sb.write(line);
    print(sb.toString());
}
```

How it works...

In step 1, we see that the `args` package is used to parse the command-line arguments. The option `-n` is either there or not (on or off, a Boolean value), which is why it is added as a flag to the `parser` object; `negatable: false` prevents you from writing `no-n` as an argument. We parse and then test to see that we have at least two arguments (a search term and a filename). If not, the string in the constant `HOWTOUSE` is displayed as a help option. Step 2 prepares the scene; the last argument is the filename, the rest of the arguments given by the `sublist` method is the list of search terms.

The actual searching happens in step 3; we use the `readAsLines` method to read the file. When this is done, the callback function `searchLines` is called in a nested for loop, where each line in succession is tested for all search terms as to whether it contains the term. So `printMatch` prints the line out and also whether `-n` specified was preceded by its line number.

See also

- ▶ See the *Parsing command-line arguments* recipe in *Chapter 2, Structuring, Testing, and Deploying an Application*, for more information on using the `args` package at <https://pub.dartlang.org/packages/args>
- ▶ You can find an example that searches recursively through a folder structure at https://code.google.com/p/dart/source/browse/branches/bleeding_edge/dart/samples/dgrep/bin/dgrep.dart

Concatenating files

Let's suppose that we have a number of text files we want to glue together in one big file. This recipe with code in the project `concat_files` shows you how this can be done.

How to do it...

The program is launched from the command line in the `bin` folder (or in Dart Editor with a Managed Launch with Script arguments `file1.txt file2.txt file.txt`) as `dart concat.dart file1.txt file2.txt file.txt`, where `file1.txt` and `file2.txt` are the files to be concatenated (there can be two or more files) into `file.txt`. The following is the code to perform this:

```
import 'dart:io';
import 'package:args/args.dart';

ArgResults argResults;
File output;

void main(List<String> arguments) {
    final parser = new ArgParser();
    argResults = parser.parse(arguments);
    final outFile = argResults.rest.last;
    List<String> files = argResults.rest.sublist(0, argResults.
        rest.length - 1);
    if (files.isEmpty) {
        print('No files provided to concatenate!');
        exit(1);
    }
    output = new File(outFile);
    if (output.existsSync()) {
        output.delete();
    }
    concat(files);
}

concat(List<String> files) {
    for (var file in files) {
        var input = new File(file);
        try {
            var content = input.readAsStringSync();
            content += "\n";
            output.writeAsStringSync(content, mode: FileMode.APPEND);
        } catch (e) {
            print("An error $e occurred");
        }
    }
}
```

How it works...

We use the `args` package to get the output file name and the files to concatenate. To start with an empty output file, we delete it when it already exists. Then, we loop over all the input files, successively reading an input file and write it to the output in the append mode. We do all these operations in the synchronous mode, because we don't want the content of the files to be mingled.

There's more...

In `concat2.dart`, which you can find within the `concat_files` folder, we see an asynchronous version that also works here—only the code in the `concat` method has to change. Have a look at the following code:

```
IOSink snk;

Future concat(List<String> files) {
  snk = output.openWrite(mode: FileMode.APPEND);
  return Future.forEach(files, (file) {
    Stream<List<int>> stream = new File(file).openRead();
    return stream.transform(UTF8.decoder)
      .transform(const LineSplitter())
      .listen((line) {
        snk.write(line + "\n");
      }).asFuture().catchError((_) => _handleError(file));
  });
}

_handleError(String file) {
  FileSystemEntity.isDirectory(file).then((isDir) {
    if (isDir) {
      print('error: $file is a directory');
    } else {
      print('error: $file not found');
    }
  });
}
```

We write to an `IOSink` object `snk` using the `openWrite` method in the append mode. The `Future.forEach` method asynchronously runs the callback provided on each file. The `forEach` method runs the callback for each element in order, moving to the next element only when the `Future` returned by the callback completes. The stream is transformed; transformers are used here to convert the data to UTF-8 and split string values into individual lines.

See also

- ▶ Refer to the *Parsing command-line arguments* recipe, *Chapter 2, Structuring, Testing, and Deploying an Application*, for more information on using the `args` package, and the *Transforming streams* recipe in this chapter.

Downloading a file

This recipe shows you the simplest ways to download a file through code, first in a command-line application and then from a web application. As an example, we download the front page of the Learning Dart website from `http://learningdart.org`.

Getting ready

A client program (be it web or command-line) receives content, such as files or web pages, from a web server using the HTTP protocol. The `dart:html` and `dart:io` package provides us with the basic classes we need to do this, which are as follows:

- ▶ The `Uri` class (from `dart:core`) has all we need to parse, encode, and decode web addresses; the method `Uri.parse` is often used
- ▶ The `HttpRequest` class (from `dart:html`) has the `getString` method to fetch a file from a URL
- ▶ The `HttpClient` class (from `dart:io`) has all kinds of methods, such as `get` and `post`, to send a request (class `HttpClientRequest`) to a web server and get a response (class `HttpClientResponse`) back

How to do it...

1. For a web app, this is shown in `download_string.dart`, which is started from `download_string.html` (these files can be found in the `download_file` project) as shown in the following code:

```
import 'dart:html';

main() {
  HttpRequest.getString('http://learningdart.org')
    .then(processString)
    .catchError(print);
}

processString(str) {
  print(str);
}
```

2. For a command-line app in the program `download_file.dart`, we see the basic mechanism of how to do this for a command-line app as follows:

```
import 'dart:io';
import 'dart:convert';

var client;

main() {
  var url = Uri.parse('http://learningdart.org');
  client = new HttpClient();
  client.getUrl(url)
    .then((HttpClientRequest req) => req.close())
    .then((HttpClientResponse resp) => writeToFile(resp));
}

writeToFile(resp) {
  resp.transform(UTF8.decoder)
    .toList().then((data) {
      var body = data.join('');
      var file = new File('dart.txt');
      file.writeAsString(body).then((_) {
        client.close();
      });
    });
}
```

How it works...

For a web app, we use the `getString` method on `HttpRequest` to fetch the file from the URL as one big string, which is asynchronously passed to `processString`. It could do just about anything with the string it gets back, for example, if it were a JSON or XML string, we could parse this and get data out of it to show on our web page. So `HttpRequest` can be used to fetch data over HTTP and FTP protocols from a URL, without producing complete web page updates. This is, in fact, the way to make AJAX calls (or XMLHttpRequest) and as a consequence, partial page updates. We will use it in the following recipe to fetch a large blob file.

[ Don't confuse this class with another class with the same name `HttpRequest` from `dart:io`, which must be used in server-side applications (we will use it extensively in the recipes of the following chapter). A web server, or more formally, an HTTP server, that listens for HTTP requests coming in on a specific host and port, generates such an object for each request it receives.]

For a command-line app, first we transform the web address string to a `Uri` object with the static method `parse`. Then we make an `HttpClient` object and invoke the `getUrl` request on URL (the `Uri` object). This works in two steps, each returning a `Future`, which are:

- ▶ The first `.then` completes with a `Request` object that has been made but not sent yet. In the callback, you can still change or add to the request headers or the body. A call to `close` sends the request to the server. This step serves to make the request and send it.
- ▶ The second `.then` completes when the `Response` object is received from the server, and you can access headers and the body (the body is available as a stream). This step serves to process the response; here, we call `writeToFile`.



If there is a body, it must be processed. Avoid memory leaks by calling the method `drain()`.



In `writeToFile`, we read the response data, transforming it from UTF-8 to a string (helped by the `join` method that transforms the list into a string), and write it to a file with `writeAsString`. When this finishes, the `HttpClient` object is closed; this releases the network connections that have been made.

There's more...

The following are some variations to accomplish the same thing for a command-line app:

Using pipe

The `.then` variable in the command line can be simplified to the following:

```
.then((HttpClientResponse resp) => resp.pipe(new  
File('dart.txt').openWrite()));
```

The `pipe` method on the response object can send the stream immediately to the file to be written. This will perform better when downloading bigger files.

Using the http package

An even more simplified approach can be taken using the `http` package by the Dart team, which was made to facilitate coding requests and responses (see `download_file2.dart`). Have a look at the following code:

```
import 'dart:io';  
import 'package:http/http.dart' as http;  
  
main() {  
  var url = Uri.parse('http://learningdart.org');
```

```
http.get(url).then((response) {  
  new File('dart.txt').writeAsBytes(response.bodyBytes);  
});  
}
```

See also

- ▶ See the *Writing to a file* recipe for information on the `writeAsBytes` method

Working with blobs

In the previous recipe, in step 1, we used a client `HttpRequest` object and its method `getString`. In this recipe, we want to download a blob (`binarylargeobject`) file, for example, a large image, audio, or video file. But first, you need to prepare for this if you need to do more than just download a string from a URL resource to process it on the client. You need to go through the following steps (for the code, see `request_prep.dart` in the project `request_blob`).

Getting ready

1. Create an `HttpRequest` object as shown in the following code:

```
import 'dart:html';  
  
void main() {  
  var path = 'http://learningdart.org';  
  var request = new HttpRequest();
```

2. Open it (here, with the HTTP GET method) as shown in the following code:

```
request  
..open('GET', path)
```

3. In this stage, we could also have configured its header with the `setRequestHeader()` method, for example, `request.setRequestHeader('Content-type', 'application/json')` when you are sending a JSON string.
4. Define a callback function, such as `requestComplete`, to execute when the response comes back; this is done in the `onLoadEnd` event as shown in the following code:

```
..onLoadEnd.listen((e) => requestComplete(request))
```

5. Use the `send` method to make the request as shown in the following code:

```
..send();  
}
```

Here, we send an empty string, because it is a GET request, but with a POST request, we could send data such as `request.send(JSON.encode(data))`.

In the callback, we test the status of the request if it is 200; if everything is OK, we can process `responseText`. Have a look at the following code:

```
requestComplete(HTTPRequest request) {
    if (request.status == 200) {
        print('headers: ${request.responseHeaders}');
        print('type: ${request.responseType}');
        print('text: ${request.responseText}');
    }
    else {
        print('Request failed, status=${request.status}');
    }
}
```

How to do it...

Now, we show you how to do the same thing for a blob in `request_blob.dart` (run it from `request_blob.html`) using the following code:

```
import 'dart:html';

1. To make a FileReader object, use the following code:
FileReader flr = new FileReader();
ImageElement img;

void main() {
    img = document.querySelector('#anImage');
    // var path = 'stadium.jpg';
    var path =
        'https://farm1.staticflickr.com/2/
        1418878_1e92283336_m.jpg';

2. To build the request for a blob, use the following code:
var request = new HttpRequest();
request
    ..open('GET', path)
    ..responseType = 'blob'
    ..overrideMimeType("image/jpg")
    ..onLoadEnd.listen((e) => requestComplete(request))
    ..send('');
}
```

3. To handle the response, use the following code:

```
requestComplete(HttpRequest request) {  
  if (request.status == 200 && request.readyState == HttpRequest.  
  DONE) {  
    Blob blob = request.response;  
    flr.onLoadEnd.listen( (e) {  
      img.src = flr.result;  
    });  
    flr.readAsDataUrl(blob);  
  }  
  else {  
    print('Request failed, status=${request.status}');  
  }  
}
```

Try it out with the 5 MB file stadium.jpg.

How it works...

To read the blob, we will need a `FileReader` object; we constructed this in step 1. In step 2, we build the request object, which is fairly general, except that we set `responseType` and `mimeType`. In the callback function in step 3, we test with `request.readyState == HttpRequest.DONE` to be sure that the request has been fully handled. Then, we performed the following steps:

- ▶ We created a `Blob` object from the `Blob` class in `dart:html` and set it to the response
- ▶ We read the blob with the method `readAsDataUrl`
- ▶ When this was completed (signaled by the `onLoadEnd` event), the source of the image tag was set to the result of the `FileReader` object

Transforming streams

Listening to a stream captures the sequence of results coming from an event-like action, such as clicking on a button in a web page or opening a file with the `openRead` method. These results are data that can be processed, but the errors that occur are also part of the stream. Dart can work with streams in a very functional way, such as filtering the results with `where` or mapping the results to a new stream (for a complete list of these methods, refer to <https://api.dartlang.org/apidocs/channels/stable/dartdoc-viewer/dart:async.Stream>). To modify the incoming results, we can also use a transformer; this recipe shows you how to do this (refer to the project `transforming_stream`).

How to do it...

In our script, we have a list, `persons`, where the items are themselves lists consisting of a name and a gender. We want to walk through the list and emit a greeting message based on the gender of the person, but if the gender is unknown, we skip that person. The following code shows us how we can do this with a transformer:

```
import 'dart:async';

var persons = [
    ['Carter', 'F'],
    ['Gates', 'M'],
    ['Nuryev', 'M'],
    ['Liszt', 'U'],
    ['Besançon', 'F']
];

void main() {
```

We need to perform the following steps to transform the streams:

1. To make a stream from the list, use the following code:

```
var stream = new Stream.fromIterable(persons);
```

2. To define a stream transformer, use the following code:

```
var transformer = new
    StreamTransformer.fromHandlers(handleData: convert);
```

3. To filter and transform the stream, and listen to its output to process further, use the following code:

```
stream
    .where((value) => value[1] != 'U')
    .transform(transformer)
    .listen((value) => print("$value"));
}

convert(value, sink) {
    // create new value from the original value
    var greeting = "Hello Mr. or Mrs. ${value[0]}";
    if (value[1] == 'F') {
        greeting = "Hello Mrs. ${value[0]}";
    }
}
```

```
        else if (value[1] == 'M') {
            greeting = "Hello Mr. ${value[0]}";
        }
        sink.add(greeting);
    }
```

After performing the preceding steps, we get the following output:

Hello Mrs. Carter

Hello Mr. Gates

Hello Mr. Nuryev

Hello Mrs. Besançon

How it works...

To turn a list into a stream, we used the `fromIterable` method as in step 1. Discarding some values from the stream can be done with `where`; see the first clause in step 3.

Step 2 details how to transform a stream. This method takes an object (here called `transformer`) of the class `StreamTransformer`, which allows you to change the contents of the stream. The constructor named `fromHandlers` takes an optional `handleData` argument that calls our callback function `convert` for each value passed from the stream. The `convert` option builds a new value based on the content of the old value and adds it in place of the old value of the `sink` variable. Only those transformed values are output on the stream, passed on to `listen`, and processed there. The `sink` option is an instance of the abstract class `StreamSink`, which is a generic destination of data and can be implemented by any data receiver.

There's more...

We have already used `transform` in this chapter when reading a file with `openRead`, as shown in the following code:

```
Stream<List<int>> input = file.openRead();
input
    .transform(UTF8.decoder)
    .transform(const LineSplitter())
```

The `inputStream` stream is a `List<int>` list, and thus strongly typed. First, the incoming integers are transformed into a stream of UTF-8 characters, and then the input is split into subsequent lines. Instead of transform, we could have used the `map` method on the stream as well.

`HttpRequest` in the browser does not support getting the response as a stream. To work along that pattern, you have to use `WebSockets` (refer to *Chapter 7, Working with Web Servers*).

See also

- ▶ Refer to the *Reading a file* recipe, the second example in the *Concatenating files* recipe, and the *Writing files* recipe for more examples on transforming streams

7

Working with Web Servers

In this chapter, we will cover the following recipes:

- ▶ Creating a web server
- ▶ Posting JSON-formatted data
- ▶ Receiving data on the web server
- ▶ Serving files with `http_server`
- ▶ Using sockets
- ▶ Using WebSockets
- ▶ Using secure sockets and servers
- ▶ Using a JSON web service

Introduction

Dart, besides being an excellent web programming language is, also suitable for writing server applications. In this chapter, we will specifically look at Dart's `dart:io` library to write web servers and their functionality. This library is built to work asynchronously so that the server can handle many requests at the same time (concurrently). It provides the class `HttpRequest` to write command-line clients. The Dart team also wrote the `http_server` package available from pub package manager. This package needs `dart:io` and provides some higher-level classes to make it easier to write clients and servers.

Creating a web server

The class `HttpServer` is used to write web servers; this server listens on (or binds to) a particular host and port for incoming HTTP requests. It provides event handlers (better called `request handlers` in this case) that are triggered when a request with incoming data from a web client is received.

How to do it...

We make a project called `simple_webserver` starting from the template command-line application and import `dart:io` as follows:

```
import 'dart:io';
//Define host and port:
InternetAddress HOST = InternetAddress.LOCOPBACK_IP_V6;
const int PORT = 8080;

main() {
    // Starting the web server:
    HttpServer.bind(HOST, PORT)
        .then((server) {
            print('server starts listening on port
${server.port}');
            // Starting the request handler:
            server.listen(handleRequest);
        })
        .catchError(print);
}

handleRequest(HttpRequest req) {
    print('request coming in');
    req.response
        ..headers.contentType = new ContentType("text", "plain",
            charset: "utf-8")
        ..write(' I heard you loud and clear.')
        ..write(' Send me the data!')
        ..close();
}
```

How it works...

A web server runs on a host (either specified by a name or an IP address) and uses a port on that host to listen for requests. We define these here upfront as stated in comment 1. HOST could be a string, such as localhost, or an object of the class `InternetAddress`. The LOOPBACK schema is the same as localhost; this is used for testing on a local machine. For production purposes, use `ANY_IP_V6` to allow for incoming connections from the network. Instead of `IP_V6`, you could also use `IP_V4`, but `IP_V6` is more general and includes an `IP_V4` listener.

A port can be any valid number above 1024 that is not in use. If another program is already listening on the same port (or the server is still running), an error occurs.

Next, we use the static method `bind` to create the web server; this returns a `Future` object to run asynchronously. When the bind is successful, the callback `then()` is called with the new `HttpServer` object as a parameter. We print out the port to the console, so we can confirm it is running. The `catchError` function will be triggered in the case of an exception and equally prints to the console.



Always provide error handling in the code of a server!

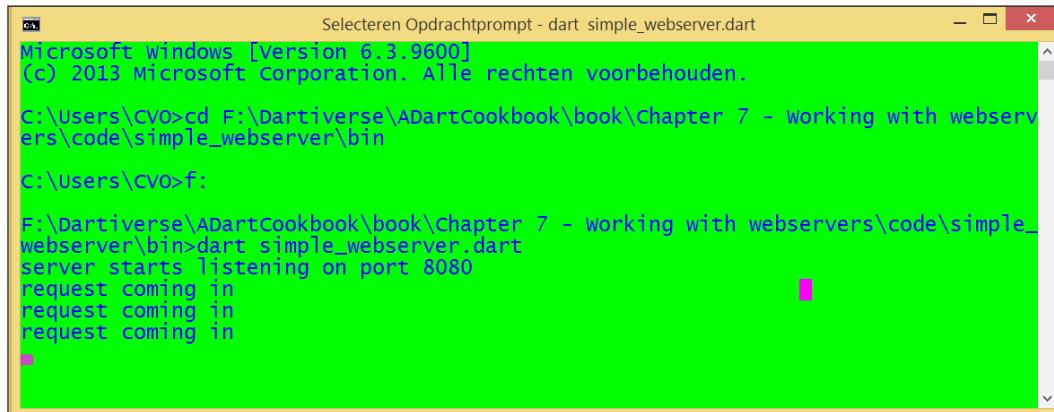
Next, the callback handler `handleRequest` is set up; it will be triggered for any incoming request that it accepts as a parameter. In other words when a request comes in, the server creates an `HttpRequest` object and passes it to the callback `handleRequest` of `listen()`.

In this first example, we write to its `response` object after first setting the content type in the headers and close it when we're done. The `response` object is of the class `HttpResponse`; it will contain the server's answer to the request.

To see it in action, start the server from the editor or on the command line with the following command:

```
dart simple_webserver.dart
```

This produces the following server console output:



```
Microsoft Windows [version 6.3.9600]
(c) 2013 Microsoft Corporation. Alle rechten voorbehouden.

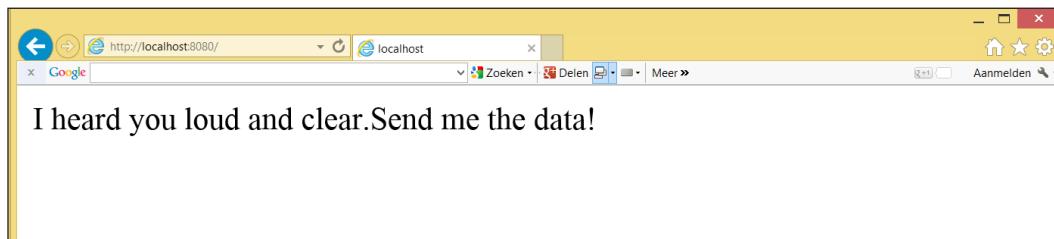
c:\users\cvo>cd F:\Dartiverse\ADartCookbook\book\Chapter 7 - working with web servers\code\simple_webserver\bin

c:\users\cvo>F:

F:\Dartiverse\ADartCookbook\book\Chapter 7 - working with web servers\code\simple_webserver\bin>dart simple_webserver.dart
server starts listening on port 8080
request coming in
request coming in
request coming in
```

Console output from the web server

Then, start any browser with the URL `http://localhost:8080` to see the response text appear on the client as shown in the following screenshot:



The browser client shows the response

There is more...

The `HttpRequest` object also has properties that provide information about the client's request; the most important ones are as follows:

- ▶ `method`: This is derived from the way the web form was submitted. In the `<form action="http://localhost:4041" method="GET">` code, the values it can take are GET, POST, PUT, or DELETE.
- ▶ `headers`: This gives general information on the request, such as content type, content length, and date.
- ▶ `uri`: This gives the location where the request originated from.

Posting JSON-formatted data

This is a recipe for a client web app that sends a request to a web server. The request contains the form's data that is posted in the JSON format.

How to do it...

Look at the project `post_form` for the code.

1. Our form (refer the next diagram) will post data for a job in IT; we reuse the class `Job` in the *Making toJSON and fromJSON methods in your class* recipe from *Chapter 4, Object Orientation*. We keep the example short and simple, but add two new properties, `posted` and `open`. Have a look at the following code:

```
class Job {  
    String type;  
    int salary;  
    String company;  
    DateTime posted; // date of publication of job  
    bool open = true; // is job still vacant ?  
    Job(this.type, this.salary, this.company, this.posted);  
    // toJSON and fromJSON methods  
}
```

2. The `model` class is made available to the code in `post_form.dart` using the following code:

```
import '../model/job.dart';
```

3. We add our own event handler for the submit button using the following code:

```
void main() {  
    querySelector("#submit").onClick.listen(submitForm);  
}
```

4. The method `submitForm` makes and sends the request as follows:

```
submitForm(e) {  
    e.preventDefault(); // Don't do the default submit.  
    // send data to web server:  
    req = new HttpRequest();  
    req.onReadyStateChange.listen(onResponse);  
    // POST the data to the server.  
    var url = 'http://127.0.0.1:PORT';  
    req.open('POST', url);  
    req.send(_jobData()); // send JSON String to server  
}
```

5. The `_jobData` function prepares the data to send as follows:

```
_jobData() {
    // read out data:
    InputElement ictmp, isal, iposted, iopen;
    SelectElement icode;
    icode = querySelector("#comp");
    icode = querySelector("#type");
    isal = querySelector("#sal");
    iposted = querySelector("#posted");
    iopen = querySelector("#open");
    var comp = icode.value;
    var type = icode.value;
    var sal = isal.value.trim();
    var posted = DateTime.parse(iposted.value.trim());
    var open = iopen.value;
    // make Job object
    Job jb = new Job(type, int.parse(sal), comp, posted);
    // JSON encode object:
    return jb.toJson();
}
```

6. The `onResponse` function gets the response from the server and shows it on the screen as shown in the following code:

```
void onResponse(_) {
    if (req.readyState == HttpRequest.DONE) {
        if (req.status == 200) {
            serverResponse = 'Server: ' + req.responseText;
        }
    } else if (req.status == 0) {
        // Status is 0: most likely the server isn't running.
        serverResponse = 'No server';
    }
    querySelector("#resp").text = serverResponse;
}
```

The following screenshot shows how our screen looks after sending the data:

A screenshot of a web browser window titled "Post form". The URL in the address bar is "127.0.0.1:3030/post_form/web/post_form.html". The page content is titled "Posting job data". It contains the following form fields:

- Company: Google
- Choose a job type: Web developer
- Salary: 5600
- Date posted: 11/06/2014
- Job is vacant:

Below the form is a message box with the text "No server". At the bottom left are "Submit" and "Clear" buttons.

The client sends job data

In the previous screenshot, no server is shown because there is no web server to process the request.

How it works...

In step 1, we added a `DateTime` property. Such a type is not natively serializable to JSON; the `encode` method does not know how to handle this case. We have to define this ourselves and provide a `toEncodable` closure as the second optional argument of `JSON.encode`; this returns an appropriate serialization of `DateTime`. The following code is the revised `toJson` method in the class `Job`:

```
String toJson() {
    var jsm = new Map<String, Object>();
    jsm["type"] = type;
    jsm["salary"] = salary;
    jsm["company"] = company;
    jsm["posted"] = JSON.encode(posted, toEncodable: (p){
        if(p is DateTime)
            return p.toIso8601String();
        return p;
    });
    jsm["open"] = open;
    var jss = JSON.encode(jsm);
    return jss;
}
```

The important part happens in step 4, where the `HttpRequest` object is sent; `req.open` posts the data to the URL of the server (here we test it locally with the localhost address `127.0.0.1`). We also define a callback function `onResponse` for the `onReadyStateChange` event that signals when a server response comes in.

The `send()` function happens asynchronously, so it returns as soon as the request is sent; `req.send` takes the data to be sent as the argument, and this is the JSON string prepared in the function `_jobData` in step 5. This reads out the data values from the screen and makes a `Job` object with them, and JSON formats that object with `toJson`.

Finally in step 6, when the request is complete and the server responds with the `OK` status `200`, which means success, the text response from the server is shown; otherwise it shows **No server**. The state in which the communication with the server is carried, is given by the `readyState` field. The ready state can have five possible values: `unsent`, `opened`, `headers received`, `loading`, and `done`. When the ready state changes, `HttpRequest` fires an event named `onReadyStateChange` and the `onResponse` callback function gets called.

See also

- ▶ See the *Working with blobs* recipe in *Chapter 6, Working with Files and Streams*, to learn how to make a request to download a blob file

Receiving data on the web server

In the previous recipe, we made a client app that sends its data to a web server in JSON format. In this recipe, we will make the web server that receives this data step by step, possibly process it, and then send it back to the client. You can find the code in the script `server\webserver.dart` in the project `post_form`.

How to do it...

Perform the following steps to make this work:

1. The following is the code that starts the web server:

```
import 'dart:io';

const HOST = '127.0.0.1';
const PORT = 4040;

void main() {
  HttpServer.bind(HOST, PORT).then(acceptRequests,
    onError: handleError);
}
```

2. The `acceptRequests` function describes how the web server handles incoming requests based on their method as follows:

```
void acceptRequests(server) {
    server.listen((HttpRequest req) {
        switch (req.method) {
            case 'POST':
                handlePost(req);
                break;
            case 'GET':
                handleGet(req);
                break;
            case 'OPTIONS':
                handleOptions(req);
                break;
            default: defaultHandler(req);
        }
    },
    onError: handleError, // Listen failed.
    onDone: () => print('Web server shuts down.')),
    print('Listening for GET and POST on http://$HOST:$PORT');
}
```

3. The different request handlers are shown in the following code:

