

**Kamesh Balasubramanian**

Foreword by:

**Mat Ryer**

Founder, *Machine Box*

# Isomorphic Go

Learn how to build modern isomorphic web applications using the Go programming language, GopherJS, and the Isomorphic Go toolkit



**Packt**

# Isomorphic Go

Learn how to build modern isomorphic web applications using the Go programming language, GopherJS, and the Isomorphic Go toolkit

**Kamesh Balasubramanian**

**Packt**

BIRMINGHAM - MUMBAI

# Isomorphic Go

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book 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 book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, 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 book.

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

First published: December 2017

Production reference: 1271217

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham  
B3 2PB, UK.  
ISBN 978-1-78839-418-5

[www.packtpub.com](http://www.packtpub.com)

# Credits

**Author**  
Kamesh Balasubramanian

**Copy Editor**  
Tom Jacob

**Reviewer**  
Veena Kamesh

**Proofreader**  
Safis Editing

**Acquisition Editor**  
Frank Pohlmann

**Indexer**  
Tejal Daruwale Soni

**Project Editor**  
Suzanne Coutinho

**Graphics**  
Tom Scaria

**Content Development Editor**  
Venugopal Commuri

**Production Coordinator**  
Shraddha Falebhai

**Technical Editor**  
Anupam Tiwari

# Foreword

The dream of having a single language for the entire stack of a project is an honorable one. It reduces or removes the need for mixed disciplines and avoids knowledge silos where only particular people with particular skills can work on particular parts of the codebase. In theory, this dramatically reduces the overhead of starting new projects or maintaining existing ones.

Back in 2009, Ryan Dahl created the first real attempt at this, Node.js, and it was tremendously successful. Developers only needed to know JavaScript, and they could write UI code that ran in the browser as well as backend code that ran on the server.

Since then, the demands for performance and scale have only increased, and Go is taking over as the language of the server. But there's no way to run Go code in the browser.

All hope is not lost; GopherJS solves this by automatically translating Go code into JavaScript code. So the developer works in Go, but the tools turn it into JavaScript that can then be downloaded and executed in the browser.

I first saw GopherJS in action in a talk by Paul Jolly at the Golang UK Conference in London, 2017, where he had built a fully working clone of Gopherize Me. I realized that there was a way to use Go across the entire stack without having to write confused and complicated code.

In *Isomorphic Go*, Kamesh Balasubramanian takes you through pragmatic steps towards building your own complete end-to-end web application written entirely in Go. You'll explore the tooling that makes it possible and look at familiar ideas with fresh eyes and a new perspective. You'll learn new things too and see how they fit into the wider architecture.

Enjoy the adventure you're about to embark on, and please consider writing about your experiences and contributing to the various projects with feedback, bug fixes, and new features.

**Mat Ryer**

Founder, Machine Box

# About the Author

**Kamesh Balasubramanian** is the founder and CEO of Wirecog, LLC. He is the inventor of Wireframe Cognition (Wirecog), a patented, award-winning technology. He has over two decades of software engineering experience, and has created acclaimed solutions in numerous business sectors.

Kamesh is passionate about using the Go programming language for full-stack web development. He is the founder of the Isomorphic Go and the UX toolkit open source projects. He has authored several titles on Go programming published by Packt Publishing.

As a thought leader, Kamesh has delivered insightful talks and presentations on emerging technology at GopherCon India, MIT, and The World Maker Faire (New York Hall of Science). He is also a contributor to Silicon India Magazine.

Kamesh earned his bachelor's degree in Computer Engineering at California State University, Northridge (CSUN).

Additional information about Kamesh can be found on his website: <http://www.kamesh.com>.

# Acknowledgments

This book would not have been possible without the help, guidance, and support of my father, C.V. Balasubramanian. Thank you Dad for your love, for your dedication, and most importantly, for your evergreen outlook towards life. This book is for you, Dad.

A very special thanks to my wife, Veena, and our daughter, Kanika.

In loving memory of C.S. Vaidianathan and Sivakami Vadianathan.

A special recognition to Mahi Mahmoodi, who provided valuable feedback, encouragement, and support that made all the difference in ensuring the success of the Isomorphic Go talk that I delivered at GopherCon India 2017. This book is a direct result of the reception that the talk received. Her contributions to Isomorphic Go are greatly appreciated.

Thanks to all my family members: Dr. Professor K. Chinnaswamy (Professor Emeritus, JSS College of Pharmacy, Ooty), F. Richard and Malika, Dr. Selvam and Selvi, T.S. Balasubramanian (Karnatic Music Book Centre), Dr. Perumal Sampathkumar, Sri. T.K. Krishnan (Financial Advisor, Chartered Accountant, Pune), Kanti Patel and Tara Patel and family, Usha Subramanian, Late Dr. T.V.R. Subramanian, Anthony Azavedo, Salus D'Souza, James D'Souza, and Paul Singh and family.