```
void handlePost(HttpRequest req) {
    HttpResponse res = req.response;
    addCorsHeaders(res);
    res.statusCode = HttpStatus.OK;
    req.listen(processData, onError: handleError),
}

processData(List<int> buffer) {
    res.write('OK, I received: ');
    res.write(new String.fromCharCodes(buffer));
    // process incoming data
    res.close();
}

handleGet(HttpRequest req) { // not needed here }

void handleOptions(HttpRequest req) { // not needed here }

void addCorsHeaders(HttpResponse res) {
    res.headers.add('Access-Control-Allow-Origin', '*');
```

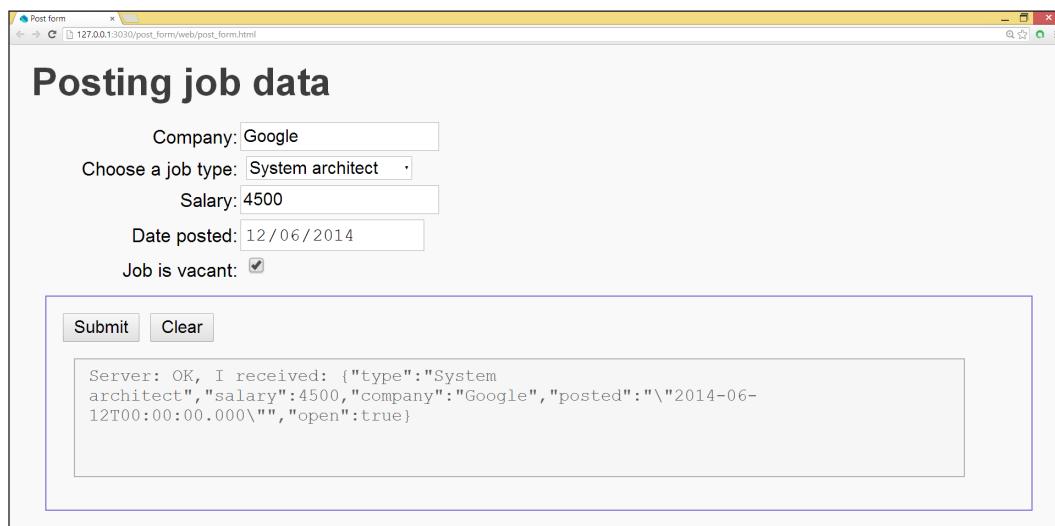
Working with Web Servers

```
res.headers.add('Access-Control-Allow-Methods', 'POST, OPTIONS');
res.headers.add('Access-Control-Allow-Headers',
'Origin, X-Requested-With, Content-Type, Accept');
}

void defaultHandler(HttpServletRequest req) {
    res = req.response;
    res.statusCode = HttpStatus.METHOD_NOT_ALLOWED;
    res.write("Unsupported request: ${req.method}.");
    res.close();
}

handleError(e) {
    print(e);
    // other error handling
}
```

Run the client from the previous recipe (start web\post_form.html) and post a job in JSON format to the server. The web server responds with an acknowledgement and returns the data back to the client. The client shows the following response:



How it works...

In step 1, we used an alternative way (compared to the *Making a web server* recipe) to start up the server; we give two callback functions for the `Future` object returned by `bind`:

- ▶ The first parameter is the `acceptRequests` function, which receives an `HttpServer` object as a parameter and then listens for incoming requests
- ▶ The second parameter is the optional `onError` argument with the callback function `handleError`; this is invoked when the binding fails, for example, when the port is in use

Another, more elegant way of writing this is shown in the following code:

```
HttpServer.bind(HOST, PORT)
  .then(acceptRequests)
  .catchError(handleError);
```

Step 2 gives us the processing of requests. For every incoming request, the server creates an `HttpRequest` object and passes it to the callback of `listen()`. So, the `HttpServer` object produces a stream of `HttpRequest` objects to be processed. Here, we see how you can use a switch/case to act differently on different kinds of requests (other request method), using the same exception-catching mechanism as in step 1. A second optional `onDone` parameter is a function that is called when the server is shut down.

In step 3, we built different request handlers. We always set the status code of the response, such as `res.statusCode = HttpStatus.OK;`; there are a lot of predefined values. See the docs for the class `HttpStatus`. In particular, you can use `HttpStatus.NOT_FOUND` in an error handler to signal a 404 File not Found HTTP error.

One thing to notice here is that we let the server send CORS headers to the client. This allows the client to send POST requests in the event that this web server is different from the one serving the original web application. Then, the client must first send an OPTIONS request, but for this, we don't have to write client code; it is handled automatically by the `HttpRequest` object. For a POST request, the code in `handlePost` listens for the client's data in `req`. `listen`. When all of the data is received, this is passed as a `List<int>` buffer to the callback function `processData`. In our case, this makes a string from the data and writes it back to the response. The response is a data stream that the server can use to send data back to the client. Other methods of writing to this stream are `writeln()`, `writeAll()`, and `writeCharCodes()`.

At this point in the code, the real server processing of the data, such as writing to a file (for example code see the *There's more...* section) or saving in a database, will be done. Closing the response sends it to the client.

There's more...

If the server has to set the content type for the response, do this as follows before the first write to the response in `handlePost`:

```
res.headers.contentType =  
    new ContentType("application", "json", charset: 'utf-8');
```

Here, we make it clear that we send JSON data using the UTF-8 character set.

Writing data to a file on the server

If we wanted to write the data received from the client to a file, we could do this as follows:

- ▶ Add the following line to `handlePost` before `req.listen`:

```
BytesBuilder builder = new BytesBuilder();
```
- ▶ In the following code, we see `processData`:

```
processData(List<int> buffer) {  
    builder.add(buffer);  
}
```
- ▶ The `builder` option collects the buffered data in chunks through the `add` method until all the data is delivered. Then, the `onDone` method in `acceptRequests` is called, such as `onDone writeToFile`). In the following code, we see `writeToFile`:

```
writeToFile(builder) {  
    var strJson = UTF8.decode(builder.takeBytes());  
    var filename = "jobs.json";  
    new File(filename).writeAsString(strJson, mode:  
        FileMode.APPEND).then((_) {  
        res.write('Job data was appended to file');  
        res.close();  
    });  
}
```

See also

- ▶ See the *Using CORS headers* recipe in *Chapter 5, Handling Web Applications*, for more information on CORS

Serving files with http_server

One of the main functions of a web server that we take for granted is the serving of static files. We can write this functionality completely with the classes from `dart:io`, but the Dart team has written a pub package called `http_server` with the aim to simplify web server programming to provide web content. We will use `http_server` in this recipe to code a web server that serves files. You can find the code in the project `serving_files`.

How to do it...

Perform the following steps to construct a web server for server files:

1. In the first example, `serving_file.dart`, you see the code for a web server delivering a file called `Learning Dart Packt Publishing.html`:

```
import 'dart:io';
import 'package:http_server/http_server.dart';

InternetAddress HOST = InternetAddress.LOOPBACK_IP_V6;
const PORT = 8080;

void main() {
    VirtualDirectory staticFiles = new VirtualDirectory('.');
    HttpServer.bind(HOST, PORT).then((server) {
        server.listen((req) {
            staticFiles.serveFile(new File('Learning Dart Packt
Publishing.html'), req);
        });
    });
}
```

Start the server by running `bin\serving_file.dart` and open a browser with the URL `localhost:8080`. You will see the **Learning Dart** web page.

2. To serve all files from the current directory, and all its subfolders, expand the code as shown in the following code (to see `serving_curdir.dart`):

```
import 'dart:io';
import 'dart:async';
import 'package:http_server/http_server.dart';

InternetAddress HOST = InternetAddress.LOOPBACK_IP_V6;
const PORT = 8080;
VirtualDirectory staticFiles;

void main() {
```

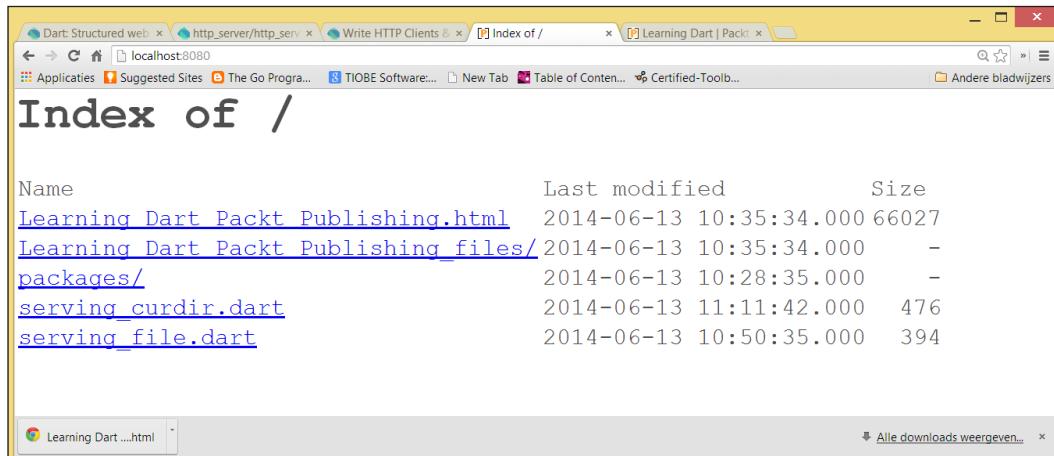
```
staticFiles = new VirtualDirectory('.')
..allowDirectoryListing = true;

runZoned( startServer, onError: handleError);
}

startServer() {
  HttpServer.bind(HOST, PORT).then((server) {
    server.listen(staticFiles.serveRequest);
  });
}

handleError(e, stackTrace) {
  print('An error occurred: $e $stackTrace');
}
```

Start the server by running `bin\serving_curdir.dart`, and open a browser with the URL `localhost:8080`. You will see a list of what's inside the bin folder as shown in the following screenshot:



Directory listing from a web server

How it works...

In step 1, we first add the package `http_server` to our project (adding it to `pubspec.yaml` and importing it in the code). Then, we define the folder from which serving will take place by making an object `staticFiles` from the class `VirtualDirectory`. This class handles all the details of serving files, specifying here that it will serve from the current directory. On this object, we call the method `serveFile` with a reference to the file as an argument.

In step 2, the `serveRequest` method handles a single request for any file in the current directory (the property `allowDirectoryListing` makes the content viewable). Notice that the server request handler is enveloped in a call to the method `runZoned` from `dart:async`. This is not needed to make it work, but it illustrates a way to make our code more robust. When using `runZoned`, the function given as its first argument is executed as if in a sandbox, and the second optional argument `onError` handles all uncaught exceptions, synchronous or asynchronous. You can find more details about the use of zones at <https://www.dartlang.org/articles/zones/>.

There's more...

- ▶ If you need the path to the current executing script, use the following code:

```
var path = Platform.script.toFilePath();
```

- ▶ If you need the path to the web folder, use the following code:

```
final HTTP_ROOT_PATH =
    Platform.script.resolve('web').toFilePath();
```

- ▶ You can point a virtual directory to that path in order to start serving files from that folder using the following code:

```
final virDir = new VirtualDirectory(HTTP_ROOT_PATH);
```

Together, the classes `HttpServer`, `VirtualDirectory`, and `Platform` are sufficient to implement a basic web server. When different responses are needed according to the URL, the class `Router` (see the *Using WebSockets* recipe) can simplify the code.



Use the `pub` package `path` and its cross-platform methods, such as `join` and `split` if you need to manipulate the path to certain folders or files. In particular, `toUri()` and `fromUri()` are useful when converting between a URI and a path.

Using sockets

At a somewhat lower level in the OSI model than HTTP clients and servers, we find sockets. They also enable interprocess communications across a network between clients and servers and are implemented on top of the TCP/IP. The classes that offer that functionality can be again found in `dart:io` as follows:

- ▶ `Socket`: This is used by a client to establish a connection to a server
- ▶ `ServerSocket`: This is used by a server to accept client connections

How to do it...

The following steps will show you how to make a server socket work:

1. The following is the code for the server (see the project `sockets, socket_server.dart`):

```
import 'dart:io';
import 'dart:convert';

InternetAddress HOST = InternetAddress LOOPBACK_IP_V6;
const PORT = 7654;

void main() {
    ServerSocket.bind(HOST, PORT)
        .then((ServerSocket srv) {
            print('serversocket is ready');
            srv.listen(handleClient);
        })
        .catchError(print);
}

void handleClient(Socket client) {
    print('Connection from: '
        '${client.remoteAddress.address}:${client.remotePort}');
    // data from client:
    client.transform(UTF8.decoder).listen(print);
    // data to client:
    client.write("Hello from Simple Socket Server!\n");
    client.close();
}
```

Start the server by running `bin\socket_server.dart`.

2. The following is the code for a client (see the project `sockets, socket_client.dart`):

```
import 'dart:io';

InternetAddress HOST = InternetAddress LOOPBACK_IP_V6;
const PORT = 7654;

void main() {
    Socket.connect("google.com", 80).then((socket) {
        print('Connected to: ')
```

```
'${socket.remoteAddress.address}:${socket.remotePort}');
socket.destroy();
});
// prints: Connected to: 173.194.65.101:80

Socket.connect(HOST, PORT).then((socket) {
print(socket.runtimeType);
// data to server:
socket.write('Hello, World from a client!');
// data from server:
socket.listen(onData);
});
}

onData(List<int> data) {
print(new String.fromCharCodes(data));
}
```

3. Start one (or more) client(s) by running bin\socket_client.dart.

The following is the output from the server console:

serversocket is ready

Connection from: ::200:0:8017:7b01%211558873:6564

Hello, World from a client!

The following is the output from a client console:

_Socket

Hello from Simple Socket Server!

Connected to: 74.125.136.139:80

We can see that there is two-way communication.

How it works...

In step 1, a server to handle client socket connections is created by binding `ServerSocket` to a specific TCP port that it will listen on. This `bind` method returns a `Future <ServerSocket>`. Again, we will use the `Future.then` method to register our callback so that we know when the socket has been bound to the port. Then, the server starts listening and calls `handleClient` for each incoming connection. This callback prints the remote address from the client, prints the data the client has sent, writes a message to the client, and then closes the connection.

In step 2, the client first opens a connection to `www.google.com` on port 80 (the port that serves web pages). After the socket is connected to the server, the IP address and port are printed to the screen. Then, the socket is shut down using `socket.destroy`. In the second part of this step, we connect to the local socket server, write a message to it with `socket.write`, and start listening to the server with `socket.listen`. We transform the data that comes in as a list of integers into a string that is printed out.

There's more...

Socket communication is often blocked by firewalls; if this is an issue, take a look at WebSockets in the following recipe.

Dart also supports UDP socket programming; the article by James Locum offers a detailed discussion at <http://jamesslocum.com/post/77759061182>.

Using WebSockets

This recipe will show you how to use WebSockets in a client-server application (both web and command-line clients) and what its advantages are. You can find the code in the project `websockets`.

Getting ready

HTTP is a simple request-response-based protocol, and then the connection is broken from the application's point of view until the next request. In a modern web application (for example, in online multiplayer games), the client and server are of equal importance; changes in the state of the application can take place on both sides. So, we need a bi-directional communication channel between the client(s) and the server that allows for two-way real time updates and more interaction; this is exactly what WebSockets has to offer. WebSocket connections between a browser and a server are made through a handshake request. This is a regular HTTP client request with an upgrade flag in the header, also containing a `Sec-WebSocket-Key`, which is a random value that has been base64 encoded. To complete the handshake, the server responds with a `Sec-WebSocket-Accept` response.

WebSockets are implemented in all modern browsers (Internet Explorer v10 and above), and can also be used in non-web applications. The communication takes place over TCP port 80, so it can pass unhindered through firewalls.

How to do it...

Perform the following steps to make a WebSockets server work:

1. Use the following code to set up a WebSockets server (see `websocket_server.dart`):

```
import 'dart:io';
import 'dart:async';

InternetAddress HOST = InternetAddress.ANY_IP_V6;
const PORT = 8080;

main() {
    runZoned(startWSServer, onError: handleError);
}

startWSServer() {
    HttpServer.bind(HOST, PORT)
        .then((server) {
            print('Http server started on $HOST $PORT');
            server.listen(handleRequest);
        });
}

handleError(e, stackTrace) {
    print('An error occurred: $e $stackTrace');
}

handleRequest(HttpRequest req) {
    if ( (req.uri.path == '/ws') // command-line client
        || WebSocketTransformer.isUpgradeRequest(req) // web-client
    ) {
        // Upgrade a HttpRequest to a WebSocket connection.
        WebSocketTransformer.upgrade(req).then(handleWebSocket);
    }
    else {
        print("Regular ${req.method} request for: ${req.uri.path}");
        serveNonWSRequest(req);
    }
}

handleWebSocket(WebSocket socket) {
    print('Client connected!');
    socket.listen((String msg) {
        print('Message received: $msg');
    });
}
```

```
        socket.add('echo from server: $msg') ;
    },
    onError: (err) {
    print('Bad WebSocket request $err') ;
},
onDone: () {
    print('Client disconnected') ;
}),
}
serveNonWSRequest(req) {
    var resp = req.response;
    resp.statusCode = HttpStatus.FORBIDDEN;
    resp.reasonPhrase = "WebSocket connections only";
    resp.response.close();
}
```

2. The following code is used for a command-line WebSocket client (`websocket_client.dart`):

```
import 'dart:io';

const HOST = 'localhost';
const PORT = 8080;

main() {
    var wsurl = "ws://$HOST:$PORT/ws";
    WebSocket.connect(wsurl)
    //Open the websocket and attach the callbacks
    .then((socket) {
        socket.add('from client: Hello Websockets Server!');
        socket.listen(onMessage, onDone: connectionClosed);
    })
    .catchError(print);
}

void onMessage(String msg) {
    print(msg);
}

void connectionClosed() {
    print('Connection to server closed');
}
```

If we run the server script, `websocket_server.dart`, and then start a client `websocket_client.dart`, we get the following output on the server console:

Http server started on InternetAddress('::1', IP_V6)8080 connected!	Client
	Message received: from client: Hello Websockets Server!

The client console prints the following output:

echo from server: Hello Websockets Server!

To make a web client, we need a web page `websocket_webclient.html` that invokes a script `websocket_webclient.dart`. The web page is kept very simple with an input field that will collect a string to send to the server and a `<div>` element that shows the response from the server as follows:

```
<h1>WebSocket Sample</h1>
<input id="input" type="text"></input>
<div id="output"></div>
<script type="application/dart"
src="websocket_webclient.dart"></script>
```

The following is the script code:

```
import 'dart:html';

void main() {
    TextInputElement inp = querySelector('#input');
   DivElement out = querySelector('#output');

    String srvuri = 'ws://localhost:8080/ws';
    WebSocket ws = new WebSocket(srvuri);

    inp.onChange.listen((Event e){
        ws.send(inp.value.trim());
        inp.value = "";
    });

    ws.onOpen.listen((Event e) {
        outputMessage(out, 'Connected to server');
    });

    ws.onMessage.listen((MessageEvent e){
        outputMessage(out, e.data);
    });

    ws.onClose.listen((Event e) {
        outputMessage(out, 'Connection to server lost...');

    });
}

void outputMessageDivElement(String message) {
    out.text = message;
}
```

```
});  
  
ws.onError.first.then((_) {  
  print("Failed to connect to ${ws.url}. "  
    "Please rerun bin/websocket_server.dart and try again.");  
});  
}  
  
void outputMessage(Element e, String message) {  
  print(message);  
  e.appendText(message);  
  e.appendHtml('<br/>');  
  //Make sure we 'autoscroll' the new messages  
  e.scrollTop = e.scrollHeight;  
}
```

When the server runs, start the web client by opening `websocket_webclient.html`, and type in some text. The text received by the web server is echoed back and shown both on the page and in the editor console as follows:



A WebSocket web client

How it works...

In step 1, the server is run in a `runZoned()` clause to add additional exception handling capabilities (see the *Serving files with http_server recipe*). We start a web server as usual. In `handleRequest`, we check whether the path of the request ends in `/ws`. In that case, we have a command-line client issuing a WebSocket request. A web client making a WebSocket request will add an upgrade flag in the headers. For this, we can test it with the `isUpgradeRequest` method of the `WebSocketTransformer` class. If that is the case, we call the `upgrade` method on the same class, and when done, we call the `handleWebSocket` method. This starts listening for client connections, prints out any client message in the server console, and echoes this back to the client. If the message was a JSON string, we could have decoded it before it starts listening to client connections with `socket.map((string) => JSON.decode(string))`.

In the case of a normal HTTP request, `serveNonWSRequest` is used to block it, but of course, we could do normal web request handling as well.

The command-line client in step 2 uses the `WebSocket` class from `dart:io`. It connects to a WebSocket server with a `ws://` URL as a parameter to the `connect` method. Then, it can write to the socket with `add` and receive messages on the socket with `listen`.

The web client in step 3 uses the `WebSocket` class from `dart:html`. Calling its constructor with the URI of the server opens the web socket connection. The `send` method called on this instance sends the client data (here, the text of the input field) to the server. When the response from the server can be read from the socket, the `onMessage` event is fired and shows the response. Other useful events of the `WebSocket` instance are:

- ▶ `onOpen`: This is called when the connection is made
- ▶ `onClose`: This is called when the socket is no longer available (because the server was shut down or a network connection failure)
- ▶ `onError`: This is called when an error occurs during the client-server dialog

There's more...

The pub package `route` can be used to associate callbacks with URL patterns. In this recipe, instead of testing the `/ws` pattern, we could have used `Router` from the package `route` to do that for us. We import this package, and then `startWSServer` will contain the following code:

```
import 'package:route/server.dart' show Router;

startWSServer() {
  HttpServer.bind(HOST, PORT).then((server) {
    print('Http server started on $HOST $PORT');
    Router router = new Router(server);
  });
}
```

```
    router.serve('/ws').transform(new WebSocketTransformer()) .  
      listen(handleWebSocket);  
  } );  
}
```

As a more general example of how routing can be useful, consider the following example. Let's suppose our clients search for stock data with URLs ending with /stocks and /stocks/GOOG. Then, we can define pattern1 and pattern2 as instances of the class `UrlPattern` with a regular expression containing the following pattern:

```
// Pattern for all stocks(plural).  
final stocksUrl = new UrlPattern(r'/stocks\/?');  
// Pattern for a single stock('/stock/GOOG', for example).  
final stockUrl = new UrlPattern(r'/stock/(\d+)\/?');
```

Our router instance will then bind the callback functions `serveStocks` and `serveStock` to those patterns through the `serve` method:

```
var router = new Router(server)  
.serve(stocksUrl, method: 'GET').listen(serveStocks)  
.serve(stockUrl, method: 'GET').listen(serveStock)  
// all other possible patterns and method combinations  
.defaultStream.listen(serveNotFound);
```

As shown in the first example, patterns can also be simple strings like `/stockdata`.

See also

- ▶ Look at the [Serving files with http_server recipe](#) for more information on `runZoned`
- ▶ The Dart website has a very good tutorial on a search app implemented with WebSockets and Dartiverse Search, at <https://www.dartlang.org/docs/dart-up-and-running/contents/ch05.html>

Using secure sockets and servers

In this recipe, we describe the steps to make your web server encrypt its communication with clients using **Secure Sockets Layer (SSL)** on the HTTPS protocol.

Getting ready

Dart uses SSL/TSL security; it relies on X.509 certificates to validate servers and (optionally) clients. The server provides a certificate that will verify itself as a trusted server to the client. When the client accepts the certificate, symmetric session keys will be exchanged and used to encrypt the communications between the server and the client. So, in order for your server to provide a secured connection, it has to have a security certificate installed, provided by a **Certificate Authority (CA)**.

Dart uses a **Network Security Services (NSS)** database to store the server's private key and certificate. For our example, we will use the test certificate database in the subfolder pkcert, which is also provided as an illustration in the tutorial at <https://www.dartlang.org/docs/tutorials/httpserver/>.

You can set up an NSS key database yourself to create certificates for test purposes. James Locum provides a detailed description on how to do this at <http://jamesslocum.com/post/70003236123>.

How to do it...

The program `secure_server.dart` shows us the code needed to start a secure server; perform the following steps to use secure sockets and service:

1. Import the `dart:io` package as follows:

```
import 'dart:io';

InternetAddress HOST = InternetAddress LOOPBACK_IP_V6;
const int PORT = 8080;

main() {
```

2. Read the certificate using the following code:

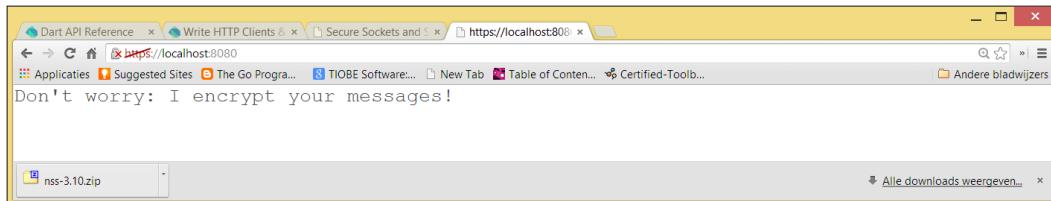
```
var testcertDb =
    Platform.script.resolve('pkcert').toFilePath();
SecureSocket.initialize(database: testcertDb, password:
    'dartdart');
```

3. Start the HTTP server with the certificate using the following code:

```
HttpServer.bindSecure(HOST, PORT, certificateName:
    'localhost_cert').then((server) {
    print('Secure Server listening');
    server.listen((HttpRequest req) {
        print('Request for ${req.uri.path}');
        var resp = req.response;
        resp.write("Don't worry: I encrypt your messages!");
```

```
        resp.close();
    });
}
}
```

4. If we now use the URL `https://localhost:8080` in a browser, we first get a screen warning us that this connection is not trusted (because it is only a test certificate). If we continue, we see the response of the server in the browser's screen as shown in the following screenshot:



A secure socket connection

The following is the code for a secure command-line client (`secure_client.dart`):

```
import 'dart:io';

InternetAddress HOST = InternetAddress.LOCOPBACK_IP_V6;
const int PORT = 4777;
SecureSocket socket;

void main() {
    SecureSocket.connect(HOST, PORT, onBadCertificate:
        (X509Certificate c) {
        print("Certificate WARNING: ${c.issuer}:${c.subject}");
        return true;
    }).then(handleSecureSocket);
}

handleSecureSocket(SecureSocket ss) {
    // send to server:
    ss.write("From client: can you encrypt me server?");
    // read from server:
    ss.listen((List data) {
        String msg = new String.fromCharCodes(data).trim();
        print(msg);
    });
}
```

The client console gives the following output:

Certificate WARNING: CN=myauthority:CN=localhost

How it works...

In step 1, we read the certificate. The first line with `Platform.script` finds the path to the folder `pkcert`, where the certificate database is located. Then we call the `initialize` method on the class `SecureSocket`, providing the certificate. In step 2, the secure server is started by binding to a host and port and providing the name of the certificate. Step 3 shows us a browser connecting to the secure server.

In step 4, we see how a command-line client can connect to the secure server by calling `SecureSocket.connect()`. This needs an `onBadCertificate` callback, which must return a Boolean value that indicates whether to accept or reject a bad certificate. The test certificate will trigger this callback, so we need to return `true` in order to use this certificate. With respect to the `write` and `listen` methods of `SecureSocket`, let's write to and read from the secure server.

See also

- ▶ Refer to the *Getting information from the operating system* recipe in Chapter 1, *Working with Dart Tools*, for more details about the `Platform` class
- ▶ For more information about certificates and creating them, refer to <https://help.ubuntu.com/12.04/serverguide/certificates-and-security.html>

Using a JSON web service

In this recipe, we make a browser app ask data from a web service (Yahoo stock data) in JSON format, decode that data, and dynamically build up the web page showing the data.

Getting ready

This is what the URL we will use will look like: `http://query.yahooapis.com/v1/public/yql?q=SELECT`.

To get the data, we use the **Yahoo Query Language (YQL)**, `q=` indicating the start of the query represented by `SELECT`. Suppose we want to look up stock data for Yahoo, Google, Apple, and Microsoft, the selected query will be of the following form:

```
select * from yahoo.finance.quotes where symbol  
in(YHOO,AAPL,GOOG,CMSFT)  
&env=http://datatables.org/Falltables.env&format=json
```

How to do it...

Look at the code in `stockviewer_dart`:

```
import 'dart:html';
import 'dart:convert';

main() {
    LoadData();
}
```

1. Call the web server asynchronously using the following code:

```
void LoadData() {
    var stock = "GOOG";
    var request =
        "http://query.yahooapis.com/v1/public/
        yql?q=select%20*%20from%20yahoo.finance.quotes%20"
        "where%20symbol%20in%20(%22$stock%22)%0A%09%09"
        "&env=http%3A%2F%2Fdatatables.org%2Falltables.
        env&format=json";
    var result =
        HttpRequest.getString(request).then(OnDataLoaded);
}
```

2. Web service responses callback as shown in the following code:

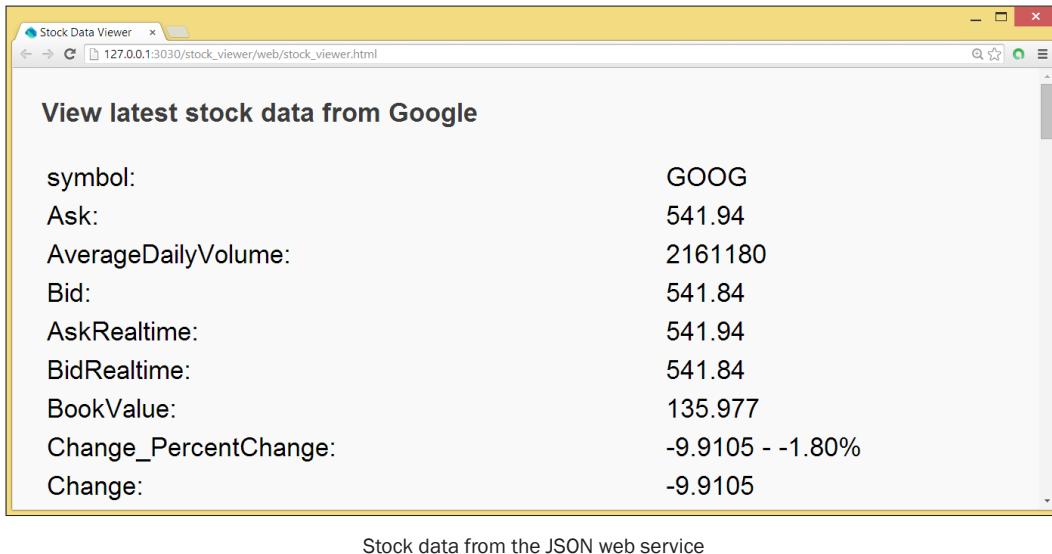
```
void OnDataLoaded(String response) {
    String json = response.substring(response.indexOf("symbol")
        - 2, response.length - 3);
    Map data = JSON.decode(json);
    var table = CreateTable();
    var props = data.keys;
    props.forEach((prop) => ProcessStockEntry(prop, data,
        table));
    document.body.nodes.add(table);
}
```

3. Create the HTML table with the data as shown in the following code:

```
TableElement CreateTable() {
    TableElement table = new TableElement();
    var tBody = table.createTBody();
    return table;
}

void ProcessStockEntry(String prop, Map data, TableElement
table) {
    String value = data["$prop"];
    var row = table.insertRow(-1); // Add new row to our table
    var propCell = row.insertCell(0); // Add new cell for
        property
    String prophtml = '$prop:';
    propCell.setInnerHTML(prophtml);
    var valueCell = row.insertCell(1); // Add new cell for the
        value
    String valuehtml = '$value';
    valueCell.setInnerHTML(valuehtml);
}
```

The browser shows the stock data as shown in the following screenshot:



How it works...

In step 1, the request string is URI encoded, and the stock symbol we want is inserted. Then, the web server is called with the `getString` method from `HttpRequest`. Step 2 shows the code that analyzes the response when this has arrived. We extracted the map with the stock data (starting with the symbol) and decoded that JSON string into the map data. We then created an HTML table, and for each of the properties in the stock data (`Ask`, `AverageDailyVolume`, `Bid`, and so on), we inserted a table row with the key and the data in step 3.

See also

- ▶ See the *Downloading a file* recipe for more information on the `getString` method

8

Working with Futures, Tasks, and Isolates

In this chapter, we will cover the following recipes:

- ▶ Writing a game loop
- ▶ Error handling with Futures
- ▶ Scheduling tasks using Futures
- ▶ Running a recurring function
- ▶ Using isolates in the Dart VM
- ▶ Using isolates in web apps
- ▶ Using multiple cores with isolates
- ▶ Using the Worker Task framework

Introduction

The `Future` class from `dart:async` lies in the basis of all asynchronous programming in Dart. A Future is, in fact, a computation that is deferred; it represents an object whose value will be available sometime in the future. It is not available immediately, because the function that returns its value depends on some kind of input/output and is, thus, unpredictable by nature. Here are some examples: a time-consuming computation, reading in a big dataset, and searching through a number of websites.

In the two previous chapters, quite a lot of recipes used Futures; in *Chapter 6, Working with Files and Streams*, we had the following recipes using Futures:

- ▶ *Reading and processing a file line by line*
- ▶ *Concatenating files the asynchronous way*
- ▶ *Downloading a file*

In the preceding chapter, we used Futures in the following recipes:

- ▶ *Making a web server*
- ▶ *Receiving data on the web server*
- ▶ *Using sockets*

In this chapter, we will concentrate on how to write elegant code for Futures and combine their possibilities with the execution of tasks and isolates to enhance the performance of our apps.

Dart runs single-threaded, so it uses, by default, only one CPU on a multi-core machine; if you want concurrency, you must use isolates. In the second part of the chapter, you will find recipes featuring isolates, Dart's mechanism to provide concurrency and parallelism in applications.

Writing a game loop

In game apps, the refresh rate of the screen is vital; it must be high enough to ensure agreeable and realistic gameplay. Refreshing means periodically redrawing the screen. This recipe shows you how to build that in your app. It is illustrated in the code of `gameloop`, a simple version of the well-known memory game that uses the `boarding` package by Dzenan Ridzanovic. The goal is to click quickly enough to get identical pairs of the colored squares. Start it by running `game.html` (don't use `pub serve` for the launch, select **Run** and then **Manage Launches**, and in **Pub Settings**, uncheck **use pub serve to serve the application**).

How to do it...

1. The game starts off in `main()` of `game.dart` (only the relevant parts of the code are shown here):

```
import 'dart:async';
import 'dart:html';
// ... other code
part 'model/memory.dart';
part 'view/board.dart';

main() {
    new Board(new Memory(4), querySelector('#canvas')).draw();
}
```

2. In the constructor of the Board class, the game loop is started with the call to `window.animationFrame`:

```
class Board extends Surface {  
    // code left out  
  
    Board(Memory memory, CanvasElement canvas) :  
        this.memory = memory,  
        super(memory, canvas) {  
        // code left out  
        querySelector('#canvas').onMouseDown.listen((MouseEvent e) {  
            // code left out  
            if (memory.recalled) { // game over  
                new Timer(const Duration(milliseconds: 5000), ()  
                    =>memory.hide());  
            }  
            else if (cell.twin.hidden) {  
                new Timer(const Duration(milliseconds: 800), ()  
                    =>cell.hidden = true);  
            }  
        });  
  
        window.animationFrame.then(gameLoop);  
    }  
}
```

3. And here is the `gameLoop` method itself:

```
void gameLoop(num delta) {  
    draw();  
    window.animationFrame.then(gameLoop);  
}  
  
void draw() {  
    super.draw();  
    if (memory.recalled) { // game over  
        // code left out  
    }  
}
```

How it works...

In step 1, the game is started by instantiating an object of the `Board` class (from `view/board.dart`) and calling the `draw` method on it. In step 2, the most important statement is the last one in the constructor, `window.animationFrame.then(gameLoop)`.

This method of the `Window` class in `dart:html` returns a `Future` that completes just before the window is about to repaint, so this is the right time to redraw our screen.



Use `animationFrame` to do this, because the animation then uses a consistent frame rate, and thus, looks smoother, and it also works at the screen's refresh rate. Don't use `Future` or `Timer` to draw frames; use `Timer` only if you have to code for a browser that does not support `animationFrame`!

This is done in the callback method `gameloop` in step 3; first `draw()` is executed, then `window.animationFrame`, and then `(gameLoop)`. So, this recursively calls `gameloop` again and again. This way, we are sure that the animation will continue.

There's more...

We also see how the class `Timer` from `dart:async` is used. For example, in the end-of-game condition (`memory` is recalled), the colored pattern is hidden from sight after 5 seconds by the following `Timer` object:

```
new Timer(const Duration(milliseconds: 5000), () => memory.hide());
```

After this duration of time, the anonymous callback function, `() => memory.hide()`, is executed. Use the named constructor, `Timer.periodic`, with the same arguments to execute a callback function periodically.

See also

- ▶ Refer to the *Running a recurring function* recipe in this chapter to find out more about the `Timer` class

Error handling with Futures

This recipe shows you how to handle errors comprehensively when working with Futures. The accompanying code `future_errors.dart` (inside the `bin` map in the `future_errors` project) illustrates the different possibilities; however, this is not a real project, so it is not meant to be run as is.

Getting ready

When the function that returns a Future value completes successfully (calls back) signaled in the code by then, a callback function handleValue is executed that receives the value returned. If an error condition took place, the callback handleError handles it. Let's say this function is getFuture(), with Future as the result and a return value of type T, then this becomes equivalent to the following code:

```
Future<T> future = getFuture();
future.then(handleValue)
    .catchError(handleError);

handleValue(val) {
    // processing the value
}

handleError(err) {
    // handling the error
}
```

The highlighted code is sometimes also written as follows, only to make the return values explicit:

```
future.then( (val) =>handleValue(val) )
    .catchError( (err) =>handleError(err) );
```

When there is no return value, this can also be written as shown in the following code:

```
future.then( () =>nextStep() )
```

When the return value doesn't matter in the code, this can be written with an _ in place of that value, as shown in the following code:

```
future.then( (_) =>nextStep(_) )
```

But, in any case, we prefer to write succinct code, as follows:

```
future.then(nextStep)
```

The then and catcherror objects are chained as they are called, but that doesn't mean that they are both executed. Only one executes completely; compare it to the try-catch block in synchronous code. The catcherror object can even catch an error thrown in handleValue.

This is quite an elegant mechanism, but what do we do when the code gets a little more complicated?

How to do it...

Let's see the different ways you can work with Futures in action:

- ▶ **Chaining Futures:** Let's suppose we have a number of steps that each will return a Future and so run asynchronously, but the steps have to execute in a certain order. We could chain these as shown in the following code:

```
firstStep()
    .then((_) =>secondStep())
    .then((_) =>thirdStep())
    .then((_) =>fourthStep())
    .catchError(handleError);
```

- ▶ **Concurrent Futures:** If, on the other hand, all the steps return a value of the type T, and their order of execution is not important, you can use the static method of Future, wait, as shown in the following code:

```
List futs = [firstStep(), secondStep(), thirdStep(),
fourthStep()];
Future.wait(futs)
.then((_) =>processValues(_))
.catchError(handleError);
```

- ▶ Run the script wait_error.dart to see what happens when an error occurs in one of the steps (either by throw or a Future.error call):

```
import 'dart:async';

main() {
  Future<int> a = new Future(() {
    print('a');
    return 1;
  });
  Future<int> b = new Future.error('Error occurred in b!');
  Future<int> c = new Future(() {
    print('c');
    return 3;
  });
  Future<int> d = new Future(() {
    print('d');
    return 4;
  });

  Future.wait([a, b, c, d]).then((List<int> values) =>
  print(values)).catchError(print);
```

```
    print('happy end') ;
}
```

The output is as follows:

happy end

a

c

d

Error occurred in b!

- ▶ The following code helps to catch a specific error or more than one error:

```
firstStep()
  .then((_) =>secondStep())
    // more .then( steps )
  .catchError(handleArgumentError,
  test: (e) => e is ArgumentError)
  .catchError(handleFormatException,
  test: (e) => e is FormatException)
  .catchError(handleRangeError,
  test: (e) => e is RangeError)
  .catchError(handleException, test: (e) => e is Exception);
```

- ▶ Often, you want to execute a method that does a cleanup after asynchronous processing no matter whether this processing succeeds or ends in an error. In that case, use `whenComplete`:

```
firstStep()
  .then((_) =>secondStep())
  .catchError(handleError)
  .whenComplete(cleanup);
```

With respect to handling synchronous and asynchronous errors, let's suppose that we want to call a function `mixedFunction`, with a synchronous call to `synFunc` that could throw an exception and an asynchronous call to `asynFunc` that could do likewise, as shown in the following code:

```
mixedFunction(data) {
  var var2 = new Var2();
  var var1 = synFunc(data);           // Could throw error.
  return var2.asynFunc().then(processResult); // Could throw error.
}
```

If we call this function `mixedFunction(data).catchError(handleError);`, then `catchError` cannot catch an error thrown by `synFunc`. To solve this, they call a `Future.sync` function as shown in the following code:

```
mixedFunction(data) {  
    return new Future.sync(() {  
        var var1 = synFunc(data);           // Could throw error.  
        return var1.asynFunc().then(processResult); // Could throw error.  
    });  
}
```

That way, `catchError` can catch both synchronous and asynchronous errors.

How it works...

In variation 1, `catchError` will handle all errors that occur in any of the executed steps. For variation 2, we make a list with all the steps. The `Future.wait` option will do exactly as its name says: it will wait until all of the steps are completed. But they are executed in no particular order, so they can run concurrently. All of the functions are triggered without first waiting for any particular function to complete. When they are all done, their return values are collected in a list (here called `val`) and can be processed. Again, `catchError` handles any possible error that occurs in any of the steps.

In the case of an error, the `List` value is not returned; we see that, in the example on `wait_error`, **happy end** is first printed, then `a`, `c`, and `d` complete, and then the error from `b` is caught; if `d` also throws an error, only the `b` error is caught. The `catchError` function doesn't know in which step the error occurred unless that is explicitly conveyed in the error.

In the same way as in the `catch` block, we can also test in `catchError` when a specific exception occurs using its second optional `test` argument, where you test the type of the exception. This is shown in variation 3; be sure then, to test for a general exception as the last clause. This scenario will certainly be useful if a number of different exceptions can occur and we want a specific treatment for each of them.

Analogous to the optional `finally` clause in a `try` statement, asynchronous processing can have a `whenComplete` handler as in variation 4, which always executes whether there is an error or not. Use it to clean up and close files, databases, and network connections, and so on.

Finally, in variation 5, the normal `catchError` function won't work, because it can only handle exceptions arising from asynchronous code execution. Use `Future.syncHere`, which is able to return the result or error from both synchronous and asynchronous method calls.

Scheduling tasks using Futures

The Dart VM is single-threaded, so all of an app's code runs in one thread, also called the main isolate. This is because `main()` is the function where Dart code starts executing an isolate, because Dart's concurrency model is based on isolates as separate processes that exchange messages. We will talk about isolates in depth in the coming recipes, but if your code doesn't start a new isolate, all of it runs in one isolate. But, in this one isolate, you can have lots of asynchronous pieces of code (let's call them tasks) running at the same time; in what order do they execute, and can we influence that order? It turns out that a better understanding of Dart's event loop and task queuing mechanism enables us to do that. This recipe will clarify Dart's scheduling mechanism and give you hints and tips for an ordered execution of tasks.

How to do it...

Have a look at the program `tasks_scheduling.dart` (the tasks are numbered consecutively and according to the way they are started):

```
import'dart:async';

main() {
  print('1) main task #1');
  scheduleMicrotask(() => print('2) microtask #1'));
  newFuture.delayed(new Duration(seconds:1),
    () =>print('3) future #1 (delayed)');
  new Future(() => print('4) future #2'));
  print('5) main task #2');
  scheduleMicrotask(() => print('6) microtask #2'));
  new Future(() => print('7) future #3'))
    .then((_) => print('8) future #4'))
    .then((_) => print('9) future #5'))
    .whenComplete(cleanup);
  scheduleMicrotask(() => print('11) microtask #3'));
  print('12) main task #3');
}

cleanup() {
  print('10) whenComplete #6');
}
```

The following screenshot shows the output of the program, the order of which is explained in the next section:

```
1) main task #1
5) main task #2
12) main task #3
2) microtask #1
6) microtask #2
11) microtask #3
4) future #2
7) future #3
8) future #4
9) future #5
10) whenComplete #6
3) future #1 (delayed)
```

How it works...

The main isolate proceeds as follows:

1. First the code in `main()` is executed, line after line and synchronously.
2. Then the event-loop mechanism kicks in, and this looks at the two queues in the system in the following order:
 - ❑ First, it takes a look at the microtask queue this queue is for all tasks that need to be executed before the event queue, for example, the changes that have to be done internally before the DOM starts rendering the modified screen. All of the tasks in this queue are executed before going to the following step.
 - ❑ Then, the event queue is handled, here, the tasks of all outside events, such as mouse events, drawing events, I/O, timers, messages between Dart isolates, and so on, are scheduled. Of course, in each queue, the tasks are handled one by one in the *first in first out* order.



While the event-loop is handling the microtask queue, no work is done on the event-queue, so the app can't draw or react to user events and seems effectively blocked. So, keep the microtask queue as short as possible. Preferably, put your tasks on the event queue.

In principle, when both queues are empty, the app can exit. Tasks (these are pieces of code to run later, asynchronously) can be scheduled using the following classes and methods from `dart:async`:

1. Make a new `Future` object with a function to execute; this is appended to the event queue.
2. With `Future.delayed`, you can specify the execution of a function to occur after a certain duration; this also goes to the event queue.
3. Call the top-level `scheduleMicrotask()` function, which appends an item to the microtask queue.

 Don't use a `Timer` class to schedule a task; this class has no facilities to catch exceptions, so an error during a timer task will blow up your app.

Chaining `Futures` in a series of `then` statements effectively ensures that they are executed in that order (see step 1 of the previous recipe). Also, a `whencomplete` clause will execute immediately after all the previous `then` statements.

So this is the order of execution: `main()`, then à microtask queue, then event queue, and then delayed tasks.

 As a general best practice, don't put a compute-intensive task on either queue, but create that task in a separate isolate (or worker for a web app).

See also

- ▶ See the upcoming recipes about isolates in this chapter, such as *Using isolates in web apps* and *Using multiple cores with isolates*

Running a recurring function

Let's suppose your code needs to run a certain function periodically at a certain interval. This recipe shows how you can do this very easily.

How to do it...

Look at the following code of `recurring_function.dart`:

```
import 'dart:async';
var count = 0;
```

```
const TIMEOUT = const Duration(seconds: 5);
const MS = const Duration(milliseconds: 1);

void main() {
    // 1. Running a function repeatedly:
    const PERIOD = const Duration(seconds:2);
    newTimer.periodic(PERIOD, repeatMe);
    // 3. Running a function once after some time:
    const AFTER70MS = const Duration(milliseconds:70);
    new Timer(AFTER70MS, () => print('this was quick!'));
    // 4. Running a function asap:
    Timer.run(() => print('I ran asap!'));
    // 5. Calculating a period and provoking a timeout:
    startTimeout(500);
}

repeatMe(Timer t) {
    print("I have a repetitive job, and I'm active is ${t.isActive}!");
    count++;
    // 2. Stop the repetition:
    if (count==4 &&t.isActive) {
        t.cancel();
        print("I'm active is ${t.isActive} now");
    }
}

startTimeout([intvariableMS ]) {
    var duration = variableMS == null ? TIMEOUT : MS * variableMS;
    return new Timer(duration, handleTimeout);
}

handleTimeout() {
    print('I was timed out!');
}
```

The following is the output from this code; if you need help figuring out the order, read the next section:

I ran asap!

this was quick!

I was timed out!

I have a repetitive job, and I'm active is true!

I have a repetitive job, and I'm active is true!

I have a repetitive job, and I'm active is true!

I have a repetitive job, and I'm active is true!

I'm active is false now

How it works...

The Timer class from `dart:async` gives us this functionality through the `periodic` named constructor as shown in comment 1. This takes two arguments, a `Duration` object and a function (here `repeatMe`) to run, which has the timer as the single parameter. This comes in handy to stop the repetition, which is shown in comment 2 with the `cancel()` method after 4 repetitions. The `isActive` property can be used to test whether the repetition is still going on. Comment 3 shows how to run a function only once after a time interval; just use the normal `Timer` constructor with the same arguments, but the callback doesn't have a `Timer` parameter. To run a function as soon as the event-loop mechanism permits, use the static `run` method as shown in comment 4. A negative or zero duration is equivalent to calling `run`. Comment 5 shows that a `Timer` class can also be useful to stop a running function or even the entire app.

There's more...

The durations or periods don't have to be constant from the start; they can be calculated before starting the `Timer`. Timers can also be used in web applications, but for drawing purposes, use `window.animationFrame`. When your app is compiled to JavaScript, the finest time granularity that the browser can give you is 4 milliseconds.

See also

- ▶ Refer to the *Writing a game-loop* recipe in this chapter for more information on `window.animationFrame`
- ▶ Refer to the *Exiting from an app* recipe in *Chapter 2, Structuring, Testing, and Deploying an Application*, for other alternatives to stop a running app

Using isolates in the Dart VM

Dart code runs in a single thread, in what is called a single process or isolate. In the standalone Dart VM, all code starts from `main()` and is executed in the so-called `root` isolate. To make an app more responsive or to increase its performance, you can assign parts of the code to other isolates. Moreover, because there are no shared resources between isolates, isolating third-party code increases the application's overall security. A Dart server app or command-line app can run part of its code concurrently by creating multiple isolates. If the app runs on a machine with more than one core or processor, it means these isolates can run truly in parallel. When the root isolate terminates the Dart VM exits and, with it, all isolates that are still running. Dart web apps currently can't create additional isolates, but they can create workers by adding instances of the `dart:html Worker` class, thereby adding JavaScript Web workers to the web app. In this recipe, we show you how to start up a new isolate from the main isolate and how to communicate with it using ports.

How to do it...

Examine the code in `using_spawn.dart` to create isolates with `spawn`, as shown in the following code:

```
import'dart:async';
import'dart:isolate';

main() {
  // 1- make a ReceivePort for the main isolate:
  varrecv = new ReceivePort();
  // 2- spawn a new isolate that runs the code from the echo
  // function
  // and pass it a sendPort to send messages to the main isolate
  Future<Isolate> remote = Isolate.spawn(echo, recv.sendPort);
  // 3- when the isolate is spawned (then), take the first message
  remote.then((_) =>recv.first).then((sendPort) {
    // 4- send a message to the isolate:
    sendReceive(sendPort, "Do you hear me?").then((msg) {
      // 5- listen and print the answer from the isolate
      print("MAIN: received $msg");
      // 6- send signal to end isolate:
      returnsendReceive(sendPort, "END");
    }).catchError((e) => print('Error in spawning isolate $e'));
  });
}

// the spawned isolate:
```

```
void echo(sender) {
    // 7- make a ReceivePort for the 2nd isolate:
    var port = new ReceivePort();
    // 8- send its sendPort to main isolate:
    sender.send(port.sendPort);
    // 9- listen to messages
    port.listen((msg) {
        var data = msg[0];
        print("ISOL: received $msg");
        SendPort replyTo = msg[1];
        replyTo.send('Yes I hear you: $msg, echoed from spawned isolate');
        // 10- received END signal, close the ReceivePort to save
        // resources:
        if (data == "END") {
            print('ISOL: my receivePort will be closed');
            port.close();
        }
    });
}
}

Future sendReceive(SendPort port, msg) {
    ReceivePort recv = new ReceivePort();
    port.send([msg, recv.sendPort]);
    returnrecv.first;
}
```

This script produces the following output:

ISOL: received [Do you hear me?,SendPort]

MAIN: received Yes I hear you: [Do you hear me?,SendPort], echoed from spawned isolate

ISOL: received [END, SendPort]

ISOL: my receivePort will be closed

From the output, we see that the main isolate receives its message echoed back from the second isolate. Examine the code in `using_spawnuri.dart` to create isolates with `spawnUri`:

```
import'dart:async';
import'dart:isolate';

main() {
    varrecv = new ReceivePort();
    Future<Isolate> remote =
```

```
Isolate.spawnUri(Uri.parse("echo.dart"),
  ["Do you hear me?"], recv.sendPort);
remote.then((_) =>recv.first).then((msg) {
  print("MAIN: received $msg");
});
}
```

The following is the code from echo.dart:

```
import'dart:isolate';

void main(List<String>args, SendPortreplyTo) {
  replyTo.send(args[0]);
}
```

The following is the output:

MAIN: received Do you hear me?

How it works...

Isolates are defined in their own library called `dart:isolate`. They conform to the well-known actor-model: they are like separate little applications that only communicate with each other by passing asynchronous messages back and forth; in no way can they share variables in memory. The messages get received in the order in which you send them. Each isolate has its own heap, which means that all values in memory, including global variables, are available only to that isolate. Sending messages, which comes down to serializing objects across isolates, has to obey certain restrictions. The messages can contain only the following things:

- ▶ Primitive values (`null`, `bool`, `num`, `double`, and `String`)
- ▶ Instances of `SendPort`
- ▶ Lists and maps whose elements are any of these

When isolates are created via `spawn`, they are running in the same process, and then it is also possible to send objects that are copied (currently only in the Dart VM).

An isolate has one `ReceivePort` to receive messages (containing data) on; it can listen to messages. Calling the `sendport` getter on this port returns `SendPort`. All messages sent through `SendPort` are delivered to the `ReceivePort` they were created from. On `SendPort`, you use the `send` method to send messages; a `ReceivePort` uses the `listen` method to capture these messages.

For each `ReceivePort` port there can be many `SendPort`. A `ReceivePort` is meant to live for as long as there is communication, don't create a new one for every message. Because Dart does not have cross-isolate garbage collection, `ReceivePort` is not automatically garbage-collected when nobody sends messages to them anymore.



Treat `ReceivePort` like resources, and close them when they aren't used anymore.



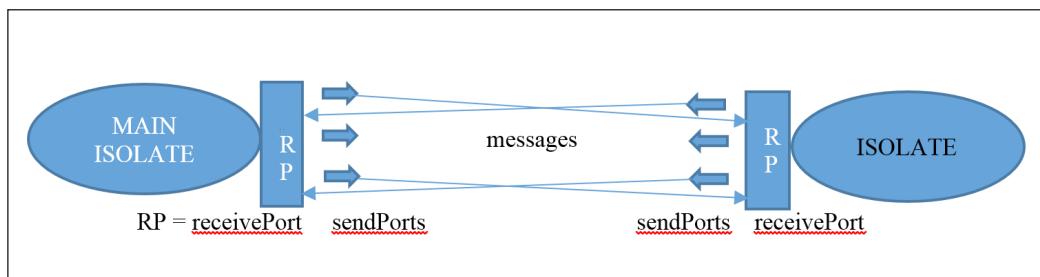
When working with isolates, a `ReceivePort` in the main or root isolate is obligatory.



Keeping the `ReceivePort` open will keep this main isolate alive; close it only when the program can stop.



Schematically, we could represent it as shown in the following diagram:



A new isolate is created in one of the following two ways:

- ▶ Through the static method `Isolate.spawn(fnIsolate, msg)`, the new isolate shares the same code from which it was spawned. This code must contain a top-level function or static one-argument method `fnIsolate`, which is the code the isolate will execute (in our example, the `echo` function); `msg` is a message. In step 2, `msg` is the `SendPort` of the main isolate; this is necessary because the spawned isolate will not know where to send its results.
- ▶ Through the static method `Isolate.spawnUri(uriOfCode, List<String>args, msg)`, the new isolate executes the code specified in the Uri `uriOfCode` (in our example, the script `echo.dart`), and passes it the argument list `args` and a message `msg` (again containing the `SendPort`).

Isolates start out by exchanging `SendPort` in order to be able to communicate. Both methods return a `Future` with `isolate`, or an error of type `IsolateSpawnException`, which must be caught. This can be done by chaining a `catchError` clause or using the optional `onError` argument of `spawn`. However, an error occurring in a spawned isolate cannot be caught by the main isolate, which is logical, because both are independent code sets being executed. You can see this for yourself by running `isolates_errors.dart`. Keep in mind the following restrictions when using `spawn` and `spawnUri`:

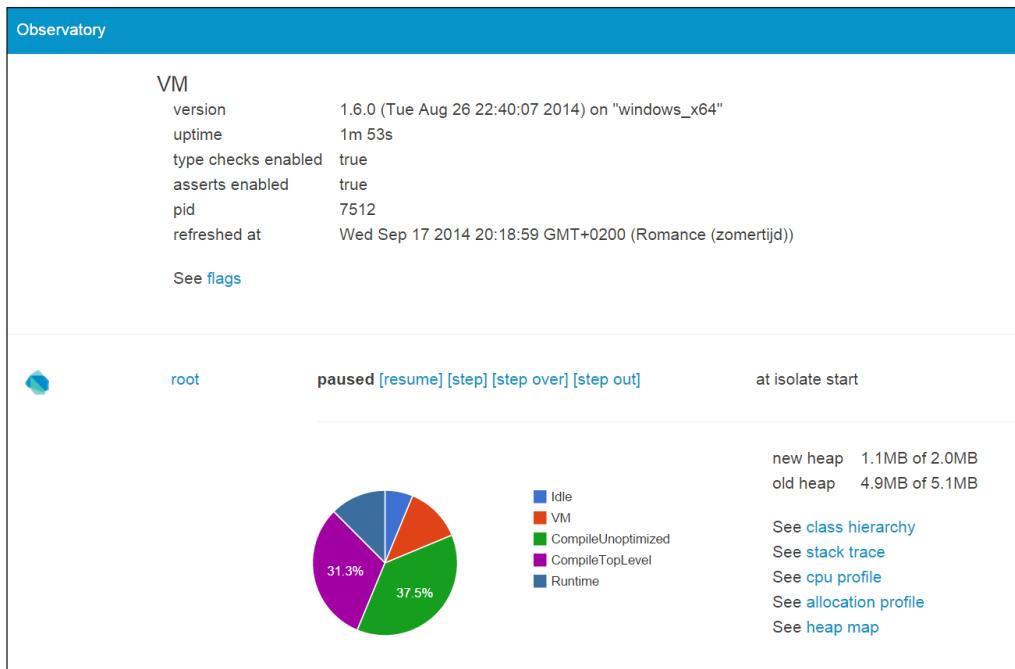
- ▶ `Spawn` works in server apps but doesn't work in Dart web apps. The browser's main isolate, the DOM isolate, does not allow this. This is meant to prevent concurrent access to the DOM.
- ▶ However, `spawnUri` does work in Dart web apps and server apps, and the isolate resulting from this invocation can itself spawn other isolates. The Dart VM translates these isolates into web workers (refer to <http://www.html5rocks.com/en/tutorials/workers/basics/>).

There's more...

So, if you have a compute-intensive task to run inside your app, then to keep your app responsive, you should put the task into its own isolate or worker. If you have many such tasks, then how many isolates should you deploy? In general, when the tasks are compute-intensive, you should use as many isolates as you expect to have cores or processors available. Additional isolates that are purely computational are redundant. But if some of them also perform asynchronous calls (such as I/O for example), then they won't use much processor time. In that case, having more isolates than processors makes sense; it all depends on the architecture of your app. In extreme cases, you could use a separate isolate for each piece of functionality or to ensure that data isn't shared.



Always benchmark your app to check whether the number of isolates are optimized for the job. You can do this as follows: in the Run Manage Launches tool, tick the choices in the VM settings, pause isolate on start, and pause isolate on exit. Then, open the observatory tool through the image button on the left-hand side of Dart Editor to the red square for termination, where you can find interesting information about allocations and performance.



As demonstrated in the second example, `spawnUri` provides a way to dynamically (that is, in run-time) load and execute code (perhaps even an entire library). Don't confuse Futures and isolates; they are different and are also applied differently.

- ▶ An isolate is used when you want some code to truly run in parallel, such as a mini program running separately from your main program. You send isolate messages, and you can receive messages from isolates. Each isolate has its own event-loop.
- ▶ A Future is used when you want to be notified when a value is available later in the event-loop. Just asking a Future to run a function doesn't make that function run in parallel. It just schedules the function onto the event-loop to be run at a later time.

At this moment, isolates are not very lightweight in the sense of Erlang processes, where each process only consumes a small amount of memory (of the order of Kb). Evolving isolates towards that ideal is a longer-term goal of the Dart team. Also, exception handling and debugging within isolates are a bit rough or difficult; expect this to change. It is also not specified how isolates map to operating system entities such as threads or processes; this can depend on the environment and platform. Isolates haven't been extended yet to inter-VM communication.



When two Dart VMs are running on the server, it is best to use TCP sockets for communication. You can start `ServerSocket` to listen for incoming requests and use `Socket` to connect to the other server.

See also

- ▶ Refer to the *Using isolates in web apps* recipe for another example using `spawnUri`. Find another example of isolates in the *Using multiple cores with isolates* recipe.
- ▶ Refer to the *Using Sockets* recipe for more information on `Sockets` and `ServerSockets` in the next chapter.
- ▶ Refer to the *Profiling and benchmarking your app* recipe in *Chapter 2, Structuring, Testing, and Deploying an Application*, for more information on benchmarking.

Using isolates in web apps

In this recipe, you will learn how to use isolates in a web application in the project `using_web_isolates`. This example runs in the Dart VM embedded in a browser as well as compiled to JavaScript. In the latter case, it uses HTML5 Web workers, which runs in the background independently of other scripts without affecting the performance of the page.

How to do it...

The main isolate in `using_web_isolates.dart` runs the following code:

```
import'dart:isolate';
import'dart:html';
import'dart:async';

main() {
  Element output = querySelector('output');
  SendPort sendPort;
  ReceivePort receivePort = new ReceivePort();
  receivePort.listen((msg) {
    if (sendPort == null) {
      sendPort = msg;
    } else {
```

```
        output.text += 'Received from isolate: $msg\n';
    }
});

String workerUri = 'worker.dart';
int counter = 0;
// start 3 isolates:
for (int i = 1; i <= 3; i++) {
    Isolate.spawnUri(Uri.parse(workerUri), [], receivePort.sendPort).
        then((isolate) {
        print('isolate spawned');
        newTimer.periodic(const Duration(seconds: 1), (t) {
            sendPort.send('From main app: ${counter++}');
            if (counter == 10) {
                sendPort.send('END');
                t.cancel();
            }
        });
    });
}
}

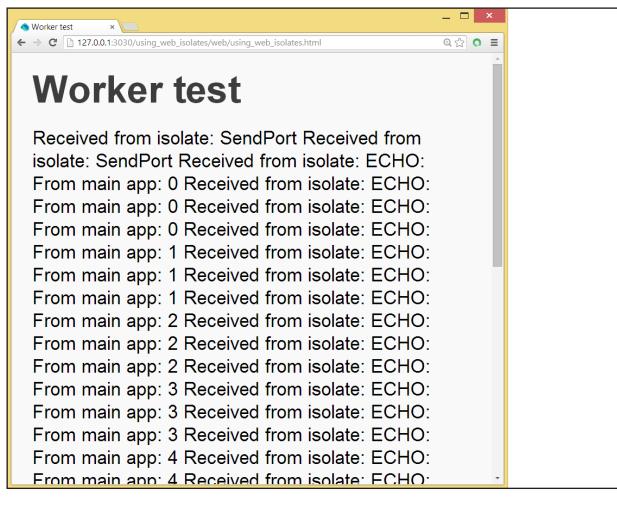
The following is the code for the isolate worker.dart:
```

```
import'dart:isolate';

main(List<String>args, SendPortsendPort) {
    ReceivePortreceivePort = new ReceivePort();
    sendPort.send(receivePort.sendPort);

    receivePort.listen((msg) {
        if (msg == 'END') receivePort.close;
        sendPort.send('ECHO: $msg');
    });
}
```

The following is the output shown on the web page:



Isolates in a web app

We see the messages coming in from the three isolates. At the count of 10, the timer stops, and the isolates receive a message to shut down.

How it works...

The main isolate first sets up a `ReceivePort` and a listener to the isolates. This listener first captures the isolate's `sendPort` and then appends each message on the web page. An isolate in a web app must be started with `spawnUri`; the code sits in the script `worker.dart`. Then the root isolate starts a timer and sends each second the number of seconds passed. After 10 seconds, the timer is canceled, and an `END` message is sent.

We start three isolates, each executing the same code. The main method in the isolates code receives `List<String>args` as its first argument. This comes from the second parameter of `spawnUri`; this is a way for the main isolate to send initial data to the isolate, but here it was the empty list. Each isolate first sets up its `ReceivePort`, and sends its `sendPort` to the main isolate, so communication lines are established. Then, it starts listening and echoes back what it receives. The isolate exits when the `END` message is received. When compiling to JavaScript with `dart2js`, `.js` is automatically added to the end of the script name `worker.dart`.

See also

- ▶ Refer to the *Using isolates in the Dart VM* recipe for a general explanation on isolates.

Using multiple cores with isolates

In this recipe, we show you that the Dart VM uses multiple cores on a processor without having to specify anything to the VM. This allows a much better performance and throughput than if a Dart app could only use one processor.

How to do it...

Look at the following code for `many_cores.dart` (in the project `using_isolates`):

```
import'dart:isolate';

main() {
    int counter = 0;
    ReceivePortreceivePort = new ReceivePort();
    receivePort.listen((msg) {
        if (msg is SendPort) {
            msg.send(counter++);
        } else {
            print(msg);
        }
    });
}

// starting isolates:
for (var i = 0; i < 5; i++) {
    Isolate.spawn(runInIsolate, receivePort.sendPort);
}
}

// code to run isolates
runInIsolate(SendPortsendPort) {
    ReceivePortreceivePort = new ReceivePort();
    // send own sendPort to main isolate:
    sendPort.send(receivePort.sendPort);

    receivePort.listen((msg) {
        var k = 0;
        var max = (5 - msg) * 500000000;
        for (var i = 0; i < max; ++i) {
            i = ++i - 1;
            k = i;
        }
    });
}
```

Working with Futures, Tasks, and Isolates

```
    sendPort.send("I received: $msg and calculated $k");
}
}
```

After some time, the calculated results are shown as follows:

I received: 4 and calculated 499999999

I received: 3 and calculated 999999999

I received: 2 and calculated 1499999999

I received: 1 and calculated 1999999999

I received: 0 and calculated 2499999999

The following is a screenshot of the CPU activity on an eight-core machine. It is clear that four cores are busy running the isolates corresponding to the four isolates that were spawned in the for-loop previously:



Multicore processing with isolates

How it works...

The first thing the main and other isolates do is make their `ReceivePort`. The isolates run the code in `runInIsolate` and get the port to send their results to. Have a look at the following command:

```
Isolate.spawn(runInIsolate, receivePort.sendPort);
```

The isolates send their own `sendPort` to `main`. The main isolate listens on its port; when it receives `sendPort`, it sends an integer counter to the isolate:

```
if (msg is SendPort) {  
    msg.send(counter++);  
    // other code  
}
```

The isolates listen until they receive their counter value, and then start their long-running calculation. Upon completion, the result is sent to the root isolate, where it is displayed. The program never stops because the `ReceivePort` in `main` is never closed.

Using the Worker Task framework

The `isolate` library only gives us the low-level, basic building blocks, so working with isolates in a realistic application environment can be challenging. Specifically for this purpose, the `worker` concurrent task executor framework was developed by Diego Rocha, available from pub package manager. It was made to abstract all the isolate managing and message passing and make concurrency in Dart as easy as possible. `Worker` also contains built-in error handling, so you needn't worry about that either. This recipe will show you how to use this framework so that you can concentrate on higher-level application details.

How to do it...

In the project `using_worker`, you can find a number of programs (`using_worker1` through `using_worker4`) illustrating the use of this framework. The script `using_worker.dart` illustrates the main steps, namely creating a task, creating a worker, give a task to the worker, and process the results:

```
import 'package:worker/worker.dart';  
  
Worker worker;  
  
void main() {  
    // 1- Make a Task object:  
    Task task = new HeavyTask();  
    // 2- Construct a Worker object:  
    worker = new Worker();  
    // specifying poolSize and spawning lazy isolates or not:  
    // worker = new Worker(poolSize: noIsol, spawnLazily: false);  
    // 3- Give a task to the worker  
    // 4- when the results return, process them  
    worker.handle(task).then(processResult);  
}
```

```
//5 - Task custom class must implement Task interface
class HeavyTask implements Task {

    execute() {
        return longRunningComputation();
    }

    boolean longRunningComputation() {
        var stopWatch = new Stopwatch();
        stopWatch.start();
        while (stopWatch.elapsedMilliseconds < 1000) {
            stopWatch.stop();
            return true;
        }
    }

    processResult(result) {
        print(result);
        // process result
        // 4- Close the worker object(s)
        worker.close();
    }
}
```

How it works...

First, add the package `worker` to `pubspec.yaml`, and import it in the code. A task is something that needs to be executed. This is an abstract class in the library, providing an interface for tasks and specifying that your custom Task class must implement the `execute` method, which returns a result. In our script the custom Task is `HeavyTask`, and `execute` simulates a long running computation using the `Stopwatch` class.

A `worker` object creates and manages a pool containing a number (`poolSize`) of isolates providing you with an easy way to perform blocking tasks concurrently. It spawns isolates lazily as Tasks are required to execute; the spawned isolates are available in a queue named `isolates`. The currently active isolates are stored in an iterable called `workingIsolates`, and the free isolates can be retrieved from an iterable called `availableIsolates`.

In the language of isolates, this is what happens; when a `Worker` instance is created, it starts listening to the `ReceivePort` of the current isolate, and a pool of isolates is created. The isolates in this pool are used to process any task passed to the worker. When a task is passed to the worker to be handled, it returns a Future. This Future will only complete when the Task is executed or when it fails to be executed.

By default, `worker` is created with `poolSize` equal to the number of processors on the machine (`Platform.numberOfProcessors`), and the isolates are spawned lazily (that is, only when needed). You can, however, change the number of isolates and also whether the isolates are spawned lazily or not using optional constructor parameters, as follows:

```
worker = new Worker(poolSize: noIsol, spawnLazily: false);
```

The work is started by handing over the task to the worker with `worker.handle(task)`. The `handle` method takes a `task`, and returns a `Future` object, so we process the result when it is returned with `worker.handle(task).then(processResult);`.

Also, make sure that, after the processing is done, `worker` gets closed or the program keeps running.

When executing a number of tasks, you can add them to a `List<Future>` task as shown in the following code:

```
int noTasks = 500;
for (var i=1; i<=noTasks; i++) {
  tasks.add(worker.handle(new HeavyTask()));
}
```

And then process them with:

```
Future.wait(tasks).then(processResult);
```

This mechanism is illustrated in the `using_worker2` and `using_worker3` examples.

There's more...

Another good scenario to use isolates or use `worker` class in particular is that of a web server that has a few different services. Perhaps one of those services has to do some calculations and takes a while to respond, whereas, the others are light and respond right away. When the heavy service is requested not using isolates, all other requests are blocked, waiting to be processed, even if they are requesting one of the light services. If you do use an isolate or `worker` and run the heavy service in parallel, it will take roughly the same time to respond to the first request, but all the subsequent requests won't have to wait.

See also

- ▶ See the *Using isolates in the Dart VM* recipe in this chapter for background information on isolates
- ▶ See the *Error handling with Futures* recipe in this chapter for more information on how to use Futures

9

Working with Databases

In this chapter, we will cover the following recipes:

- ▶ Storing data locally with IndexedDB
- ▶ Using Lawndart to write offline web apps
- ▶ Storing data in MySQL
- ▶ Storing data in PostgreSQL
- ▶ Storing data in Oracle
- ▶ Storing data in MongoDB
- ▶ Storing data in RethinkDB

Introduction

Data is like food for applications; without it, they would have no meaning. Furthermore, data must be persisted; storing data can be done on the client, server, or both. On the client side, we look at IndexedDB and the Lawndart data manager, which provides offline data storage without having to worry whether IndexedDB is supported or not. Then, we investigate how to store data on the server in SQL as well as NoSQL database systems. By the end of this chapter, you will have a whole spectrum of choices to select the database in which you will store your app's data.

Storing data locally with IndexedDB

IndexedDB is a more robust client-side storage mechanism than local storage in a browser. Likewise, it provides offline capabilities and is based on saving and retrieving data as key-value pairs, but it lets you store significantly bigger amounts of data and allows for high performance searching using database keys. IndexedDB is supported in modern browsers, but is only partially supported in Internet Explorer above Version 10 (refer to <http://caniuse.com/#feat=indexeddb> for details).

How to do it...

You can find the code for this recipe in the `using_indexeddb` project. We use the same method from the *Posting JSON-formatted data* recipe in *Chapter 7, Working with Web Servers*, but now we only store the data locally in IndexedDB. The following is the relevant code from `using_indexeddb.dart`:

```
import 'dart:indexed_db';

void main() {
    //test if browser supports IndexedDB:
    if (!IdbFactory.supported) {
        window.alert("Sorry, this browser does not support IndexedDB");
        return;
    }
    js = new JobStore();
    //creating and opening the database:
    js.openDB();
    querySelector("#store").onClick.listen(storeData);
}

storeData(Event e) {
    var job = _jobData();
    //writing data to IndexedDB
    js.add(job);
}
```

It is important that the data access code is isolated from the business logic code in `job.dart`. This is according to the principle of separation of concerns, which makes it a lot easier to change to another database system; only the data access code needs to be changed. The functionalities required to work with IndexedDB is found in the `JobStore` class in `jobstore_idb.dart`:

```
library store;

import 'dart:html';
import 'dart:async';
```

```
import 'dart:indexed_db';
import 'job.dart';

class JobStore {
  static const String JOB_STORE = 'jobStore';
  static const String TYPE_INDEX = 'type_index';
  Database _db;
  final List<Job> jobs = new List();

  Future openDB() {
    return window.indexedDB
      .open('JobDB', version: 1, onUpgradeNeeded: _initDb)
      .then(_loadDB)
      .catchError(print);
  }

  void _initDb(VersionChangeEvent e) {
    _db = (e.target as Request).result;
    var store = _db.createObjectStore(JOB_STORE, autoIncrement: true);
    store.createIndex(TYPE_INDEX, 'type', unique: false);
  }

  Future add(Job job) {
    // create transaction and get objectstore:
    var trans = _db.transaction(JOB_STORE, 'readwrite');
    var store = trans.objectStore(JOB_STORE);
    store.add(job.toMap())
    // called when add completes
    .then((addedKey) {
      print(addedKey);
      job.dbKey = addedKey;
    });
    return trans.completed.then((_) {
      // called when transaction completes
      jobs.add(job);
      return job;
    });
  }

  Future _loadDB(Database db) {
    _db = db;
    var trans = db.transaction(JOB_STORE, 'readonly');
    var store = trans.objectStore(JOB_STORE);
```

```
var cursors = store.openCursor(autoAdvance: true).
    asBroadcastStream();
cursors.listen((cursor) {
    var job = new Job.fromJson(cursor.value);
    jobs.add(job);
});

return cursors.length.then((_) {
    return jobs.length;
});
}

Future update(Job job) {
    var trans = _db.transaction(JOB_STORE, 'readwrite');
    var store = trans.objectStore(JOB_STORE);
    return store.put(job.toMap());
}

Future remove(Job job) {
    var trans = _db.transaction(JOB_STORE, 'readwrite');
    var store = trans.objectStore(JOB_STORE);
    store.delete(job.dbKey);
    return trans.completed
        .then((_) {
            job.dbKey = null;
            jobs.remove(job);
        });
}

Future clear() {
    var trans = _db.transaction(JOB_STORE, 'readwrite');
    var store = trans.objectStore(JOB_STORE);
    store.clear();
    return trans.completed
        .then((_) {
            jobs.clear();
        });
}
```

The following is a screenshot along with a view of the IndexedDB database in Chrome Developer Tools:

The screenshot shows a web browser window with a form titled "Posting job data". The form fields are: Company: Microsoft, Choose a job type: Tester, Salary: 6600, Date posted: 28/06/2014, and Job is vacant: checked. Below the form are "Store" and "Clear" buttons. The browser's address bar shows "127.0.0.1:8080/using_indexeddb.html". The page title is "Seth Ladd's Blog". The developer tools Resources panel is open, showing the IndexedDB section. The database structure is as follows:

#	Value
0	{type:Web developer, salary:7500, company:Google, posted:"2014-06-30T00:00:00.000", open}
1	{type:Tester, salary:6600, company:Microsoft, posted:"2014-06-28T00:00:00.000", open}

How it works...

Interacting with IndexedDB is implemented in the `dart : indexed_db` library, so we have to import it wherever needed. It is always good to test whether the browser can store data in IndexedDB. To determine whether IndexedDB is supported, use the `supported` getter from the `IdbFactory` class. All interactions with IndexedDB are asynchronous and return `Futures`.

As shown in the second comment, `openDB()` is called. The `window.indexedDB.open` method is the method used to open or create a new database. If the given database name and version already exist, then that database is opened. For a new database name or higher version number, the upgrade needed event is triggered. In its event handler, this lacks clarity; insert object type for the screen text term too, where the records get an automatically incremented key, is created within database `JobDB`. We also show how to create an index in `jobstore` in the `type` field of class `Job` with the `createIndex` method. An IndexedDB database can contain many object stores, and each store can have many indexes. Handling the possible errors when opening a database with `.catchError` is really important. We also keep a central `Database` object `_db` to be used in the database methods.



A common scenario is that all the records from a certain store are read into the app to show them after opening the database. This is done in `_loadDB`. If we look at the other methods (`add`, `update`, `remove`, and `clear`), we will see a common pattern, which all database operations must perform within a `Transaction` object. This retrieves as parameters the object store name `JOB_STORE` and a certain mode such as `readonly` or `readwrite`. Reading a number of records, such as in `_loadDB`, works through the opening cursor and then listening to it (using `listen`). Each `listen` event reads a new record through `cursor.value`, which is transformed into a `Job` object and added to the list.

An object to be stored must be given in the map format to the store's `add` method. When `autoincrement` is set to `true` for this store, the generated key number is returned as `addedKey`. When the transaction is complete (`trans.completed.then`), we add the newly created job to our list. The same pattern is followed in the `update`, `remove`, and `clear` methods, which call the methods `put`, `delete`, and `clear`, respectively, in the object store.

See also

- ▶ Refer to the *Using Browser Local Storage* recipe in *Chapter 5, Handling Web Applications*, if you want to compare local storage with IndexedDB

Using Lawndart to write offline web apps

What if the main requirement is that your app can work detached, providing universal offline key-value storage, whether IndexedDB is supported by your client's browsers or not? Then, Lawndart comes to the rescue.

Lawndart (<https://github.com/sethladd/lawndart>, but available via pub package) is not a new database, but rather a manager, which automatically chooses the best local storage mechanism available on the client. It was developed by Seth Ladd as a Dart rework of Lawnchair (<http://brian.io/lawnchair/>). You can see it in action in the project `using_lawndart`.

How to do it...

Import the Lawndart package by adding `lawndart: any` to your `pubspec.yaml` file. The following is the relevant code from the startup script `using_lawndart.dart`:

```
void main() {
  js = new JobStore();
  // 1- creating and opening the database:
```

```
js.open();
querySelector("#store").onClick.listen(storeData);
}

storeData(Event e) {
  var job = _jobData();
  // 2- writing data to storage
  js.add(job);
}
The database specific code is isolated in jobstore_lawndart.dart:
library store;

import 'dart:async';
import 'package:lawndart/lawndart.dart';
import 'job.dart';

class JobStore {
  static const String JOB_DB = 'jobDb';
  static const String JOB_STORE = 'jobStore';
  final List<Job> jobs = new List();
  // 3- making a Store object:
  var _store = new Store(JOB_DB, JOB_STORE);

  Future open() {
    // 4- opening storage and retrieving all records:
    return _store.open().then(_loadDB).catchError(print);
  }

  _loadDB(_) {
  Stream dataStream = _store.all();
  return dataStream.forEach((dbjob) {
    var job = new Job.fromJson(dbjob);
    jobs.add(job);
  });
}

add(Job job) {
  // 5- storing data:
  _store.save(job.toMap(), job.dbKey.toString()).then((addedKey) {
    jobs.add(job);
  });
}
```

```
Future update(Job job) {
    return _store.save(job.toMap(), job.dbKey.toString());
}

Future remove(Job job) {
    return _store.removeByKey(job.dbKey.toString());
}

clear() {
    _store.nuke().then((_) {
        jobs.clear();
    });
}
```

How it works...

Lawndart presents an asynchronous, but consistent, interface to the local storage, IndexedDB, and Web SQL. Your app simply works with an instance of the class `Store`, and the factory constructor will try IndexedDB, Web SQL, and finally, local storage. This is shown in the third comment, where an object of class `Store` is made; the object also constructs the local database needed. In the first comment, the `open` method in the `JobStore` object triggers `openmethod` on the `Store` object (fourth comment). The `_loadDB` option then reads the entire store; this is accomplished through the `all` method, which returns `Stream` of values. Adding and updating data is done by calling the `save` method on the `store` object (fifth 5). Deleting a record is done through `removeByKey`; clearing an entire database needs the `nuke` method. Notice that the code that uses Lawndart is much cleaner than IndexedDB, as shown in the previous recipe.

See also

- ▶ [Cargo](#) is another pub package developed by Joris Hermans, which accomplishes a similar goal. It is a storage package that abstracts local storage, storage on the client as well as on the server, and stores JSON files on the disk. For more information, refer to <https://pub.dartlang.org/packages/cargo>.
- ▶ The complete API documents can be found at <http://www.dartdocs.org/documentation/lawndart/>.

Storing data in MySQL

MySQL is undeniably the most popular open source SQL database. Dart can talk to MySQL using the pub package `sqljockey` by James Ots (<https://github.com/jamesots/sqljockey>). In this section, we will demonstrate how to use this driver step by step. You can see it in action in the `using_mysql` project.

Getting ready

- ▶ To get the database software, download and install the MySQL Community Server installer from <http://dev.mysql.com/downloads/mysql/>. This is straightforward. However, if you need any help with the installation, visit <http://dev.mysql.com/doc/refman/5.7/en/installing.html>.
- ▶ Run the MySQL database system by starting `mysqld` on a command prompt from the bin folder of the MySQL installation. We need to create a database and table to store data. The easiest way is to start the MySQL Workbench program, make a connection, and then click on the button **Create a new schema in the connected server**, name it `jobsdb`, and click on **apply**.
- ▶ Select the schema by double-clicking on it, and then clicking on the button on the right to the previous button **Create a new table in the active schema**. Name the table `jobs`, and create the `dbKey`, `type`, `salary`, `company`, `posted` and `open` columns; `dbKey` is the primary key. To import the driver to your application, add `sqljockey` to `pubspec.yaml` and save it. A `pub get` command is then done automatically.

How to do it...

The application starts with `using_mysql.dart`, where a `JobStore` object is created, the database is opened, records are written to the `jobs` table, and then these records are retrieved, as shown in the following code:

```
import 'job.dart';
import 'jobstore_mysql.dart';

Job job;
JobStore js;

void main() {
    js = new JobStore();
    // 1- create some jobs:
    job = new Job("Web Developer", 7500, "Google", new DateTime.now());
    js.jobs.add(job);
    job = new Job("Software Engineer", 5500, "Microsoft",
        new DateTime.now());
```

Working with Databases