Thanks for supporting and encouraging me: Dr. Suresh Bhojraj (President of Pharmacy Council of India and Vice Chancellor at JSS University, Mysore) and Seema Bhojraj, Dr. V. Vedagiri (Professor, Madras Medical College), and Dr. Somanathan Balasubramanian (Director of Research and Dean, JSS University, Mysore).

Thanks to all my friends and well wishers: Mahi Mahmoodi, Olga Shalakhina, Don Johnson, Robert Morrison, Himanshu Patel, bootup (Ariana Thacker and Alexandra Thacker), Markus Hederström, Kartik Money, Yaron Partovi, Sashank Nepal, Pooja Kumari, Nilam Saini, Nidhi Sharma, Sara Aghdaie, Mimi K. Yen, Stephanie Santoso, Franky Telles, Joshua English, Khim Wong, Kulong Cha, Sahasak Kemasith, and Saikumar Ramaswami.

A special thanks to Mat Ryer for contributing to this book with the foreword. Mat has created some of the most inspiring and innovative solutions in the Go community. His book, *Go Programming Blueprints*, published by *Packt Publishing*, has been recognized as being among the best titles for building effective solutions in Go, using cutting-edge techniques. It is truly an honor to have Mat's contribution to this book.

Thanks to all the amazing Gophers: Mat Ryer, James Bowman, Alex Ewetumo, Brian Ketelsen, Ahsan Deliri, Shiju Varghese, Larry Clapp, Narendran, Satish Manohar Talim, Gautham Rege, Sameer Oak, Romin Irani, Alex Edwards, The Google Go Team, The GopherJS Team, Richard Musiol, Dmitri Shuralyov, Dominik Honnef, Ashley McNamara, Stephen Gutekanst, Hai Thanh Nguyen, Alex Browne, Alex Alderton, Rafael Cossowan, Tony Pujals, and Cassandra Salisbury.

Additional credits and thanks: The Isomorphic Go logo was created by Olga Shalakhina and inspired by the artwork of Renée French. The racing motif featured in the Isomorphic Go logo is inspired by the concept of "Mechanical Sympathy", which is attributed to the legendary race car driver, Jackie Stewart. The car and the Gopher's wardrobe are fashioned from that classic racing era. Gopher artwork on IGWEB was created using [gopherize.me](http://gopherize.me) (Ashley McNamara and Mat Ryer). IGWEB utilizes the Go font family created by Bigelow & Holmes. CSUN, MIT, World Maker Faire, The Emerging Technology Trust, and Silicon India Magazine. The following companies provided generous support to help make Isomorphic Go possible: The Walt Disney Company, Infosys, Linode, and Wirecog.

# About the Reviewer

**Veena Kamesh** is a software engineer based in Los Angeles, California. She has over 15 years of experience in designing, architecting, and developing high-profile software projects for Tata, McKinsey, HDFC Securities, ICICI Brokerage Services, and the City of Los Angeles.

Veena earned her bachelor's degree in Computer Engineering at Nagpur University, and she is an Oracle Certified DBA. She has served as a reviewer for multiple titles on Go published by Packt Publishing.

Additional information about Veena can be found on her website: <http://veena.io>.

# www.PacktPub.com

For support files and downloads related to your book, please visit [www.PacktPub.com](http://www.PacktPub.com).

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

## Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

# Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1788394186>.

If you'd like to join our team of regular reviewers, you can email us at [customerreviews@packtpub.com](mailto:customerreviews@packtpub.com). We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

*This book is dedicated to my father, C.V. Balasubramanian*

# Table of Contents

<b>Preface</b>	1
<b>Chapter 1: Isomorphic Web Applications with Go</b>	8
<b>Why Isomorphic Go?</b>	9
Go comes with type checking	9
Avoiding Vanilla JavaScript with transpilers	10
Transpiling code	10
Benefits of Go on the front-end	11
Robust standard library	11
Promoting modularization using Go packages	11
An implicit build system	11
Avoiding callback hell	12
Concurrency	12
Isomorphic web application development using Go	12
<b>An overview of web application architectures</b>	13
<b>The classic web application architecture</b>	13
Advantages	15
Faster initial page loads	15
Greater search engine discoverability	16
The primary disadvantage	17
<b>The AJAX web application architecture</b>	19
The primary advantage	20
Disadvantages	21
Mental context shifts	22
Increased rendering complexity	22
Duplication of efforts	23
<b>The single page application (SPA) architecture</b>	23
The primary advantage	25
Disadvantages	25
Slower initial page loads	26
Reduced search engine discoverability	27
<b>The isomorphic web application architecture</b>	29
Wish list fulfilled	30
1. Enhancing the user experience	31
2. Increasing the maintainability	31
3. Increasing the efficiency	31
4. Increasing the productivity	32
5. Making the best first impression	32