```
js.jobs.add(job);
job = new Job("Tester", 4500, "Mozilla", new DateTime.now());
js.jobs.add(job);
// 2- opening the database:
js.open();
// 3- storing data in database:
js.storeData();
// 4- retrieving and displaying data from database:
js.readData();
}
```

After running the preceding code, we can verify that the insertions succeeded in MySQL Workbench in the `jobs` table:

The screenshot shows the MySQL Workbench interface. On the left is the Navigator pane with sections for MANAGEMENT, INSTANCE, PERFORMANCE, and SCHEMAS. In the center, a SQL editor window titled 'SQL File 4' contains the query: `1 • SELECT * FROM jobsdb.jobs;`. Below the query is a Result Grid showing the data from the 'jobs' table. The table has columns: dbKey, type, salary, company, posted, and open. There are four rows of data: 1. dbKey: 1, type: Web Developer, salary: 7500, company: Google, posted: 2014-07-02 14:34:12, open: 1; 2. dbKey: 2, type: Software Engineer, salary: 5500, company: Microsoft, posted: 2014-07-02 14:34:12, open: 1; 3. dbKey: 3, type: Tester, salary: 4500, company: Mozilla, posted: 2014-07-02 14:34:12, open: 1; and a final row with NULL values for all columns.

dbKey	type	salary	company	posted	open
1	Web Developer	7500	Google	2014-07-02 14:34:12	1
2	Software Engineer	5500	Microsoft	2014-07-02 14:34:12	1
3	Tester	4500	Mozilla	2014-07-02 14:34:12	1
*	NULL	NULL	NULL	NULL	NULL

The `readData` method reads the data from the `jobs` table and prints it to the console:

dbKey: 1, type : Web Developer, salary: 7500, company: Google, posted: 2014-07-02 14:34:12.000, open: 1

dbKey: 2, type : Software Engineer, salary: 5500, company: Microsoft, posted: 2014-07-02 14:34:12.000, open: 1

dbKey: 3, type : Tester, salary: 4500, company: Mozilla, posted: 2014-07-02 14:34:12.000, open: 1

After the first run, `readData` will not show any output because it gets executed before the insertions in the database are complete; verify the output through a second run.

The interaction with MySQL through `sqljockey` takes place in the `JobStore` class in `jobstore_mysql.dart`:

```
import 'package:sqljockey/sqljockey.dart';
import 'package:options_file/options_file.dart';
import 'job.dart';

ConnectionPool pool;

class JobStore {
    final List<Job> jobs = new List();
    Job job;

    // 5- opening a connection to the database:
    open() {
        pool = getPool(new OptionsFile('connection.options'));
    }

    ConnectionPool getPool(OptionsFile options) {
        String user = options.getString('user');
        String password = options.getString('password');
        int port = options.getInt('port', 3306);
        String db = options.getString('db');
        String host = options.getString('host', 'localhost');
        return new ConnectionPool(host: host, port: port, user: user,
            password: password, db: db);
    }

    storeData() {
        for (job in jobs) {
            insert(job);
        }
    }

    // 6- inserting a record in a table:
    insert(Job job) {
        var jobMap = job.toMap();
        pool.prepare('insert into jobs (dbKey, type, salary, company, posted,
open) values (?, ?, ?, ?, ?, ?, ?)').then((query) {
            var params = new List();
            params.add(job.dbKey);
```

```
params.add(job.type);
params.add(job.salary);
params.add(job.company);
params.add(job.posted);
params.add(job.open);
return query.execute(params);
}).then((_) {
}).catchError(print);
}

readData() {
  pool.query('select * from jobs').then((results) {
    processJob(results);
  });
}

processJob(results) {
  results.forEach((row) {
    print('dbKey: ${row.dbKey}, type : ${row.type}, ' 'salary:
      ${row.salary}, company: ${row.company}, ' 'posted: ${row.posted},
      open: ${row.open}');
  });
}

// 7- updating a record in a table:
update(Job job) {
  var jobMap = job.toMap();
  pool.prepare('update jobs set type = ?, salary = ?, company = ?,
    posted = ?, open = ? where dbKey = ?').then((query) {
  var params = new List();
  params.add(job.dbKey);
  params.add(job.type);
  params.add(job.salary);
  params.add(job.company);
  params.add(job.posted);
  params.add(job.open);
  return query.execute(params);
}).then((_) {
}).catchError(print);
}

// 8- deleting a record in a table:
delete(Job job) {
  var jobMap = job.toMap();
```

```
pool.prepare('delete from jobs where dbKey = ?').then((query) {
  var params = new List();
  params.add(job.dbKey);
  return query.execute(params);
}).then((_) {
}).catchError(print);
}
}
```

How it works...

The connection information for the database is stored in the `connection.options` file. In the fifth comment, a connection to the database is opened by calling `getPool`, which returns a `ConnectionPool` object that maintains a pool of database connections. The `getPool` option takes as argument an `OptionsFile` object, from the `options_file` package, and reads the information from `connection.options`:

```
# connection.options to define how to connect to a mysql db
user=root
password=????? # substitute your password here
# port defaults to 3306
port=3306
db=jobsdb
# host defaults to localhost
host=localhost
```

All interactions with the database work via queries. If there is a free connection in `ConnectionPool` when queries are executed, it will be used; otherwise, the query is queued until there is a free connection. As we can see in `readData`, a select query is made via `pool.query(selectStr);`. This returns a `Future` object, so the processing of the results takes place in the `then` section. This can be done using the following code:

```
results.forEach((row) {
  print('dbKey: ${row.dbKey}, type : ${row.type}, ...');
```

The preceding line can also be written as follows:

```
results.forEach((row) {
  print('dbKey: ${row[0]}, type : ${row[1]}, ...');
```

Insert, update, and delete queries (coded in the methods with the same name) have to first go through the `prepare` stage and then the `execute` stage. Let's see a few examples. For insert, this query becomes as follows (refer to the sixth comment):

```
pool.prepare('insert into jobs (dbKey, type, salary, company,  
posted, open) values (?, ?, ?, ?, ?, ?)').then((query) {})
```

An insert query on a table that contains an autoincrement field will return the value of that field in `result.insertId`. For update, this query becomes as follows (refer to the seventh comment):

```
pool.prepare('update jobs set type = ?, salary = ?, company = ?,  
posted = ?, open = ? where dbKey = ?').then((query) {})
```

The query for delete is as follows (refer to comment 8):

```
pool.prepare('delete from jobs where dbKey = ?').then((query) {})
```

The `?` character represents values to be substituted in the SQL string. These values are placed in the specified order in the `params` list, which is given as an argument to `query.execute`. A query with multiple parameter sets can be executed with `query.executeMulti()`. If you need a number of queries to be executed as a whole (or all or none succeed), use the `Transaction` class from `sqljockey` in the following format:

```
pool.startTransaction().then((trans) {  
  trans.query('...').then((result) {  
    trans.commit().then(() {...});  
  });  
});
```

There's more...

The complete API documents can be found at <http://jamesots.github.io/sqljockey/docs/>.

Storing data in PostgreSQL

PostgreSQL is another popular open source SQL database. Dart can talk to PostgreSQL using the pub package `postgresql` by Greg Lowe (<https://github.com/xxgreg/postgresql>). In this section, we will demonstrate how to use this driver step by step. You can see it in action in the `using_postgresql` project.

Getting ready

To get the database software, download and install the PostgreSQL Server installer from <http://www.postgresql.org/download/> using the following steps:

1. The database server is configured to start automatically. We need to create a database and table to store data. The easiest way is to start the pgAdmin program, make a new connection, and then select **Edit**, **New Object**, and **New Database** from the menu, name it **jobsdb**, and click on **OK**.
2. Select the public schema, and again select **Edit**, **New Object**, and then **New Table**.
3. Name the table **jobs**, and create **dbKey**, **type**, **salary**, **company**, **posted**, and **open** as columns; **dbKey** is the primary key.
4. To import the driver to your application, add `postgresql` to the `pubspec.yaml` file and save it. A `pub get` command is then done automatically.

How to do it...

The application starts from `using_postgresql.dart`, where a `JobStore` object is created, the database is opened, records are written to the `jobs` table, and then these records are retrieved:

```
import 'job.dart';
import 'jobstore_postgresql.dart';

Job job;
JobStore js;

void main() {
    js = new JobStore();
    // 1- create some jobs:
    job = new Job("Web Developer", 7500, "Google", new DateTime.now());
    js.jobs.add(job);
    job = new Job("Software Engineer", 5500, "Microsoft",
        new DateTime.now());
    js.jobs.add(job);
    job = new Job("Tester", 4500, "Mozilla", new DateTime.now());
    js.jobs.add(job);
    // 2- storing data in database:
    js.openAndStore();
    // 3- retrieving and displaying data from database:
    js.openAndRead();
}
```

After running the preceding code, we can verify via pg Admin in the `jobs` table that the insertions succeeded, as shown in the following screenshot:

dbKey	type	salary	company	posted date	open
integer	character varying(75)	integer	character varying(150)	date	boolean
1	Web Developer	7500	Google	2014-07-03	TRUE
2	Software Engineer	5500	Microsoft	2014-07-03	TRUE
3	Tester	4500	Mozilla	2014-07-03	TRUE

Data in PostgreSQL

The `readData` method reads the data from the `jobs` table and prints it to the console:

- 1 - Web Developer - 7500 - Google - 2014-07-03 00:00:00.000 true**
- 2 - Software Engineer - 5500 - Microsoft - 2014-07-03 00:00:00.000 true**
- 3 - Tester - 4500 - Mozilla - 2014-07-03 00:00:00.000 true**

The interaction with PostgreSQL through the driver takes place in the `JobStore` class in `jobstore_postgresql.dart`:

```
library store;

import 'package:postgresql/postgresql.dart';
import 'job.dart';

class JobStore {
    final List<Job> jobs = new List();
    Job job;
    Connection conn;
    var uri = 'postgres://username:passwd@localhost:5432/jobsdb';

    // 5- opening a connection to the database:
    openAndStore() {
        connect(uri).then((conn) {
            conn = _conn;
            storeData();
        })
        .catchError(print);
    }
}
```

```
storeData() {
    for (job in jobs) {
        insert(job);
    }
    // 6- close the database connection:
    close();
}

// 7- inserting a record in a table:
insert(Job job) {
    var jobMap = job.toMap();
    conn.execute('insert into jobs values (@dbKey, @type, @salary,
        @company, @posted, @open)',
        jobMap)
    .then((_) { print('inserted'); })
    .catchError(print);
}

openAndRead() {
    connect(uri).then((_conn) {
        conn = _conn;
        readData();
    })
    .catchError(print);
}

// 8- reading records from a table:
readData() {
    conn.query('select * from jobs').toList().then((results) {
        processJob(results);
        close();
    });
}

// 9- working with record data:
processJob(results) {
    for (var row in results) {
        // Refer to columns by name:
        print('${row.dbKey} - ${row.type} - ${row.salary} -
            ${row.company} - ${row.posted} ${row.open}');
        // print(row[0]);      // Or by column index.
    }
}
```

```
close() { conn.close(); }

// 10- updating a record in a table:
update(Job job) {
    var jobMap = job.toMap();
    conn.execute('update jobs set type = @type, salary = @salary,
        company = @company, '
        'posted = @posted, open = @open where dbKey = @dbKey', jobMap)
    .then((_) { print('updated'); })
    .catchError(print);
}

// 11- deleting a record in a table:
delete(Job job) {
    var jobMap = job.toMap();
    conn.execute('delete from jobs where dbKey = @dbKey', jobMap)
    .then((_) { print('deleted'); })
    .catchError(print);
}
```

How it works...

Obtaining a connection with a Postgres database needs a valid connection string of the following form:

```
var uri = 'postgres://username:password@localhost:5432/database';
```

This is given as an argument to the `connect` method, as shown in the fifth comment; the writing of the data in `storeData` is done in the `callback` handler, in order to be sure that we have a `Connection` object at that point. Inserting a record happens in comment 7; the values to be inserted in the `insert` SQL in the `@` markers must be given through a map (here `jobMap`):

```
conn.execute('insert into jobs values (@dbKey, @type, @salary,
    @company, @posted, @open)', jobMap).then((_) { ... })
```

So all the `insert`, `update`, and `delete` queries are given as string arguments to the method `conn.execute`. Strings will be escaped to prevent SQL injection vulnerabilities. After the inserts, we explicitly close the connection with `conn.close()` to save resources. Select queries are performed through `conn.query`:

```
conn.query('select * from jobs').toList().then((results) { ... })
```

The processing of the results is done in the callback handler:

```
for (var row in results) {  
    print('${row.type} - ${row.salary}');  
    print(row[1]);
```

Fields can be retrieved by their name or index. As always, we catch the errors with `catchError`, which at least prints the error to the console. Similar to MySQL, a connection `Pool` object can be used to avoid the overhead of obtaining a connection for each request:

```
var pool = new Pool(uri, min: 2, max: 5);  
pool.start().then((_) {  
    print('min amount connections established.');// Obtain connection from pool  
    pool.connect().then((conn) { // Obtain connection from pool  
        // ... }  
    }  
}
```

The connection method `runInTransaction` allows queries that need to be executed in "an all or none" way, to be bundled in a transaction.

See also

- ▶ The complete API docs can be found at <http://www.dartdocs.org/documentation/postgresql/0.2.14/index.html#postgresql>

Storing data in Oracle

Oracle is the most popular commercial SQL database. Dart can talk to Oracle using the pub package `oracledart` by Alexander Aprelev (<https://github.com/aam/oracledart>). This is a Dart native extension of C++, using the `dart_api` interface to integrate into Dart. It requires Oracle Instant Client to be present on the machine, and its OCCI binaries must be included in the PATH variable.

In this section, we will demonstrate how to use this driver step by step. You can see it in action in the project `using_oracle`.

Getting ready

To get the database software, download and execute the Oracle installer from <http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index-092322.html>.

- ▶ The database server is started through the menu option **Start Database**.
- ▶ We need to create a table to store data. Start the SQL command-line terminal and paste the contents of the script `jobs.txt`. This creates the table `jobs`, together with the `dbKey`, `type`, `salary`, `company`, `posted`, and `open` columns; `dbKey` is the primary key.
- ▶ To import the driver to your application, add `oracledart` to `pubspec.yaml` and save it. A `pub get` command is then done automatically.

How to do it...

The application starts from `using_oracle.dart`, where a `JobStore` object is created, the database is opened, records are written to the `jobs` table, and then these records are retrieved. The code from `using_oracle.dart` is identical to the code from `using_postgresql.dart`, except that we now import `jobstore_oracle.dart`; so please refer to the previous recipe. The `jobstore_oracle.dart` file contains the code to talk to the database driver:

```
// 1- importing the driver:  
import 'package:oracledart/oracledart.dart';  
import 'job.dart';  
  
class JobStore {  
    final List<Job> jobs = new List();  
    Job job;  
    OracleConnection conn; // connection object  
    OracleResultset resultset;  
    OracleStatement stmt;  
    var connStr = ' (DESCRIPTION= " (ADDRESS= (PROTOCOL=TCP)  
        (HOST=oracledb) (PORT=1521) ) "  
    ' "(CONNECT_DATA= (SERVICE_NAME=XE) (SERVER=DEDICATED) ) " ' ;  
  
    // 2- opening a connection to the database:  
    openAndStore() {  
        connect("SYS", "avalon", connStr).then((oracleconnection) {  
            conn = oracleconnection;  
            print('connected with Oracle!');  
            storeData();  
        });  
    }  
}
```

```
        }).catchError(print);
    }

    storeData() {
        for (job in jobs) {
            insert(job);
        }
    }

// 3- inserting a record in a table:
insert(Job job) {
    var jobMap = job.toMap();
    var insertSql = 'insert into jobs values (:1, :2, :3, :4, :5, :6)';
    stmt = conn.createStatement(insertSql);
    stmt.setInt(1, jobMap['dbKey']);
    stmt.setString(2, jobMap['type']);
    stmt.setInt(3, jobMap['salary']);
    stmt.setString(4, jobMap['company']);
    stmt.setString(5, jobMap['posted']);
    stmt.setString(6, jobMap['open']);
    stmt.executeQuery();
}

openAndRead() {
    connect("SYS", "avalon", connStr).then((oracleconnection) {
    conn = oracleconnection;
    readData();
    }).catchError(print);
}

// 8- reading records from a table:
readData() {
    resultset = conn.select("select * from jobs");
    processJob(resultset);
}

// 9- working with record data:
processJob(results) {
    while (resultset.next()) {
        print('dbKey: ${resultset.getInt(0)}');
        print('type: ${resultset.getString(1)}');
        print('salary: ${resultset.getInt(2)}');
        print('company: ${resultset.getString(3)}');
```

```
    print('posted: ${resultset.getString(4)}');
    print('open: ${resultset.getString(5)})';
}
}
}
```

How it works...

First, import the `oracledart` package. The `connect` method takes the user, his/her password, and then a connection string. In its callback, an `OracleConnection` object `conn` is made available. Querying tables is done with `conn.select(selectSql)`, where `selectSql` is the selected SQL string. This returns an `OracleResultset` object that you can iterate throughout with `next()`. Values of fields can only be extracted by an index with `getInt` or `getString`.

If you need parameters in your statement, as is probably the case for insert, update, or delete statements, you first need to call `conn.createStatement(sqlStr)`, where `sqlStr` contains `:i` indicators (`i` equals to `i`th index, starting from 0). These `:i` position holders must be filled with `stmt.setInt(i, value)` or `stmt.setString(i, value)`. Then, the statement can be executed with `stmt.executeQuery()`.

There's more...

In this and the previous recipes, we discussed the SQL database servers that Dart can talk to at this time using specialized drivers. However, for example, for the popular Microsoft SQL Server there is no driver yet. In this case, we can use the pub `odbc` package by Juan Mellado (<https://code.google.com/p/dart-odbc/>). The ODBC binding is made with a Dart native extension, which makes it a bit more involved to be used, and for now it only exists in a 32-bit version.

Storing data in MongoDB

MongoDB, by the company with the same name (<http://www.mongodb.org/>), is the most popular database among the NoSQL databases. Let's look at some facts about MongoDB:

- ▶ MongoDB is an open source, distributed, document-oriented database; each data record is actually a document.
- ▶ A table is called a collection in MongoDB. Documents are stored in a JSON-like format called BSON.
- ▶ The most advanced driver from Dart to MongoDB is the pub package `mongo_dart` by Vadim Tsushko, Ted Sander, and Paul Evans. This recipe will show you how to create, read, update, and delete actions in a MongoDB database from a Dart app. You can see it in action in the `using_mongodb` project.

Getting ready

Install the latest production release for your system from <http://www.mongodb.org/downloads>. This is easy. However, if you need more details, refer to <http://docs.mongodb.org/manual/installation/>. Start the mongod server process (for example, from c:\mongodb\bin on Windows) before the Dart app. To make the jobsdb database, start a mongo shell and type use jobsdb; it's that simple. Alternatively, this can also be done via mongo_dart. NoSQL databases are schemaless, so the collections (which is what tables are called here) are created when the first document (or record) is inserted.

How to do it...

The application starts from using_mongodb.dart, where a JobStore object is created, the database is opened, records are written to the jobs table, and then these records are retrieved. The code from using_mongodb.dart is identical to the code from using_postgresql.dart, except that we now import jobstore_mongodb.dart; so please refer to the *Storing data in PostgreSQL* recipe. When running this script, first comment out js.openAndRead();. In the second run, uncomment this and comment out js.openAndStore();; otherwise, the reads take place before the inserts are completed. The jobstore_mongodb.dart code contains the code to talk to the database driver:

```
import 'package:mongo_dart/mongo_dart.dart';
import 'job.dart';

const String DEFAULT_URI = 'mongodb://127.0.0.1/';
const String DB_NAME = 'jobsdb';
const String COLLECTION_NAME = 'jobs';

class JobStore {
    final List<Job> jobs = new List();
    Job job;
    Db db;
    DbCollection jobsColl;

    JobStore() {
        // 1- make a new database
        db = new Db('${DEFAULT_URI}${DB_NAME}');
        // make a new collection
        jobsColl = db.collection(COLLECTION_NAME);
    }

    openAndStore() {
        db.open().then((_) {

```

```
storeData();
}).catchError(print);
}

storeData() {
for (job in jobs) {
insert(job);
}
}

// 2- inserting a document in a collection:
insert(Job job) {
var jobMap = job.toMap();
jobsColl.insert(jobMap).then((_) {
print('inserted job: ${job.type}');
}).catchError(print);
}

openAndRead() {
db.open().then((_) {
readData();
}).catchError(print);
}

// 3- reading documents:
readData() {
jobsColl.find().toList().then((jobList) {
processJob(jobList);
}).catchError(print);
}

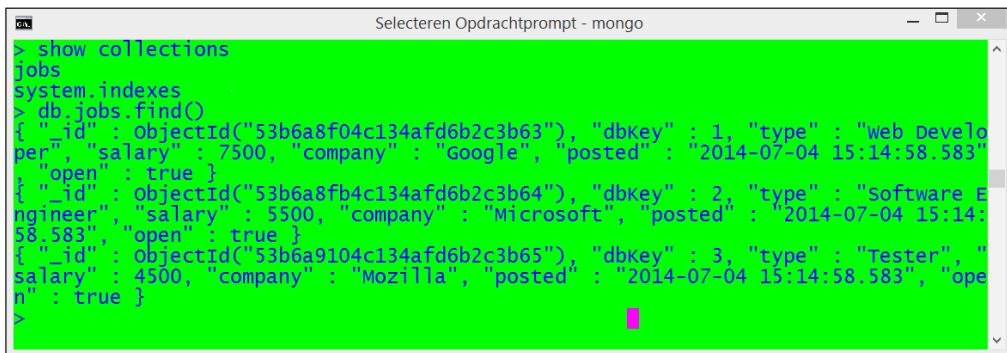
// 4- working with document data:
processJob(jobList) {
jobList.forEach((jobMap) {
Job job = new Job.fromMap(jobMap);
print('${job.dbKey} - ${job.type} - ${job.salary} - '
'${job.company} - ${job.posted} ${job.open}');
});
}

// 5- updating a document:
update(Job job) {
var jobMap = job.toMap();
```

```
jobsColl.update({ "dbKey": jobMap["dbKey"] }, jobMap).then((_) {
  print('job updated');
}).catchError(print);
}

// 6- deleting a document:
delete(Job job) {
  var jobMap = job.toMap();
  jobsColl.remove(jobMap).then((_) {
  print('job updated');
}).catchError(print);
}
}
```

After running the preceding script, verify in a mongo shell with `use jobsdb` and `db.jobs`.
`find()` that the documents have been inserted, as shown in the following screenshot:



The screenshot shows a terminal window titled "Selecteren Opdrachtprompt - mongo". The command `show collections` is run, followed by `db.jobs.find()`. The output displays three documents from the 'jobs' collection, each representing a job listing with fields like '_id', 'dbkey', 'type', 'salary', 'company', and 'posted'. The 'open' field is set to true for all documents.

```
> show collections
[...]
> db.jobs.find()
[{"_id": ObjectId("53b6a8f04c134af6b2c3b63"), "dbkey": 1, "type": "web developer", "salary": 7500, "company": "Google", "posted": "2014-07-04 15:14:58.583", "open": true}, {"_id": ObjectId("53b6a8fb4c134af6b2c3b64"), "dbkey": 2, "type": "Software Engineer", "salary": 5500, "company": "Microsoft", "posted": "2014-07-04 15:14:58.583", "open": true}, {"_id": ObjectId("53b6a9104c134af6b2c3b65"), "dbkey": 3, "type": "Tester", "salary": 4500, "company": "Mozilla", "posted": "2014-07-04 15:14:58.583", "open": true}]
```

How it works...

The `mongo_dart` library makes it very easy to work with MongoDB. In comment 1, we created the `Db` and `DbCollection` objects in the constructor of `JobStore`. To manipulate documents, you have to first call the `open` method on the `Db` object.

To write a document to the collection, use the `insert` method on the `DbCollection` object, but its argument must be in the `Map` format. Reading documents is done with the `find` method. The list that is returned contains items of type `Map`, so to construct real objects, we have to make a new named constructor `fromMap` in class `Job`. Updating a document uses a method of the same name, but its first argument must be a selector to find the document that must be updated. Deleting a document uses the `remove` method.

See also

- ▶ The complete API documents can be found at http://www.dartdocs.org/documentation/mongo_dart/0.1.39/index.html#mongo_dart
- ▶ Another MongoDB client by Vadim Tsushko is an object-document mapper tool called Objectory, which can be used on the client as well as the server (<https://github.com/vadimtsushko/objectory>)

Storing data in RethinkDB

RethinkDB (<http://www.rethinkdb.com/>) is a simple NoSQL database that stores JSON documents. Its main focus lies on ease of use, both for the developer, with an intuitive query language that can simulate table joins, as well as for the administrator, with friendly web tools to monitor, shard, and replicate. Another advantage is its automatic parallelization of queries. At the moment, the database system runs on OS X and a lot of Linux flavors.

We will talk to RethinkDB with a driver available on pub package called `rethinkdb_driver`, developed by William Welch (<https://github.com/billysometimes/rethinkdb>). You can see it in action in the `using_rethinkdb` project.

Getting ready

Install the latest production release for your system from <http://www.rethinkdb.com/docs/install/>. Then, perform the following steps:

1. Start the RethinkDB server by issuing the command `rethinkdb` in a terminal.
2. Then, go to `localhost:8080` in your browser – this starts an administrative UI where you can control the database server (from one machine to a cluster).
3. Click on the **Tables** tab at the top and use the **Add Database** button to create the database `jobsdb`.
4. To create the `jobs` table, click on the **Tables** tab at the top of the page and then use the **Add Table** button.
5. The table `jobs` has the columns `dbKey`, `type`, `salary`, `company`, `posted`, and `open` columns; `dbKey` is the primary key.

How to do it...

The application starts from `using_rethinkdb`, where a `JobStore` object is created, the database is opened, records are written to the `jobs` table, and then these records are retrieved. The code from `using_rethinkdb` is identical to the code from `using_postgresql.dart`, except that we now import `jobstore_rethinkdb.dart`, so please refer to the *Storing data in PostgreSQL* recipe. The `jobstore_rethinkdb.dart` file contains the following code to talk to the database driver:

```
import 'package:rethinkdb_driver/rethinkdb_driver.dart';
import 'job.dart';

class JobStore {
    final List<Job> jobs = new List();
    Job job;
    Rethinkdb rdb = new Rethinkdb();
    Connection conn;

    // opening a connection to the database:
    openAndStore() {
        rdb.connect(db: "jobsdb", port:8000, host: "127.0.0.1").
            then((conn) {
                conn = _conn;
                storeData(conn);
            }).catchError(print);
    }

    storeData(conn) {
        List jobsMap = new List();
        for (job in jobs) {
            var jobMap = job.toMap();
            jobsMap.add(jobMap);
        }
        // storing data:
        rdb.table("jobs").insert(jobsMap).run(conn)
            .then((response)=>print('documents inserted'))
            .catchError(print);
        // close the database connection:
        close();
    }

    openAndRead() {
        rdb.connect(db: "jobsdb", port:8000, host: "127.0.0.1").then((conn) {
            conn = _conn;
        });
    }
}
```

```
readData(conn);
}).catchError(print);
}

// reading documents:
readData(conn) {
    rdb.table("jobs").getAll("1,2,3").run(conn).then((results) {
    processJob(results);
    close();
});
}

// working with document data:
processJob(results) {
    for (var row in results) {
        // Refer to columns by name:
        print('${row.dbKey} - ${row.type} - ${row.salary} - ${row.company} -
            ${row.posted} ${row.open}');
    }
}

close() {
    conn.close();
}

// updating a document:
update(Job job) {
    rdb.table("jobs").update({"dbKey":job.dbKey})
    .run(conn).then((response)=>print('document updated'))
    .catchError(print);
}

// deleting a document:
delete(Job job) {
    rdb.table("jobs").get(job.dbKey).delete()
    .run(conn).then((response)=>print('document deleted'))
    .catchError(print);
}
```

How it works...

As always, we have to import the driver's code. Then, we define a `Rethinkdb` object on which all methods are defined, and a global `Connection` object. Opening a connection is done with the following command, where the host can also be the real name of the server. The `connect` option also takes an optional authorization key argument, `authKey`. In the callback handler, all document manipulation can be done using the following code:

```
rdb.connect(db: "jobsdb", port:8000, host: "127.0.0.1")
```

For example, to read documents, use the command `getAll`, which takes a list of the document keys as an argument:

```
rdb.table("jobs").getAll(keysList).run(conn).then((results) {...})
```

To insert documents, use the `insert` command:

```
rdb.table("jobs").insert(docMap).run(conn).then((results) {...})
```

The `insert` command takes a list as an argument along with its documents, where each document is in the map format. The `update` command matches documents with their arguments, and the `delete` command needs a document key as its argument.

See also

Of course, there are many other NoSQL databases. The following are a few references to other drivers:

- ▶ For CouchDB, there is the `wilt` package (<https://pub.dartlang.org/packages/wilt>) and the `couchclient` library (<https://pub.dartlang.org/packages/couchclient>)
- ▶ For Redis, you have the `redis_client` package (https://pub.dartlang.org/packages/redis_client)
- ▶ For Memcached, there is the `memcached_client` library (https://pub.dartlang.org/packages/memcached_client)
- ▶ For Riak, there is the `riak_client` package (https://pub.dartlang.org/packages/riak_client)

10

Polymer Dart Recipes

In this chapter, we will cover the following recipes:

- ▶ Data binding with polymer.dart
- ▶ Binding and repeating over a list
- ▶ Binding to a map
- ▶ Using custom attributes and template conditionals
- ▶ Binding to an input text field or a text area
- ▶ Binding to a checkbox
- ▶ Binding to radio buttons
- ▶ Binding to a selected field
- ▶ Event handling
- ▶ Polymer elements with JavaScript interop
- ▶ Extending DOM elements
- ▶ Working with custom elements
- ▶ Automatic node finding
- ▶ Internationalizing a Polymer app

Introduction

Polymer (as defined at <http://www.polymer-project.org/>) is a web development feature designed to fully utilize the evolving web platform on modern browsers; it modularizes the way a web client interface is defined. Web pages can now be built and composed with web components that can simply be accessed by their names. Web components are reusable chunks of styled HTML5 or extensions from native HTML tags. Moreover, they enable two-way data binding to make data from code elements visible and editable in the DOM. They can also contain code to change their behavior, either in JavaScript or Dart, through a class that backs up the components. A Polymer web component thus encapsulates structure, style, and behavior. Polymer is a library on top of web components and in some browsers that don't support web components, yet, it has been a polyfill.

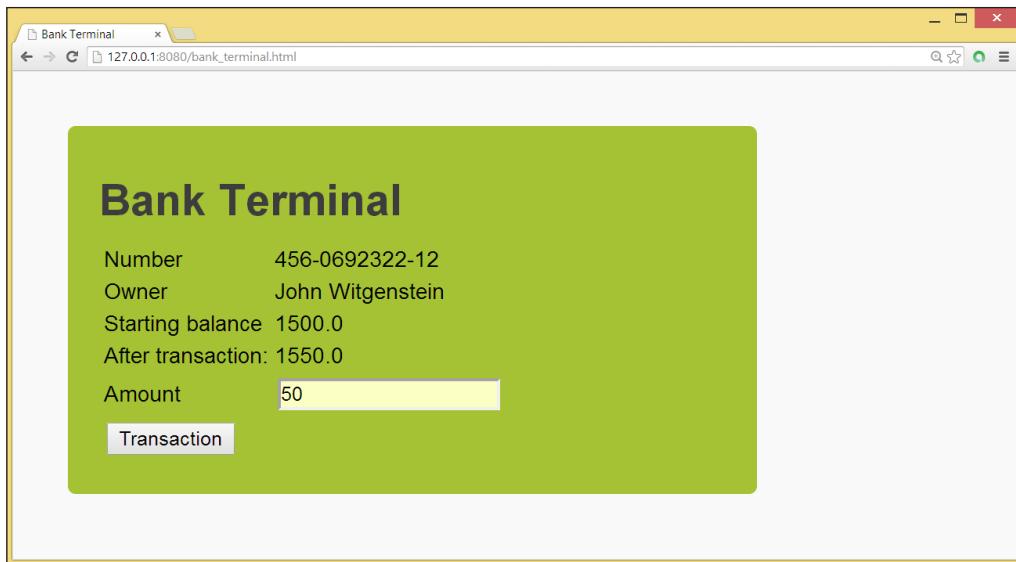
Polymer.dart is the Dart port of Polymer created and supported by the Dart team. At the time of writing this, the version of polymer.dart is 0.14.0, and it is quickly aiming towards a stable production version; its main documentation page can be found at <https://www.dartlang.org/polymer-dart/>.

Polymer.dart runs in the following browsers: IE10, IE11, Safari 6, the latest version of Chrome, the latest version of Firefox, and the latest version of Chrome for Android. A lot has changed in the polymer.dart world since its first version, so if you need to refresh your memory, refer to <https://www.dartlang.org/docs/tutorials/polymer-intro/>.

Although simple, the first recipe is a complete example showing data binding and event handling to change the state of code objects. Refer to it if you need a picture of the whole machinery of programming a component. We will look at more detailed examples of how to do certain things in the following recipes.

Data binding with polymer.dart

In the first recipe, we will go through an example that shows the complete mechanics of working with a Polymer component and data binding, and at the same time emphasizes some important best practices to work with polymer.dart. You can find the code in the project `bank_terminal`. The app creates an object of class `BankAccount` (which can be found in the `lib` folder) and populates it with the data that is shown. You can also provide a transaction amount and the balance of the account is updated. This is depicted in the following screenshot:



How to do it...

Perform the following steps to bind data with polymer.dart:

1. First, install the polymer dependency in your project by adding `polymer: >=0.11.0 <0.12.0` to the `pubspec.yaml` file. Save the file in the editor and run `pub get` from the command line. This has to be accompanied by an import line; in `web\bank_app.dart`, we use `import 'package:polymer/polymer.dart'`; but also in `web\bank_app.html` with a `<link rel="import">` tag as the first line.
2. Also, in that file, you must indicate one or more starting points of app execution in the `transformers` section, as shown in the following code:

```
transformers:  
- polymer:  
entry_points:  
- web/bank_terminal.html  
- web/index.html
```

3. The root folder of the app contains a `build.dart` file, with the following content:

```
import 'package:polymer/builder.dart';

main() {
  build(entryPoints: ['web/bank_terminal.html']);
}
```

4. The entry point `bank_terminal.html` must contain the `<script>` and `<link>` tags in the `<head>` section, in the following order:

```
<script src="packages/web_components/platform.js"></script>
<script src="packages/web_components/dart_support.js"></script>
<!-- import the bank-app custom element --&gt;
&lt;link rel="import" href="bank_app.html"&gt;
  &lt;script type="application/dart"&gt;export
    'package:polymer/init.dart';&lt;/script&gt;
&lt;script src="packages/browser/dart.js"&gt;&lt;/script&gt;</pre>
```

5. The Polymer component with the name `bank-app` is instantiated in the `<body>` section of the same entry file:

```
<bank-app></bank-app>
```

6. The component is defined in `bank_app.html` as follows:

```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="bank-app">
<style>
  .auto-style1 {
    width: 25%;
    border: 1px solid #0000FF;
  }

  .auto-style2 {
    width: 107px;
  }

  .btns {
    width: 127px;
  }

  .red {
    color: red;
  }
</style>
<template>
```

```


|                    |                                  |
|--------------------|----------------------------------|
| Number             | {{bac.number}}                   |
| Owner              | {{bac.owner.name}}               |
| Starting balance   | {{bac.balance}}                  |
| After transaction: | {{balance}}                      |
| Amount             | <input id="amount" type="text"/> |
| t                  |                                  |


</template>
<script type="application/dart" src="bank_app.dart"></script>
</polymer-element>

```

7. The code of the Polymer component can be found in bank_app.dart:

```

import 'dart:html';
import 'package:polymer/polymer.dart';
import 'package:bank_terminal/bank_terminal.dart';

@CustomTag('bank-app')
class BankApp extends PolymerElement {
  @observable BankAccount bac;
  @observable double balance;
  double amount = 0.0;

  BankApp.created() : super.created() { }

  @override

```

```

attached() {
  super.attached();
  varjw = new Person("John Witgenstein");
  bac = newBankAccount(jw, "456-0692322-12", 1500.0);
  balance = bac.balance;
}

transact(Event e, vardetail, Node target) {
  InputElementamountInput = shadowRoot.querySelector("#amount");
  if (!checkAmount(amountInput.value)) return;
  bac.transact(amount);
  balance = bac.balance;
}

enter(KeyboardEvent e, vardetail, Node target) {
  if (e.keyCode == KeyCode.ENTER) {
    transact(e, detail, target);
  }
}

checkAmount(String in_amount) {
  try {
    amount = double.parseDouble(in_amount);
  } onFormatExceptioncatch(ex) {
    returnfalse;
  }
  returntrue;
}
}

```

How it works...

The actual numbers in step 1 for the polymer dependency may vary, but because the changes between versions can still be significant, it is better to use explicit versions rather than any other version here in order to not break your app; you can then also upgrade it when ready after thorough testing. The build.dart script in step 3 ensures that the project is built whenever a file in it is saved. The <link> tag in step 4 imports the Polymer element bank-app, which lives in bank_app.html. The platform.js and dart_support.js files contain the so-called platform polyfills; this is the JavaScript code to make new web standards such as custom elements, shadow DOM, template elements, and HTML imports work. They are called polyfills because at a later stage the browser should provide native code for these standards. The init.dart script starts up polymer.dart, and dart.js checks whether there is a Dart VM available to run the code directly. If this is not the case, the app runs from the compiled JavaScript code. Step 5 uses the name of the Polymer element as an HTML tag.

Step 6 comprises the HTML code for the Polymer component. It defines its style (in a `<style>` tag) and structure (within a `<template>` tag) between the `<polymer-element>` tags; its starting tag has the name of the component as an attribute, which is `name="bank-app"`. We see how one-way data binding is achieved with the `{} {}` syntax of the polymer expression, as in `<td>{{bac.number}}</td>` or `<td>{{bac.owner.name}}</td>`. The dots between the curly braces take the place of a variable from the code, whose value is shown in that place in the web page. In the code, these variables are annotated with `@published`. Interaction is achieved through declarative `on-event = "{} {}"` handlers, such as `on-click="{{transact}}"` in the button. Here, the dots stand for the name of a method in the code to be executed when the event happens. After `<template>`, the accompanying Dart script `bank_app.dart` is referenced through a `<script>` tag.

Finally, in step 7, we get to the Dart class that is backing up for our Polymer component and see the `BankApp` class extend `PolymerElement`; its class inherits from `PolymerElement`. The class declaration must be preceded by the annotation `@CustomTag('bank-app')` to associate it with the Polymer component. Variables that have to be visualized on the web page are annotated with `@observable`.

Override a custom element life cycle method such as `attached` to execute component-specific code. This method is executed when an element is inserted into the DOM; so it is a good place to initialize an app. It is obligatory to provide the created named constructor for your component. The constructor or any overridden method must call the superclass constructor or method first. The `transact` or `enter` method shows the signature for an event handler, `enter(KeyboardEvent e, var detail, Node target)`. The `transact` method shows how we can get the value for an input element in our Polymer component; use the `shadowRoot.querySelector` method. After the format of the input amount is checked, the `transact` method of the class `BankAccount` in the `bank_terminal` library changes the state of the `bac` object.

The name of a Polymer component must contain at least one dash (-). Notice the naming scheme; if the name of the Polymer component is `pol-comp1`:

- ▶ Then it is referenced in the HTML code as `< pol-comp1>`
- ▶ It is defined in the files `pol_comp1.html` and `pol_comp1.dart`
- ▶ Its class name is `Polcomp1`

There's more...

Polymer components can also be added dynamically via code, instead of being statically declared in the page's HTML. Suppose you want to add a component named `pol-comp1` in a `<div>` tag with an ID of `add-comp1`. You can do this with the following code:

```
querySelector('#add-comp1').children.add(new Element.tag('pol-comp1'));
```

In step 3, you could call a linter instead of an ordinary build by replacing this line, as shown in the following code:

```
main(args) {  
    lint(entryPoints: ['web/index.html'], options: parseOptions(args));  
}
```

This will display more extensive syntax or usage warnings in your code.

Binding and repeating over a list

In this recipe, we show you how the data of a list can be displayed in a Polymer component. So we perform data binding from the code to the UI here as well as in the following recipe. You can find the code for this recipe in the project `databinding_list`.

How to do it...

1. The script starts from `web\index.html`, where a component with the name `pol-list` is imported through the line:

```
<link rel="import" href="pol_list.html">
```

From this, we know that the component is defined in `pol_list.html`, and the code behind it is in a file named `pol_list.dart`. For a discussion of the other tags, see the previous recipe.

2. We define a list of companies that we want to display in the file `pol_list.dart`:

```
import 'dart:html';  
import 'package:polymer/polymer.dart';  
  
@CustomTag('pol-list')  
class Pollist extends PolymerElement {  
    final List companies = toObservable(['Google', 'Apple',  
    'Microsoft', 'Facebook']);  
  
    Pollist.created() : super.created() {  
        companies.add('HP');  
    }  
  
    addcompanies(Event e, var detail, Node target) {  
        companies.add('IBM');  
        companies.add('Dell');  
    }  
}
```

3. The structure of the component is outlined in `pol_list.html`:

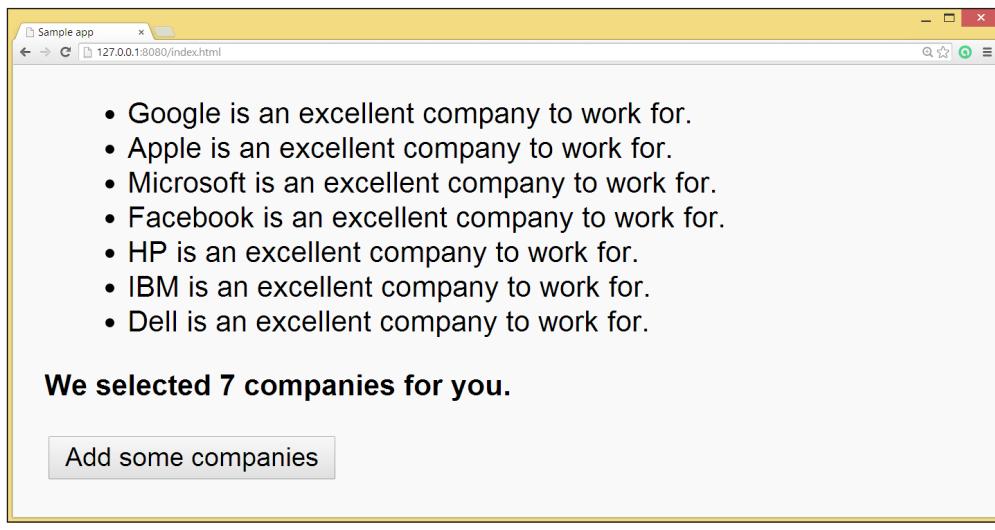
```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="pol-list">
<template>
<ul>
<template repeat="{{comp in companies}}>
<li> {{comp}} is an excellent company to work for.</li>
</template>
</ul>
<p><b>We selected {{companies.length}} companies for you. </b></p>
<button on-click="{{addCompanies}}>Add some companies</button>

</template>
<script type="application/dart"
src="pol_list.dart"></script>
</polymer-element>
```



To register the click event handler, you must use a dash in `on-click`; the `on-click` option will result in `ReferenceError`.

The following screenshot is what the web page displays when the app is run and the button is clicked:



Binding to a list

How it works...

We want to view the contents of the list on our page; this is made possible in step 2 by using the `toObservable` function from the `polymer` package. This function converts a literal `List` to an `ObservableList` (`@observable` doesn't work for a list or map). However, the list is changed in the created event, which is called when the component is instantiated. Also, `companies` is changed by pressing the button, which calls the method `addCompanies`.

Step 3 shows how a template loop is created. The `<template repeat="{{item in list}}> content </template>` syntax means that for every item in the list the content inside the template (here, the `` tag) is rendered.

From the running app, as shown in the previous screenshot, we see that whenever the observed list is changed, these changes are reflected in the web page. It also shows that the data binding syntax `{ { } }` can display properties of objects (here, `length` of the list).

There's more...

The same repeating template can be applied for every collection type, which is a type that implements `Iterable`.

Binding to a map

In this recipe, we show you how the data of a map can be displayed in a Polymer component. You can find the code in the project `databinding_map`.

How to do it...

1. The script starts with `web\index.html`, where a component with the name `pol-map` is imported through the following line:

```
<link rel="import" href="pol_map.html">
```

From this code, we know that the component is defined in `pol_map.html`, and the code behind it is in a file named `pol_map.dart`. For a discussion of the other tags, see the first recipe.

2. We define a map `companies`, which we want to display in the file `pol_map.dart`:

```
import 'dart:html';
import 'package:polymer/polymer.dart';
```

```
@CustomTag('pol-map')
class Polmap extends PolymerElement {
  Map companies = toObservable({1: 'Google', 2: 'Microsoft'});
```

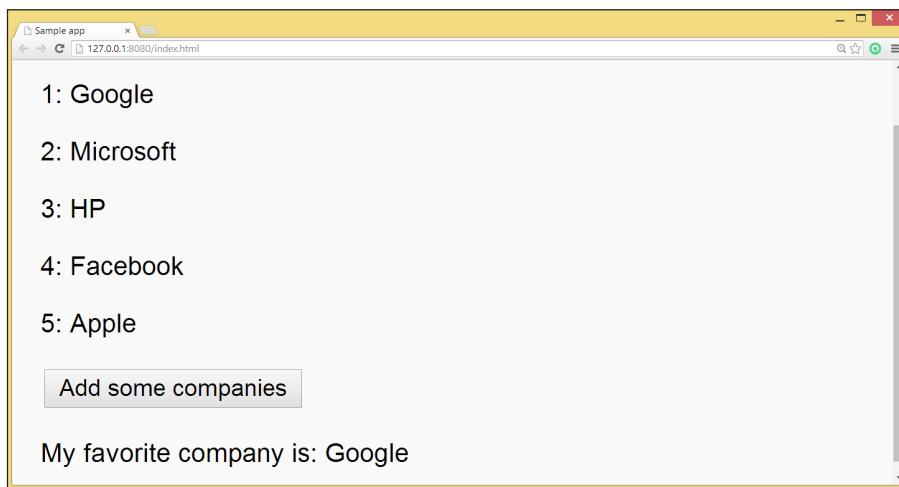
```
Polmap.created() : super.created() {
    companies[3] = 'HP';
}

addcompanies(Event e, var detail, Node target) {
    companies[4] = 'Facebook';
    companies[5] = 'Apple';
}
```

3. The structure of the Polymer component is outlined in pol_map.html:

```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="pol-map">
<template>
<p><b>The list of companies: </b></p>
<template repeat="{{key in companies.keys}}>
<p> {{key}}: {{companies[key]}}</p>
</template>
<button on-click="{{addcompanies}}>Add some companies</button>
<p>My favorite company is: {{companies[1]}}</p>
</template>
<script type="application/dart"
      src="pol_map.dart"></script>
</polymer-element>
```

The following screenshot is what the web page displays when the app is running and the button is clicked:



How it works...

We want to view the contents of `Map` in our page. This is made possible in step 2 by using the `toObservable` function from the `polymer` package. This function converts a literal map to an `Observable` map. However, `Mapcompanies` is changed in the `created` event, which is called when the component is instantiated. Also, `companies` is changed by pressing the button, which calls the method `addCompanies`.

Step 3 shows how a template loop is created using `<template repeat="{{key in companies.keys}}> content </template>`.

This means that for every key in the list `companies.keys`, the content inside the template (here, `<p>{{key}} : {{companies[key]}}</p>`) is rendered.

We could have also worked with `companies.values`, or any of the other map getters and methods such as `companies.length`. From the running app, as shown in the previous screenshot, we see that whenever the observed map is changed, these changes are reflected in the web page.

See also

- ▶ Refer to the *Binding and repeating over a list* recipe in this chapter on how to use the repeating template construct

Using custom attributes and template conditionals

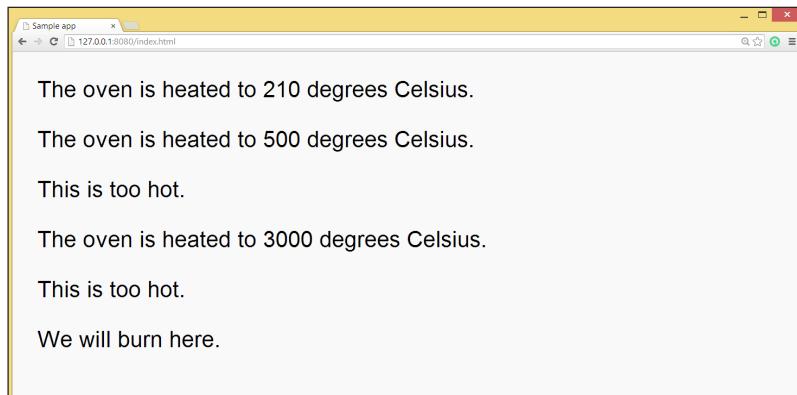
In this recipe, we will explore two `polymer.dart` features:

- ▶ **Custom attributes:** Like HTML attributes for normal tags, these are attributes for your Polymer component, and they can be changed in the code
- ▶ **Template conditionals:** The UI can be controlled by declarative conditions of the form `<template if="{{condition}}></template>`, where a condition is an expression involving observed or published variables

You can find the code in the project `custom_attrib`.

How to do it...

In this example, we simulate an oven, with the temperature as its attribute. We show the temperature in a textual form, and also test the temperature to display an appropriate message, as shown in the following screenshot:



Custom attributes and template conditionals

1. The script starts with `web\index.html`, where a component with the name `pol-oven` is imported through the following line:

```
<link rel="import" href="pol_oven.html">
```

This component is used three times on the page:

```

<body>
<pol-oven temperature="210"></pol-oven>
<pol-oven temperature="500"></pol-oven>
<pol-oven temperature="3000"></pol-oven>
</body>
  
```

From this code, we know that the component is defined in `pol_oven.html`, and the code behind it is in a file named `pol_oven.dart`. For a discussion of the other tags, see the first recipe.

2. The code for `pol-oven` is defined in `pol_oven.dart`:

```

import 'package:polymer/polymer.dart';

@CustomTag('pol-oven')
class Poloven extends PolymerElement {
  @published int temperature = 0;

  Poloven.created() : super.created() { }
}
  
```

3. The structure of the component is outlined in `pol_oven.html`:

```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="pol-oven">
<template>
  <p>The oven is heated to {{temperature}} degrees Celsius. </p>
  <template if="{{temperature > 400}}">
    <p>This is too hot. </p>
  </template>
  <template if="{{temperature > 1000}}">
    <p>We will burn here. </p>
  </template>
  </template>
<script type="application/dart" src="pol_oven.dart"></script>
</polymer-element>
```

How it works...

In step 1, we see that the same component, `pol-oven`, can be used multiple times on the same page; each occurrence being a different instance of the component. Of course, a combination of different components is also possible, each time the component contains its attribute `temperature` with a different value.

In step 2, it is shown that a custom attribute must be annotated with `@published`; this means that it is not merely displayed (and updated when changed, indicated with `@observable`), but is also an attribute to be used in the `polymer` tag.

Finally, in step 3, we see that the attribute values are displayed, but also that a piece of the UI code is displayed whether the evaluated condition is true or false. If the attribute value changes in the code, the conditional templates are automatically re-evaluated, possibly changing the UI.

See also

- ▶ Refer to the *Binding to a checkbox* recipe in this chapter for another example

Binding to an input text field or a text area

In this recipe, we will show you how to bind an input value from a text field or text area to a variable; so in effect, we now perform data binding from the UI to the code, and we have two-way data binding. You can find the code in the project `pol_text`.

How to do it...

1. The script starts with `web\index.html`, where a component with the name `pol_text` is imported through the following line:

```
<link rel="import" href="pol_text.html">
```

From this, we know that the component is defined in `pol_text.html`, and the code behind it is in a file named `pol_text.dart`. For a discussion of the other tags, see the first recipe.

2. The code for `pol_text` is defined in `pol_text.dart`:

```
import 'package:polymer/polymer.dart';

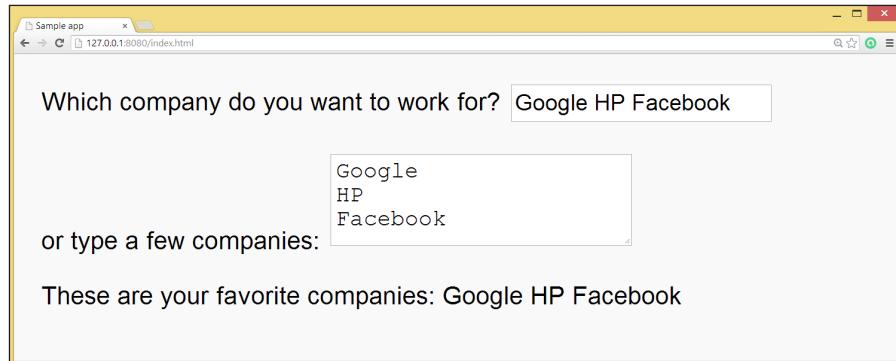
@CustomTag('pol-text')
class Poltext extends PolymerElement {
  @observable String comps;

  Poltext.created() : super.created() { }
}
```

3. The structure of the component is outlined in `pol_text.html`:

```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="pol-text">
<template>
<div>
  Which company do you want to work for?
  <input type="text" value="{{comps}}"/>
  <p> or type a few companies:
  <textarea value="{{comps}}"/></textarea>
</p>
</div>
<div>
  These are your favorite companies: {{comps}}
</div>
```

```
</div>
</template>
<script type="application/dart" src="pol_text.dart"></script>
</polymer-element>
```



Two-way binding to text fields

How it works...

The input (content) from the simple text field or text area is bounded by the expression `value = {{variable}}` to `variable`. It shows up in the simple `{{variable}}` expression on the page, but also immediately reflects in the other input field, which proves that we have two-way data binding.

Binding to a checkbox

In this recipe, we will show you how to bind an input value from a checkbox to a variable; so in effect, we now perform data binding from the UI to the code and have two-way data binding. You can find the code in the project `pol_check`.

How to do it...

1. The script starts with `web\index.html`, where a component with the name `pol-check` is imported through the following line:

```
<link rel="import" href="pol_check.html">
```

From this, we know that the component is defined in `pol_check.html`, and the code behind it is in a file named `pol_check.dart`. For a discussion of the other tags, refer to the first recipe.

2. The code for pol-check is defined in pol_check.dart:

```
import 'package:polymer/polymer.dart';
@CustomTag('pol-check')
class Polcheck extends PolymerElement {
  @observable bool receive = false;

  Polcheck.created() : super.created();
}
```

3. The structure of the component is outlined in pol_check.html:

```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="pol-check">
<template>
<div>
  Do you want to receive our jobs newsletter?
  <input type="checkbox" checked="{{receive}}"\>
</div>
<div>
  You will receive the newsletter: {{receive}}
  <p>Confirmed:</p>
  <template if="{{receive}}>You will receive the newsletter</template>
  <template if="{{!receive}}>You will not receive the newsletter</template>
</div>
</template>
<script type="application/dart" src="pol_check.dart"></script>
</polymer-element>
```

The following screenshot is what you see when you run the app:



Two-way binding to checkboxes

How it works...

In step 3, the value from the `checked` attribute of the checkbox is bound by the expression `checked="{{receive}}` to the variable `receive`. Through step 2, the value shows up in the simple `{{receive}}` expression on the page, but it is also used in template conditionals here.

See also

- ▶ Refer to the *Using custom attributes and template conditionals* recipe in this chapter for more information on template conditionals

Binding to radio buttons

In this recipe, we will show you how to bind an input value from a radio button to a variable; so in effect, we now perform data binding from the UI to the code and have two-way data binding. You can find the code in the project `pol_radio`.

How to do it...

1. The script starts with `web\index.html`, where a component with the name `pol-radio` is imported through the following line:

```
<link rel="import" href="pol_radio.html">
```

From this, we know that the component is defined in `pol_radio.html`, and the code behind it is in a file named `pol_radio.dart`. For a discussion of the other tags, refer to the first recipe.

2. The code for `pol-radio` is defined in `pol_radio.dart`:

```
import 'dart:html';
import 'package:polymer/polymer.dart';

@CustomTag('pol-radio')
class Polradio extends PolymerElement {
  @observable String favoriteJob = '';

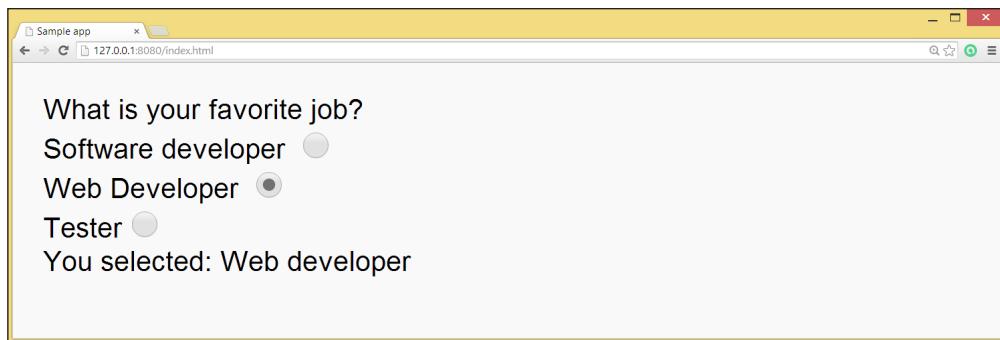
  Polradio.created() : super.created();

  void getFavoriteJob(Event e, var detail, Node target) {
    favoriteJob = (e.target as InputElement).value;
  }
}
```

3. The structure of the component is outlined in `pol_radio.html`:

```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="pol-radio">
<template>
<div on-change="{{getFavoriteJob}}>
What is your favorite job?
<div>
<label for="sd">Software developer <input name="job" type="radio"
id="sd" value="Software developer"></label>
</div>
<div>
<label for="wd">Web Developer <input name="job" type="radio"
id="wd" value="Web developer"></label>
</div>
<div>
<label for="tt">Tester<input name="job" type="radio" id="tt"
value="Tester"></label>
</div>
</div>
<div>
You selected: {{favoriteJob}}
</div>
</template>
<script type="application/dart" src="pol_radio.dart"></script>
</polymer-element>
```

The following screenshot is what you see when you run the app:



How it works...

On the web page (step 3), we see that the three radio buttons are grouped together by giving them the same value for the name attribute, which is name="job". Their values are literal strings, which they can deliver to code. This connection is made because the radio buttons are encapsulated in a <div> tag, which has a change event-handler defined in it:

```
<div on-change="{{getFavoriteJob}}">
```

The method `getFavoriteJob` in step 2 will extract the selected radio button's value, and assign it to the variable `favoriteJob`:

```
favoriteJob = (e.target as InputElement).value;
```

This variable is defined as `@observable`, so it shows up on the page with `{{favoriteJob}}`.

Binding to a selected field

In this recipe, we will show you how to bind a selected value from a <select> tag to a variable; so in effect, we now perform data binding from the UI to the code and have two-way data binding. We show a list of companies. You can find the code in the project `pol_select`.

How to do it...

1. The script starts with `web\index.html`, where a component with the name `pol-select` is imported through the following line:

```
<link rel="import" href="pol_select.html">
```

From this, we know that the component is defined in `pol_select.html`, and the code behind it is in a file named `pol_select.dart`. For a discussion of the other tags, see the first recipe.

2. The code for `pol-select` is defined in `pol_select.dart`:

```
import 'package:polymer/polymer.dart';

@CustomTag('pol-select')
class Poleselect extends PolymerElement {
  final List companies = toObservable(['Google', 'Apple',
  'Mozilla', 'Facebook']);
  @observable int selected = 2; // Make sure this is not null;
  // set it to the default selection index.
```

```

@observable String value = 'Mozilla';

    Polselect.created() : super.created();
}

```

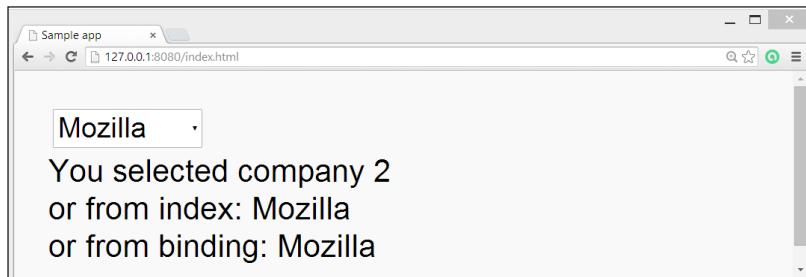
3. The structure of the component is outlined in pol_select.html:

```

<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="pol-select">
<template>
<select selectedIndex="{{selected}}" value="{{value}}>
<option template repeat="{{comp in companies}}> {{comp}} </
option>
</select>
<div>
You selected company {{selected}} <br/>
or from index: {{companies[selected] }} <br/>
or from binding: {{value}}
</div>
</template>
<script type="application/dart" src="pol_select.dart"></script>
</polymer-element>

```

The following screenshot is what you see when you run the app:



Binding with a select field

How it works...

Step 2 defines the list of companies to be shown. It also declares an index to be selected in List and value as a variable. Step 3 shows that to populate a `<select>` tag, the repeating template from the *Binding and repeating over a list* recipe in this chapter can be used. It also shows that the `selectedIndex` attribute is set from the code. When another company is selected, this also changes the values of `selected` and `value`.

Event handling

In this recipe, we will show you how to handle events in a Polymer component. You can find the code in the project `event_handling`.

How to do it...

1. The script starts with `web\index.html`, where a component with the name `pol-select` is imported through the following line:

```
<link rel="import" href="pol_events.html">
```

From this, we know that the component is defined in `pol_events.html`, and the code behind it is in a file named `pol_events.dart`. For a discussion of the other tags, refer to the first recipe.

2. The code for `pol-events` is defined in `pol_events.dart`:

```
import 'package:polymer/polymer.dart';

@CustomTag('pol-events')
class Polevents extends PolymerElement {
  @observable String which_event = "no event";
  @observable String thing = "";
  @observable String message = "";

  Polevents.created() : super.created();

  enter(KeyboardEvent e, var detail, Node target) {
    if (e.keyCode == KeyCode.ENTER) {
      which_event = "you pressed the ENTER key";
    }
  }

  btnclick(MouseEvent e, var detail, Node target) {
    message = (target as Element).attributes['data-msg'];
    which_event = "you clicked the button";
  }

  txtChange(Event e, var detail, Node target) {
    varinp = (target as InputElement).value;
    which_event = "you entered $inp in the text field";
  }

  cbClick(Event e, var detail, Node target) {
```

```

        which_event = "you changed the checkbox";
    }

    selChange(Event e, var detail, Node target) {
        which_event = "you selected another option";
    }
}

```

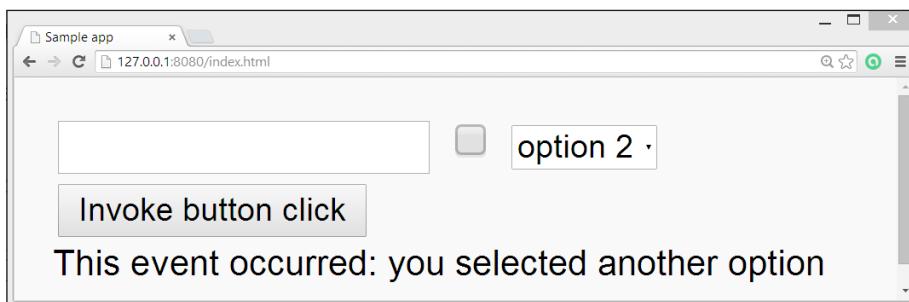
3. The structure of the component is outlined in `pol_events.html`:

```

<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="pol-events">
<template>
<div class="auto-style1" on-keypress="{{enter}}>
<input type="text" on-change="{{txtChange}}" value="{{thing}}>
<input type="checkbox" on-click="{{cbClick}}>
<select on-click="{{selChange}}>
<option>option 1</option>
<option>option 2</option>
<option>option 3</option>
</select>
<button on-click="{{btnClick}}" data-msg="Message from
button">Invoke button click</button>
</div>
<div> This event occurred: {{which_event}} </div>
<p>{{ message }}</p>
</template>
<script type="application/dart" src="pol_events.dart"></script>
</polymer-element>

```

Experiment with the different events. Each time you will see some text displaying which event was invoked, for example, when selecting from the drop-down list, you will see what's shown in the following screenshot:



Handling events

How it works...

In step 3, we see a number of event-handlers defined in different HTML elements. They all have the same form, which is `on-event={ {nameOfHandler} }`.

In Dart Editor and most plugins, you can find out which events are possible to handle in which tag by pressing `Ctrl + Space bar` within the HTML tag. Remember to put the dash between `on` and the event name; `onevent` throws a `ReferenceError` error in runtime.

In step 2, we see a number of different handlers; they all have the same signature:

```
nameOfHandler(Event e, var detail, Node target) {  
    // code to be executed when event occurs  
}
```

The `target` node is the HTLM node upon which the event occurred. If this was `InputElement`, you can get the input value with `varinp = (target as InputElement).value;;` notice how the `data-msg` attribute from the button is read out with `message = (target as Element).attributes['data-msg'];`.

Polymer elements with JavaScript interop

In this recipe, we will show you how to work with a JavaScript object in a Polymer component. You can find the code in the project `pol_js`.

How to do it...

1. The script starts with `web\index.html`, where a component with the name `pol_js` is imported through `<link rel="import" href="pol_js.html">` and instantiated through `<pol-js></pol-js>`.

From this, we know that the component is defined in `pol_js.html`, and its code is in a file named `pol_js.dart`. For a discussion of the other tags, refer to the first recipe.

2. The `index.html` file also includes a JavaScript `person.js` object:

```
function Person(name, gender) {  
    this.name = name;  
    this.gender = gender;  
    this.greeting = function(otherPerson) {  
        alert('I greet you ' + otherPerson.name);  
    };  
}  
  
Person.prototype.sayHello = function(times) {
```

```
    return times + ' x: hello, I am ' + this.name;  
};
```

3. The structure of the component is outlined in `pol_js.html`:

```
<link rel="import" href="packages/polymer/polymer.html">  
<polymer-element name="pol-js">  
  <template>  
    <p>From JS interop: {{result}}</p>  
    <p>  
      <button on-click="{{btnClick}}>Click me</button>  
    </p>  
  </template><script type="application/dart"  
src="pol_js.dart"></script>  
</polymer-element>
```

4. The code for `pol-js` is defined in `pol_js.dart`:

```
import 'dart:html';  
import 'dart:js';  
import 'package:polymer/polymer.dart';  
  
@CustomTag('pol-js')  
class PolJs extends PolymerElement {  
  @observable String result;  
  
  PolJs.created() : super.created();  
  
  btnClick(Event e, var detail, Node target) {  
    var pers1 = new JsObject(context['Person'], ['An', 'female']);  
    result = pers1.callMethod('sayHello', [10]);  
  }  
}
```

You will get what is shown in the following screenshot after you click on the **Click me** button:



Polymer interop with JavaScript

How it works...

In the JavaScript from step 2 (which is known to the web page), a class `Person` with a method `sayHello` is defined. The parameter `times` says how many times the greeting is to be repeated. In step 3, a variable `result` is bound to be displayed, and an event `btnClick` is defined in the button. Step 4 shows us the code that does this; an object `pers1` is created, and the method `sayHello` is called upon it and assigned to `result`. Notice that we had to import '`dart:js`' ; to make Dart-JavaScript interaction possible.

See also

- ▶ To learn more about the Dart and JavaScript interaction, refer to the *Talking with JavaScript* recipe in *Chapter 5, Handling Web Applications*

Extending DOM elements

Instead of making a new Polymer component from scratch, as we did in the previous recipes, you can also start from a native HTML element and build upon that. This is made possible because our component is backed up by a class that can inherit the properties and behavior of an existing HTML element class. For our example, we will extend a `Div` element. You can find the code in the project `dom_extend`.

How to do it...

1. The script starts with `web\index.html`, where a component with the name `dom-extend` is imported through the following line:

```
<link rel="import" href="dom_extend.html">
```

From this, we know that the component is defined in `dom_extend.html`, and the code behind it is in a file named `dom_extend.dart`. For a discussion of the other tags, refer to the first recipe. Because we make a specialized `<div>` tag, we have to indicate this with an `is` attribute, as follows:

```
<body>
<div is="dom-extend">Initial div content </div>
</body>
```

2. The code for `dom-extend` is defined in `dom_extend.dart`:

```
import 'dart:html';
import 'package:polymer/polymer.dart';

@CustomTag('dom-extend')
class DomExtend extends DivElement with Observable {
```

```
Domextend.created() : super.created() {
  polymerCreated();
  text = "I am not an ordinary div!";
}
```

3. The structure of the component is outlined in `dom_extend.html`:

```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="dom-extend" extends="div">
<template>
<style>
:host {
  background: lime;
  color: red;
  font-size: 12px;
  font-weight: bold, italic;
  border: 1px solid #ccc;
}
</style>
<content>Initial dom-extend content</content>
</template>
<script type="application/dart" src="dom_extend.dart"></script>
</polymer-element>
```

We will get the following output after running this app:



Extending a DOM element

How it works...

In step 1, you could read the `is="dom-extend"` attribute meaning, this is `<div>`, but it inherits the style, structure, and behavior of the `dom-extend` Polymer component. Step 2 shows us that class `Domextend` extends `DivElement` with `Polymer`, observable that `Domextend` itself inherits from class `DivElement` and mixes with the `Polymer` and `Observable` classes. Compare this with what we have for a normal Polymer component (from the *Binding and repeating over a list* recipe in this chapter), which only inherits from the `PolymerElement` class; class `Pollist` extends `PolymerElement`.

The `Polymer` class is the mixing class for Polymer elements; it provides utility features on top of the custom web elements standard. If it is used in this way, you must call `polymerCreated()` from within the constructor. The constructor also changes the text in the `<div>` tag.

In step 3, we have a second but necessary statement, `<polymer-element name="dom-extend" extends="div">` with `extends="div"`, which `Domextend` inherits from `DivElement` in full.

We have also used the `host` selector in order to style the shadow DOM `<style>:host { ... } </style>`. This style can be overridden by the embedding web page. If this is not what you want, just use the normal style selectors, such as `class`, `ID`, and so on. The `<content></content>` area is used for content insertion; when a component has children, those children go where the `<content>` tags are. Here, the initial text is overwritten by the Polymer component `dom-extend`.

Working with custom elements

Instead of making a new Polymer component from scratch or starting with an existing HTML element and building upon that as we did in the previous recipe, you can also simply use custom-made Polymer elements.

This recipe will implement some of the core and paper elements of the Polymer project (<http://www.polymer-project.org/docs/elements/>). The `paper_elements` project is the Polymer implementation of Google's Material Design UI widgets (for more information, refer to <http://www.polymer-project.org/docs/elements/material.html>). There will be more and more of these, either written in JavaScript with a custom Dart wrapper to use them, or purely in Dart, and you can also combine them with your own Polymer components. You can find the code in the project `pol_custom`.

How to do it...

1. In our pubspec.yaml file, we add the following dependencies:

```
dependencies:  
  polymer: '>=0.11.0 <0.12.0'  
  core_elements: '>=0.0.6 <0.1.0'  
  paper_elements: '>=0.0.1 <0.1.0'
```

2. The script starts with web\index.html, where our core and paper components are imported through the following lines:

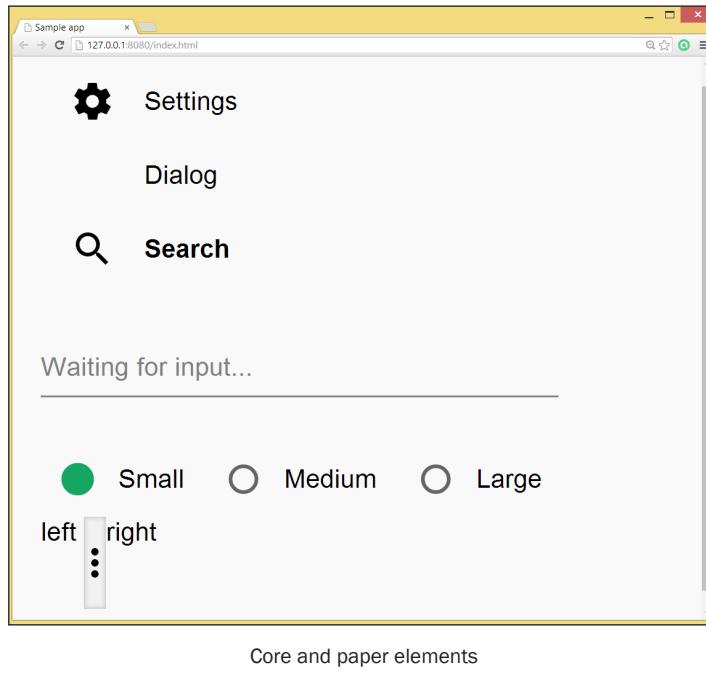
```
<link rel="import" href="packages/core_elements/core_icon.html">  
<link rel="import" href="packages/core_elements/core_icons.html">  
<link rel="import" href="packages/core_elements/core_menu.html">  
<link rel="import" href="packages/core_elements/core_item.html">  
<link rel="import" href="packages/paper_elements/paper_input.html">  
<link rel="import" href="packages/paper_elements/paper_radio_group.  
html">  
<link rel="import" href="packages/core_elements/core_splitter.  
html">
```

We randomly chose some components from the several dozen available.

The components are instantiated in the <body> tag:

```
<body unresolved>  
  <core-menu selected="0">  
    <core-item icon="settings" label="Settings"></core-item>  
    <core-item icon="dialog" label="Dialog"></core-item>  
    <core-item icon="search" label="Search"></core-item>  
  </core-menu>  
  <paper-input label="Waiting for input..."></paper-input>  
  <paper-radio-group selected="small">  
    <paper-radio-button name="small" label="Small"></paper- radio-  
    button>  
    <paper-radio-button name="medium" label="Medium"></paper-radio-  
    button>  
    <paper-radio-button name="large" label="Large"></paper-radio-  
    button>  
  </paper-radio-group>  
  <div horizontal layout>  
    <div>left <br/><br/><br/></div>  
    <core-splitter direction="left"></core-splitter>  
    <div flex>right</div>  
  </div>  
</body>
```

Running this app will give you the following output:



Core and paper elements

How it works...

The `core_elements` and `paper_elements` pub packages are wrappers and ports for Polymer's element collections with the same name. They package the elements into single pub packages to be able to add them as a `pubspec` dependency. Most core elements are wrapped with Dart proxy classes to make them easier to interact with Dart scripts.

In step 2, a random selection of these elements is imported and instantiated in the page. Most properties are changeable via attributes such as icons, labels, and so on. The `<body>` tag needs the `unresolved` attribute to ensure that no Polymer custom elements are displayed before Polymer is ready. Since these are custom elements, there are no steps 3 and 4 as in the previous recipes!

There's more...

The `Index.html` file also shows how to use the core and paper icons; import them with the following code:

```
<link rel="import" href="packages/paper_elements/paper_icon_button.html">
<link rel="import" href="packages/core_elements/src/core-icons/iconsets/social-icons.html">
```

In the preceding code, you choose the name according to the set you want to load, for example, social icons. Then, use the icon by setting the `icon` attribute; the value consists of icons set, -ID, a colon followed by the icon name. This icon for example, shows a +1 value:

```
<paper-icon-button id="bookmark-button" icon="social:plus-one" style="fill:steelblue;"></paper-icon-button>
```

Automatic node finding

Like jQuery with its `$` function, Polymer also has a very handy way to locate nodes in the DOM of the page. This recipe shows you how to use it. You can find the code in the project `find_nodes`.

How to do it...

1. The script starts with `web\index.html`, where a component with the name `find_nodes` is imported through the following line:

```
<link rel="import" href="find_nodes.html">
```

From this, we know that the component is defined in `find_nodes.html`, and the code behind it is in a file named `find_nodes.dart`. For a discussion of the other tags, refer to the first recipe.

2. The code for `find-nodes` is defined in `find_nodes.dart`:

```
import 'dart:html';
import 'package:polymer/polymer.dart';

@CustomTag('find-nodes')
class Findnodes extends PolymerElement {
  Findnodes.created() : super.created();

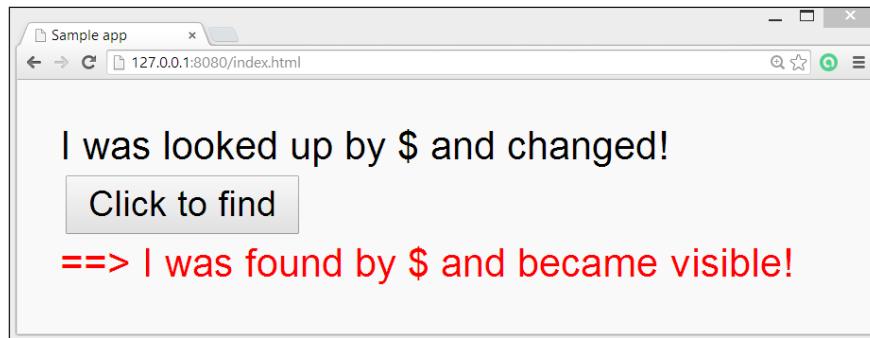
  btnclick(MouseEvent e, var detail, Node target) {
    // making the paragraph visible:
```

```
$['show'].style
..display = 'inline'
..color = 'red';
// changing the text inside the div:
Element insideDiv = $['findme'];
insideDiv.text = 'I was looked up by \$ and changed!';
}
}
```

3. The structure of the component is outlined in `find_nodes.html`:

```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="find-nodes">
<template>
<div class="auto-style1">
<div id="findme">Hello from inside a div</div>
<button on-click="{{btnclick}}>Click to find</button>
</div>
<p id="show" style="display:none">>=> I was found by $ and
became visible! </p>
</template>
<script type="application/dart"
src="find_nodes.dart"></script>
</polymer-element>
```

The following screenshot shows you what you will see when you run the app:



Binding with a select field

How it works...

In step 2, we see that the paragraph with ID `show` is shown when the button is clicked because of the line `$['show'].style.display = 'inline';` in the event handler. Likewise, an element reference to `<div>` with the ID `findme` is found with `$['findme']`. The code uses automatic node finding, a Polymer feature, to get a reference to each HTML element. Every node in a custom element (so inside the shadow DOM!) that is tagged with an `id` attribute can be referenced by its ID using the syntax `$['ID']`.



This example requires Polymer dart 0.8.0 or higher.



Internationalizing a Polymer app

What if you want your web app to display information in several languages, depending on the language of the web client user? This recipe will show you how to accomplish this. You can find the code in the project `pol_intl`.

How to do it...

1. Add the `intl` package from the `pub` package to your app through `pubspec.yaml`.
2. The script starts with `web\pol_intl.html`, where a component with the name `localized_text` is imported through the following line:

```
<link rel="import" href="localized_text.html">
```

From this, we know that the component is defined in `localized_text.html`, and the code behind it is in a file named `localized_text.dart`. For a discussion of the other tags, see the first recipe.

3. The structure of the component is outlined in `localized_text.html`:

```
<link rel="import" href="packages/polymer/polymer.html">
<polymer-element name="localized-text">
<template>
<p>{startMsg}</p>

<select value="{{selectedLocale}}>
<option value="en_US">English</option>
<option value="fr">French</option>
<option value="nl_NL">Dutch</option>
<option value="de">German</option>
<option value="it">Italian</option>
```

```
<option value="jp">Japanese</option>
</select>
```

```
</template>
<script type="application/dart"
src="localized_text.dart"></script>
</polymer-element>
```

4. The code for localized-text is defined in localized_text.dart:

```
import 'package:polymer/polymer.dart';
import 'package:intl/intl.dart';
import 'messages_all.dart';

@CustomTag('localized-text')
class LocalizedText extends PolymerElement {
  @observable String selectedLocale;
  @observable String startMsg;

  LocalizedText.created() : super.created() {
    updateLocale(Intl.defaultLocale);
  }

  void selectedLocaleChanged() {
    initializeMessages(selectedLocale).then(
      (succeeded) => updateLocale(selectedLocale));
  }

  void updateLocale(localeName) {
    Intl.defaultLocale = selectedLocale;
    startMsg = start();
  }

  start() => Intl.message("Please choose your language and then start
the tour.",
  name: 'startMsg',
  desc: "Starting the tour",
  args: [],
  examples: {"": 0});
}
```

The following screenshot shows what you will see when you run the app:



Using intl with Polymer

How it works...

The package `intl` is imported in step 1; it is maintained by the Dart team, which provides internationalization and localization facilities, for example, message translation, plurals and genders, date/number formatting and parsing, and bidirectional text. In step 3, a drop-down list of languages is offered to choose a language from. The value of the chosen option is bound to the variable `selectLocale`. The purpose of the rest of the code is to translate the content of `startMsg` to this language.

In step 4, the method `selectedLocaleChanged()` is triggered when a language is chosen. This calls the `updateLocale` method with `selectedLocale`, which sets the value as `defaultLocale` and sets `startMsg` to the result of the `start()` function. This executes `Intl.message` for the message we want to translate, returning that message in the chosen language. All messages to be localized are written as functions that return the result of an `Intl.message` call.

Internally, `intl` works as follows:

- ▶ The messages to be translated are stored in a file named `intl_messages.json`, with their names and the message in the default language. They are extracted from the program source, either manually or by running `dart extract_json.dartlocalized_text.dart`, which produces the JSON file.
- ▶ From this file, the different `translation_locale.json` files are produced; one for each language to be used. These files contain a map with the locale, and its name and translation for each message. For example, here is the Italian version `translation_it.json`:

```
{"_locale" : "it",
"startMsg" : "Scegliere la lingua e poi iniziare il tour."}
```

Now, run `dart generate_from_json.dartlocalized_text.darttranslation_fr.jsontranslation_nl.json`. This will generate `messages_all.dart`, `messages_fr.dart`, `messages_nl.dart`, and so on.

- ▶ Now import `messages_all.dart` in the main Dart script, and the mechanism is in place. If the number of messages is small, you could work with the `messages_` files themselves, without having to generate them from JSON files.

There's more...

For more detailed information on `intl`, see its documentation at <http://www.dartdocs.org/documentation/intl/0.11.3/index.html#intl>.

If instead of letting your user choose the language, you want to derive it from the information the browser gives you, then use the following code:

```
import "package:intl/intl_browser.dart";  
...  
findSystemLocale().then(runTheRestOfTheProgram);
```

11

Working with Angular Dart

This chapter contains the following recipes:

- ▶ Setting up an angular app
- ▶ Using a controller
- ▶ Using a component
- ▶ Using formatters as filters
- ▶ Creating a view
- ▶ Using a service
- ▶ Deploying your app

Introduction

In this chapter, we are going to use Angular Dart to build client web applications. Angular Dart (<https://github.com/angular/angular.dart>) is the porting of AngularJS to Dart. AngularJS, or Angular for short (refer to www.angularjs.org), is a popular open source JavaScript framework, maintained by Google, to develop single-page dynamic web applications. Its goal is to create web-based apps with **Model-View-Controller (MVC)** or **Model-View-ViewModel (MVVM)** capabilities in an effort to make both development and testing easier.

It accomplishes this using declarative programming to build UI and wire software components so that you can concentrate on your application's logic and not on DOM manipulation. It uses a templating system with a number of so-called directives (starting with `ng-`) to specify customizable and reusable HTML tags and expressions that control the behavior of certain elements: in effect, you extend HTML with custom elements and attributes. Google uses Angular Dart to build internal applications. Each recipe explored in this chapter exposes a major component of Angular. The most up-to-date documents can be found at <https://docs.angular-dart.org/>.

Setting up an Angular app

This recipe is a preliminary step necessary for every other recipe in this chapter. It shows you how to make the Angular Dart functionality available to your app.

How to do it...

1. There is no `angular` template yet in Dart Editor, so start your app from a web application (mobile friendly), and call it, for example, `angular_setup`. Clear the sample code from the `html` and `dart` files.
2. Add `angular` to the dependencies in `pubspec.yaml`. Saving will start the `pub get` procedure.
3. Also add `js` and `shadow_dom` to `pubspec.yaml`.
4. Add the shadow DOM script to the `html` file:

```
<script
  src="packages/shadow_dom/shadow_dom.min.js"></script>
```
5. Also, include the Angular transformer:

```
transformers:
- angular
```
6. Provide the `ng-app` attribute in the `<html>` element.
7. Add the following statement to the top of your main Dart script:

```
import 'package:angular/angular.dart';
import 'package:angular/application_factory.dart';
```
8. In `main()`, insert the code `applicationFactory().run();`.
9. Provide the `ng-cloak` attribute in the `<body>` element.
10. Add the following section to your CSS file:

```
[ng-cloak], .ng-cloak {
  display: none !important;
}
```

How it works...

Step 2 downloads the basic Angular Dart framework as well as the packages it depends on. Notice that it uses the `web_components` package. Steps 3 and 4 turn on the Shadow DOM for older browsers that do not yet implement this feature natively, in which case, the package `js` is needed for JavaScript interoperability.

Shadow DOM is the ability of the browser to include a sub-tree of DOM elements into the rendering of a tag, the so-called shadow root. For example, an `<input type="date">` element hides a whole HTML table to create a slick calendar that highlights the range of dates and that reacts to click events. It is on this basis that web components are built. For more detailed information, refer to <http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/>.

Step 5 is necessary to deploy your app when you convert your app to JavaScript (refer to the *Deploying your app* recipe). The `ng-app` directive in Step 6 tells Angular which element is the root element of the application: everything inside of it is part of the page template managed by Angular. In most cases, this is the outermost `<html>` tag; anything inside of this element is part of the page template managed by Angular. After the page is loaded, Angular looks for the `ng-app` directive. Upon finding it, it bootstraps the application, with the root of the application DOM being the element on which the `ng-app` directive was defined.

Step 7 makes Angular available to your code, and step 8 starts Angular's event-loop to handle every browser event. If anything in the model changes in this event handling, all corresponding bindings in the view are updated.

Steps 8 and 9 are necessary to avoid the display of `{ ... }` before the correct values from the model are inserted. This happens because there is a little time gap between the time when you load HTML and the time when Angular is ready with bootstrapping, compiling the DOM, and substituting in the real values for the data binding expressions.

There's more...

If you need the `mirrors` library, `dart:mirrors`, to use Dart's reflection capabilities, provide a temporary fix using the `mirrors` annotation:

```
@MirrorsUsed(override: '*')
import 'dart:mirrors';
```

The processes of minifying and tree-shaking your app performed by the `dart2js` compiler will generally not detect reflected code so that the use of reflection at runtime might fail, resulting in `NoSuchMethod()` errors. To prevent this from happening, use the `mirrors` annotation, which helps the `dart2js` compiler to generate smaller code.

For more information on mirrors, see the *Using reflection* recipe in *Chapter 4, Object Orientation*, and also <https://www.dartlang.org/articles/reflection-with-mirrors/>.

Using a controller

The web page is our view in the MVC pattern. The controller object is responsible for showing data from the model (binding them to DOM elements in the view) and in response to events, possibly changing model data and displaying these changes in the view. All public fields of the controller can be shown, and all public methods can be invoked from within the view. Data binding is done through the same syntax as in Polymer using double curly braces `{ { ... } }`. A controller should contain only the business logic needed for a single app or view; it should not manipulate the DOM.

In this recipe, we'll show you how to work with an Angular controller step by step. You can follow along with the code in the project `angular_controller`.

How to do it...

Our app will show job type data in a list, and when a job is selected, its details are shown, as shown in the following screenshot:



The controller in action

Its working is explained as follows:

- ▶ The Job class, which is the model, is defined in `angular_controller.dart` as follows:

```
class Job {  
    String type;  
    int salary;  
    String company;  
    DateTime posted; // date of publication of job  
    bool open = true; // is job still vacant?  
    List<String> skills;  
    String info;  
    Job(this.type, this.salary, this.company, this.posted, this.  
skills, this.info);  
}
```

- ▶ The controller is a separate class, `JobListingController`, in the same file:

```
@Controller(  
    selector: '[job-listing]',  
    publishAs: 'ctrl')  
class JobListingController {  
    Job selectedJob;  
    List<Job> jobs;  
  
    JobListingController() {  
        jobs = _loadData();  
    }  
  
    void selectJob(Job job) {  
        selectedJob = job;  
    }  
  
    // model data:  
    List<Job> _loadData() {  
        return [  
            new Job('Web Developer', 7500, 'Google', DateTime.  
parse('2014-07-03'),  
            ["HTML5", "CSS", "Dart"], "on-site job Palo Alto, California"),  
            // other job data  
        ];  
    }  
}
```

- ▶ In the web page, everything happens in `<div>` marked with `job-listing`:

```
<div job-listing>
    <h3>Job List</h3>
    <ul>
        <li class="pointer"
            ng-repeat="job in ctrl.jobs"
            ng-click="ctrl.selectJob(job) ">
            {{job.type}}
        </li>
    </ul>
    <h3>Job Details</h3>
    <div><strong>Type: </strong>{{ctrl.selectedJob.type}}</div>
    <div><strong>Company: </strong>{{ctrl.selectedJob.company}}</div>
    <div><strong>Salary: </strong>{{ctrl.selectedJob.salary}}</div>
    <div><strong>Posted: </strong>{{ctrl.selectedJob.posted}}</div>
    <div><strong>Skills: </strong>
        <ul>
            <li ng-repeat="skill in ctrl.selectedJob.skills">
                {{skill}}
            </li>
        </ul>
    </div>
    <div><strong>More info: </strong>{{ctrl.selectedJob.info}}</div>
</div>
</div>
```

- ▶ To start up the Angular machinery, something more has to be done now as shown in the following code:

```
void main() {
    applicationFactory()
        .addModule(new AppModule())
        .run();
}

class AppModule extends Module {
    AppModule() {
        bind(JobListingController);
    }
}
```

How it works...

The controller in step 2 contains the list of jobs to be shown; it loads them in through `_loadData()` in its constructor. It also holds the selected job, if any. This controller is marked with the `@` annotation. Have a look at the following code:

```
@Controller(  
    selector: '[job-listing]',  
    publishAs: 'ctrl')
```

The string after `selector`, between brackets, is the name of a CSS selector (here, `job-listing`) in the page. In step 3, we see that there is a `<div>` element that has this name as selector. This `<div>` element defines a scope in which the controller is active and known. When Angular sees this `<div>` element, it instantiates the controller class, making its content available. The `publishAs` code gives the name (here, `ctrl`) that the controller is known by in the `<div>` scope in the view: that's why we see that name appear throughout the HTML code.

In ``, all the jobs from the list in the controller known as `ctrl` are shown through the directive `ng-repeat="job in ctrl.jobs"`; this iterates over the model (the `job` property in `JobListingController`) and the clone `` in the compiled DOM for each `job` in the list. More specifically, only `type` is shown because the `` tag contains `{job.type}`.

The same tag also contains an `ng-click` directive, which registers an event-handler for a click event on the list item `ng-click="ctrl.selectJob(job)"`. This can be attached to any HTML element, and here it calls the method `selectJob` in the controller, passing the job that was clicked. In step 2, we see that this method passes this `job` to `selectedJob`. Because this variable now gets a value, the view updates, and all `{ctrl.selectJob()}` expressions are evaluated and shown, including the list of skills, where an `ng-repeat="skill in ctrl.selectedJob.skills"` directive is used.

To make this work in Angular, we need to wrap our controller in the class `AppModule`, which inherits from `Module`. To instantiate this new module, it has to be added to the Angular engine via the method `addModule()`, a dependency injection technique that is used for other Angular items too as we will see in the next recipes; in general, an Angular app will have a list of modules with which it works.

There's more...

A control in the view can be disabled when a certain condition is met if you add the following HTML attribute: `ng-disabled="ctrl.condition"`, where `condition` is a Boolean property or function in the controller. To make the control visible or not, use `ng-show` or `ng-hide`.

Using a component

In this recipe, we'll show you how to work with an Angular component step by step. Components are lightweight, reusable, and self-contained UI widgets that have a single specific purpose. We'll use a component that shows a graphical representation of the job's salary. You can follow along with the code in the project `angular_component`.

How to do it...

Our app shows job type data in a list, together with a number of stars, to indicate the salary. This is also shown when selecting a job to show its details, as shown in the following screenshot:



A component showing the salary

Its working is explained as follows:

- ▶ The following is the startup script `angular_component.dart`:

```
import 'package:angular/angular.dart';
import 'package:angular/application_factory.dart';
import 'package:angular_component/salary/salary_component.dart';
import 'package:angular_component/job_listing.dart';

void main() {
    applicationFactory()
        .addModule(new AppModule())
        .run();
}

class AppModule extends Module {
    AppModule() {
        bind(JobListingController);
        bind(SalaryComponent);
    }
}
```

- ▶ In the web page, we see a new HTML element `<x-salary>` for our component, as follows:

```
<x-salary max-sal="10" salary="job.rate_salary"></x-salary>
```

This is also found in the job details section, which is now surrounded by the following code:

```
<div ng-if="ctrl.selectedJob != null">
    ...
<x-salary max-sal="10" salary="ctrl.selectedJob.rate_salary"></x-
salary>
    ...
</div>
```

The component is defined in the folder `lib\salary`, with an HTML file `angular_component.html` that defines its structure, a Dart file that describes its behavior, and a CSS file to style it.

- ▶ The following is the Dart code:

```
import 'package:angular/angular.dart';
```

```
@Component(
    selector: 'x-salary',
```

```
templateUrl: 'packages/angular_component/salary/salary_
component.html',
cssUrl: 'packages/angular_component/salary/salary_component.
css',
publishAs: 'cmp')
class SalaryComponent {
    static const String _STAR_ON_CHAR = "\u2605";
    static const String _STAR_OFF_CHAR = "\u2606";
    static const String _STAR_ON_CLASS = "star-on";
    static const String _STAR_OFF_CLASS = "star-off";

    static final int DEFAULT_MAX = 5;

    List<int> stars = [];

    @NgOneWay('salary')
    int salary;

    @NgAttr('max-sal')
    void set maxSal(String value) {
        var count = (value == null)
            ? DEFAULT_MAX
            : int.parse(value, onError: (_)>> DEFAULT_MAX);
        stars = new List.generate(count, (i)>> i + 1);
    }

    String starClass(int star) =>
        star > salary ? _STAR_OFF_CLASS : _STAR_ON_CLASS;

    String starChar(int star) => star > salary ? _STAR_OFF_CHAR :
    _STAR_ON_CHAR;
}
```

- ▶ The following is the HTML code for the component:

```
<span class="stars"
    ng-repeat="star in cmp.stars"
    ng-class="cmp.starClass(star)">
    {{cmp.starChar(star)}}
</span>
```

How it works...

In step 1, you can see that we refactored the code of `JobListingController` into its own library in `lib\job_listing.dart`. A component called `SalaryComponent` is now also bound to `AppModule`. Step 2 demonstrates how the component is used in HTML as a new kind of tag `<x-salary>`, with properties `max-sal` and `salary`. Why `x-`? In order to be W3C compliant, custom components should have a dash (-) in their names.

Additionally, we see how we control the inclusion of HTML sections in Angular with `ng-if="condition"`. If the condition is false, then that `<div>` section is removed from the DOM.

In step 3, we see how the definition of the class `SalaryComponent` is preceded by the `@Component` annotation. Its `selector` part states the HTML element that will instantiate the component, which also defines its scope. Note that `templateUrl` and `cssUrl` refer to the definition of the component, and `publishAs` is the name for the component in its scope, which is used in step 4. How does the component depict the salary? The star character comes from the Unicode "\u2605", shown through `{ { cmp.starChar(star) } }`. When the component is instantiated, its properties get set, in particular, `salary="job.rate_salary"`. `rate_salary` is a getter in the class `Job`: the properties are set as `int get rate_salary => salary~/1000`. Thus, the salary in the component is the number of thousands in the job's salary. Note that `salary` is annotated by `@NgOneWay`, which means that it is a one-way property and its value flows from the object to the UI, but the UI cannot change it in the code. Note that `max-sal`, being an attribute, is annotated by `@NgAttr`. The `NgAttr` annotation on a field maps this to a DOM attribute. The `stars` list gets set to a list of `max-sal` numbers by `List.generate(): [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. The `ng-class` directive sets the CSS class on the component dynamically; its expression is the name of the class to be added to it, which amounts to "star-on" or "star-off". The star is pictured yellow when `star` is still smaller than or equal to `salary`. Thus, for every thousand dollars in `salary`, a yellow star is shown by `ng-repeat`, while `max-sal` determines the number of uncolored stars.

There's more...

If a user needs to be able to change a property through the web page (for example, changing the salary), they will have to declare it as `@TwoWay` in the code. In our example, this can be done by defining an `ng-click="event-handler"` for our component.

See also

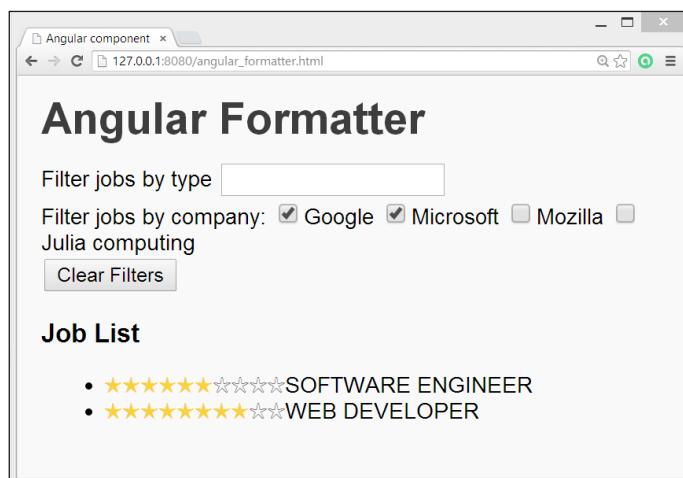
- ▶ For another example of a component, see the *Creating a view* recipe in this chapter

Using formatters as filters

Formatters are helper tools to view your data differently from how they are stored in the model. Angular has built-in formatters, for example, `Date` to format date-times, `Currency` to format money data, and `LimitTo` to limit the view to a certain number of results. The `Filter` class displays items based on whether they satisfy the criteria set up in the filter. Sorting works through an `orderBy` attribute in `ng-repeat`. In this recipe, we will show you how to use filters to make different views on your data possible. You can follow along with the code in the project `angular_formatter`.

How to do it...

The job listing is now preceded by an input field; when you start typing the job type, the list of only those jobs that start with these letters are shown. The checkboxes allow you to filter on company, and the type of job is shown in uppercase in the job details section, as shown in the following screenshot:



Using formatters to limit the view

Perform the following steps to use formatters as filters:

1. We've added two filters: the first on the job type, the second on the company. They are expressed through HTML in `angular_formatter.html`. Have a look at the following code:

```
<div id="filters">
<div>
  <label for="type-filter">Filter jobs by type</label>
  <input id="type-filter" type="text">
```

```

    ng-model="ctrl.typeFilterString">
  </div>
  <div> Filter jobs by company:
    <span ng-repeat="company in ctrl.companies">
      <label>
        <input type="checkbox"
          ng-model="ctrl.companyFilterMap[company]">{{company}}
      </label>
    </span>
  </div>
  <input type="button" value="Clear Filters" ng-click="ctrl.
clearFilters()">
</div>

```

2. The actual filtering takes place in the same file; instead of a simple `ng-repeat = "job in ctrl.jobs"`, we now have the following:

```

ng-repeat="job in ctrl.jobs
orderBy:'type'
filter:{type:ctrl.typeFilterString}
companyfilter:ctrl.companyFilterMap"

```

We have shown here the different filter parts on consecutive lines for readability, but in HTML, they must be on one line.

In the controller code (`lib\job_listing.dart`), we now have two lists:

```

List<Job> jobs;
List companies = [];

```

The `_loadData()` option now returns jobs, and in the constructor companies is filled through as follows:

```

for (job in jobs) {
  companies.add(job.company);
}

```

The filters need the following code:

```

final companyFilterMap = <String, bool>{};
String typeFilterString = "";

void clearFilters() {
  companyFilterMap.clear();
  typeFilterString = "";
}

```

Furthermore, we have refactored the model code (class `Job`) in its own file `lib\model\job.dart`.

The company filter is a custom formatter that has to be coded. We find it in lib\formatter\company_filter.dart as follows:

```
library company_filter;

import 'package:angular/angular.dart';

@Formatter(name: 'companyfilter')
class CompanyFilter {
    List call(JobList, filterMap) {
        if (JobList is Iterable && filterMap is Map) {
            // If there is nothing checked, treat it as "everything is checked"
            bool nothingChecked = filterMap.values.every((isChecked) =>
                !isChecked);
            return nothingChecked
                ? JobList.toList()
                : JobList.where((i) => filterMap[i.company] == true).toList();
        }
        return const [];
    }
}
```

3. In <div job-listing> in angular_formatter.html, we now format the job type as {{job.type | uppercase}}.
4. In lib\formatter, we code this uppercase formatter as follows:

```
import 'package:angular/angular.dart';

@Formatter(name: 'uppercase')
class UppercaseFormatter {
    call(String name) {
        if (name == null || name.isEmpty) return '';
        return name.toUpperCase();
    }
}
```

5. The angular_formatter.dart script has two additional import statements and two bind statements for the formatters, as follows:

```
import 'package:angular_formatter/formatter/company_filter.dart';
import 'package:angular_formatter/formatter/uppercase_formatter.dart';

class AppModule extends Module {
    AppModule() {
```

```

    bind(JobListingController) ;
    bind(SalaryComponent) ;
    bind(CompanyFilter) ;
    bind(UppercaseFormatter) ;
}

```

How it works...

In step 1, we see the two filters. The filter on type is done with an input text field that has a special attribute, `ng-model="ctrl.typeFilterString"`. This tells us that the `JobListingController` must have a property `typeFilterString`, which is bound to this input field by `ng-model`. The same goes for the checkboxes, which are bound to a controller property, `companyFilterMap`:

```
ng-model="ctrl.companyFilterMap[company]"
```

Indeed, we see these properties declared in the additional controller code in step 3. Note that `companyFilterMap` is a map built from the company names and the Boolean values of the checkboxes. Step 2 shows us the filtering declaration: first the jobs are ordered by type, then the type filter is applied (if any), and then the company filters (if any). The results are piped (or chained) with a `|` operator from one filter to the other.

Step 4 shows the code for the company filter: it has to be a separate class, `CompanyFilter`, that is annotated with `@Formatter(name: 'companyfilter')`, where the name is used in the `ng-repeat` attribute. This class must have a `call` method, as first argument the model object to be formatted (here, `JobList`), the second argument `filterMap` is the filter to be applied. So the return value of `call` is the filtered (formatted) job list. The `filterMap` parameter is the data that comes from the checkbox inputs. Steps 5 and 6 show how to add a simple uppercase formatter. In step 7, we add our custom filters and formatters to our `AppModule`.

See also

- ▶ The `Angular.dart` framework here makes use of the special method `call`, which is explained in the *Using the call method* recipe in *Chapter 4, Object Orientation*

Creating a view

In this recipe, we isolate the code for the filters from previous recipe in its own component: `search_job` in the folder `lib\component\`. You can follow along with the code in the project `angular_view`.

How to do it...

The change we make in this recipe is transparent to the user; the web page stays the same, but the project code is refactored.

1. In our main web page angular-view.html, the `<div id="filters">` section is now replaced by the HTML code for the component. Have a look at the following code:

```
<search-job
    type-filter="ctrl.typeFilter"
    company-filter-map="ctrl.companyFilterMap">
</search-job>
```

2. In the constructor of JobListingController, the following code is added:

```
for (var company in companies) {
    companyFilterMap[company] = false;
}
```

3. The behavior of the component is coded in `search_job_component.dart` as follows:

```
import 'package:angular/angular.dart';

@Component(
    selector: 'search-job',
    templateUrl: 'packages/angular_view/component/search_job_
component.html',
    publishAs: 'cmp')
class SearchJobComponent {
    Map<String, bool> _companyFilterMap;
    List<String> _companies;

    @NgTwoWay('type-filter')
    String typeFilter = "";

    @NgTwoWay('company-filter-map')
    Map<String, bool> get companyFilterMap => _companyFilterMap;
    void set companyFilterMap(values) {
        _companyFilterMap = values;
        _companies = companyFilterMap.keys.toList();
    }

    List<String> get companies => _companies;

    void clearFilters() {
```

```
    _companyFilterMap.keys.forEach((f) => _companyFilterMap[f] =  
false);  
    typeFilter = "";  
}  
}  
}
```

- The structure of the filter component is now coded in `search_job_component.html` as follows:

```
<div id="filters">
  <div>
    <label for="type-filter">Filter jobs by type</label>
    <input id="type-filter" type="text" ng-model="cmp.typeFilter" value=" " >
  </div>
  <div>
    Filter jobs by company: <span
      ng-repeat="company in cmp.companies" > <label> <input type="checkbox"
      ng-model="cmp.companysFilterMap[company]" checked="">{{company}}
    </label>
    </span>
  </div>
  <input type="button" value="Clear Filters"
    ng-click="cmp.clearFilters()" >
</div>
```

How it works...

In step 1, we see that the main web page is now greatly simplified; instead of containing all of the markup to set up the search and filter views, it now just contains the reference to the component `search_job`. This component has two attributes, whose values must be set by `ctrl`, our `JobListingController`. Step 2 shows the code to set `companyFilterMap`. The two filter attributes are declared in step 3 as `@NgTwoWay`. Indeed the user must be able to set them, and we want to be able to clear them in code in `clearFilters()`. The component template code in step 4 is not changed.

Using a service

The following step is to read the data from a JSON file, which we will use in this recipe. Angular has a built-in core functionality called the HTTP Service to make HTTP requests to a server. In our example, the job data has been serialized to the JSON format in the file `jobs.json`, and we will make an HTTP request to the web server to get this data. You can follow along with the code in the project `angular-service`.

How to do it...

The change we make in this recipe is nearly transparent to the user; the web page stays the same, but because making an HTTP request is asynchronous, we will work with a Future and must provide a message, such as "Loading data...", as long as the request is being executed.

1. In our `JobListingController` controller class, `lib\job_listing.dart`, we define a new variable of the type `Http`: `final Http _http;`. The constructor now becomes the following:

```
JobListingController(this._http) {  
    _loadData();  
}
```

The bulk of the change takes place in its `_loadData()` method, as shown in the following code:

```
static const String LOADING_MESSAGE = "Loading jobs listing...";  
static const String ERROR_MESSAGE = "Sorry! The jobs database  
cannot be reached.";  
String message = LOADING_MESSAGE;  
  
void _loadData() {  
    jobsLoaded = false;  
    _http.get('jobs.json')  
        .then((HttpResponse response) {  
            jobs = response.data.map((d) => new Job.fromJson(d)).toList();  
            jobsLoaded = true;  
            for (var job in jobs) { // extract companies:  
                companies.add(job.company);  
            }  
        })  
        .catchError((e) {  
            print(e);  
            message = ERROR_MESSAGE;  
        });  
}
```

In the class `Job` we have a new named constructor, as follows:

```
Job.fromJson(Map<String, dynamic> json) : this(json['type'],  
    json['salary'], json['company'], DateTime.parse(json['posted']),  
    json['skills'], json['info']);
```

How it works...

In step 1, we see that Angular uses dependency injection when instantiating the controller to supply it with an `Http` object. Step 2 shows how the `get` method from the `Http` service is used to make a `GET` request and to fetch data from the server (here the data is local, but it could be fetched from a remote site). This method returns a `Future<HttpResponse>` when the request is fulfilled. That's why we have to use a `.then` construct to register a callback function and a `.catchError` to handle exceptions. In the callback function, we have the data in `response.data`. With `map`, we call for each job as a JSON string. The named constructor `fromJson` in the class `Job` from step 3 to transform it into a `Job` object. Finally, a list is made with all job data, which is bound to the view in the same way as in the previous recipes.

See also

- ▶ For more information about Futures, refer to *Chapter 8, Working with Futures, Tasks, and Isolates*
- ▶ For more details about JSON HTTP requests, refer to the *Downloading a file* recipe in *Chapter 6, Working with Files and Streams*

Deploying your app

To run in any modern browser, your Angular app has to be compiled to minimal JavaScript. Minimal means tree-shaken (so that unused code is left out) and minified (shortening of names and minimum use of spaces).

But first, test your app in other browsers by performing the following steps:

1. First, test in your default browser by right-clicking on the startup web page and selecting **Run as JavaScript**. This will compile to JavaScript and execute the app in the browser, but the compiled code will be kept in memory and not written to disk.
2. To change browsers, go to the menu-item and navigate to **Tools | Preferences | Run | Debug**, uncheck **Use system default browser**, and select the other browser. You can also just start that browser and copy the URL from the app in your default browser.

How to do it...

1. Include the Angular transformer into the app's `pubspec.yaml` file:

```
transformers:  
- angular
```
2. Also, add the `js` and `shadow_dom` packages to your `pubspec.yaml` file.

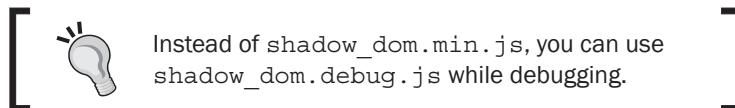
3. Add the shadow DOM script to the startup HTML file, as follows:

```
<script src="packages/shadow_dom/shadow_dom.min.js"></script>
```

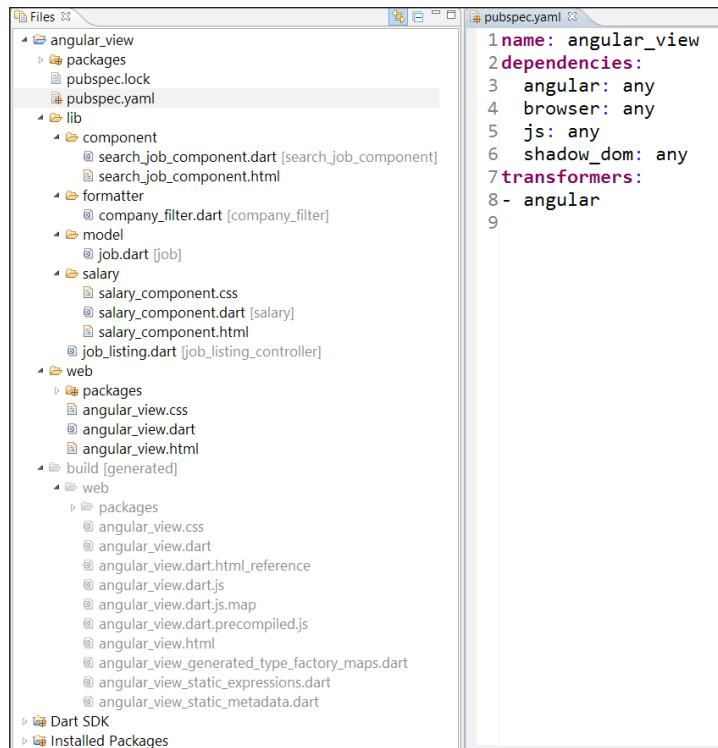
4. Use `pub build` either from the command line or from Dart Editor.

How it works...

Step 2 will make sure that Angular can work with Shadow DOM in all browsers (for more information on Shadow DOM, refer to the *How it works* section in the *Setting up an angular app* recipe in this chapter).



Step 3 will generate all deployable files in your app's directory in a subfolder named `build/web`. For example, if we apply this to the `angular_view` app, we get a JavaScript file, `angular_view.dart.js`, of about 1.7 MB to start up together with `angular_view.html`. The following screenshot gives us an overview of the `angular_view` project:



Overview of the project layout

See also

- ▶ You can learn more about transformers at <https://www.dartlang.org/tools/pub/assets-and-transformers.html>. For more information about compiling to JavaScript, see the *Compiling your app to JavaScript* recipe in *Chapter 1, Working with Dart Tools* and the *Publishing and deploying your app* recipe in *Chapter 2, Structuring, Testing, and Deploying an Application*.

Bibliography

This Learning Path is a blend of content, all packaged up keeping your journey in mind. It includes content from the following Packt products:

- *Dart By Example* By Davy Mitchell
- *Mastering Dart* By Sergey Akopkokhyants
- *Dart Cookbook* By Ivo Balbaert



Thank you for buying **Dart Scalable Application Development**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