6. Promoting discoverability	32
<b>Live demo</b>	33
<b>Measurable benefits</b>	34
<b>Nomenclature</b>	35
<b>Prerequisites</b>	35
<b>Summary</b>	36
<b>Chapter 2: The Isomorphic Go Toolchain</b>	37
<b>Installing the Isomorphic Go toolchain</b>	38
<b>Go</b>	39
Installing Go	41
Setting up your Go workspace	44
Building and running Go programs	45
<b>GopherJS</b>	46
Installing GopherJS	47
Installing essential GopherJS bindings	48
dom	48
jsbuiltin	48
xhr	48
websocket	49
Getting familiar with GopherJS on the command line	49
<b>The Isomorphic Go toolkit</b>	50
Installing isokit	50
<b>The UX toolkit</b>	51
Installing the cog package	51
<b>Setting up the IGWEB demo</b>	51
Setting up the application root environment variable	51
Transpiling the client-side application	52
Setting up Redis	52
Running the IGWEB demo	54
Loading the sample dataset	56
Using kick	59
Installing kick	59
Running kick	59
Verify that kick is working	60
<b>An introduction to the IGWEB demo</b>	61
Building IGWEB from the ground up	62
The IGWEB roadmap	62
The home page	62
The about page	63
The products page	63
The shopping cart feature	63
The contact page	64

The live chat feature	64
Reusable components	64
<b>Project structure and code organization</b>	65
The MVC pattern	68
The custom datastore	69
Dependency injections	71
<b>Summary</b>	75
<b>Chapter 3: Go on the Front-End with GopherJS</b>	76
<b>The Document Object Model</b>	77
Accessing and manipulating the DOM	79
<b>Basic DOM operations</b>	79
Displaying an alert message	79
Getting a DOM element by ID	82
Query selector	82
Changing the CSS style property of an element	83
<b>GopherJS overview</b>	84
The GopherJS transpiler	85
<b>GopherJS examples</b>	88
Displaying an alert message	93
Changing an element's CSS style property	95
The JavaScript typeof operator functionality	99
Transforming text to lowercase using an XHR post	101
<b>Inline template rendering</b>	106
The cars listing demo	106
Transmitting gob encoded data	110
<b>Local storage</b>	113
Common local storage operations	114
Setting a key-value pair	114
Getting a value for a given key	114
Getting all key value pairs	115
Clearing all entries	115
Building a local storage inspector	115
Creating the user interface	116
Setting up the server-side route	119
Implementing the client-side functionality	119
Initializing the local storage inspector web page	120
Implementing the local storage inspector	120
Running the local storage demo	123
<b>Summary</b>	126
<b>Chapter 4: Isomorphic Templates</b>	127

<b>The web template system</b>	128
The template engine	130
The template data object	130
The templates	132
<b>IGWEB page structure</b>	132
The header	133
The top bar	134
The navigation bar	134
The primary content area	134
The footer	134
<b>Template categories</b>	135
Layout templates	135
The web page layout template	136
Partial templates	136
The header partial template	137
The top bar partial template	138
The navigation bar partial template	138
The footer partial template	139
Regular templates	139
The page template for the about page	141
The content template for the about page	141
<b>Custom template functions</b>	143
<b>Feeding data to the content template</b>	144
<b>Isomorphic template rendering</b>	146
Limitations of filesystem-based template rendering	147
The in-memory template set	148
Setting up the template set on the server side	151
Registering the server-side handlers	154
Serving the template bundle items	155
Rendering the about page from the server side	156
Setting up the template set on the client side	159
Creating the client-side router	162
Registering the client-side route	163
Initializing interactive elements on the web page	163
Rendering the about page from the client side	164
The gopher team Rest API endpoint	166
<b>Summary</b>	168
<b>Chapter 5: End-to-End Routing</b>	169
<b>Routing perspectives</b>	170
Routing on the server side	170

Routing on the client side	171
<b>Design of the product-related pages</b>	173
<b>Implementing product-related templates</b>	174
Implementing the templates for the products listing page	174
Implementing the templates for the product detail page	176
<b>Modeling product data</b>	177
<b>Accessing product data</b>	178
Retrieving products from the datastore	178
Retrieving the product detail from the datastore	180
<b>Registering server-side routes with Gorilla Mux</b>	181
<b>Server-side handler functions</b>	182
The handler function for the products listing page	182
The handler function for the product detail page	185
<b>Registering client-side routes with the isokit router</b>	187
<b>Client-side handler functions</b>	188
The handler function for the products listing page	188
Fetching the list of products	190
The handler function for the product detail page	191
Fetching the product detail	192
<b>Rest API endpoints</b>	193
The endpoint to get the list of products	194
The endpoint to get the product detail	194
Verifying the client-side routing functionality	196
<b>Summary</b>	197
<b>Chapter 6: Isomorphic Handoff</b>	198
<b>The isomorphic handoff procedure</b>	199
The ERDA strategy	199
The Encode step	201
The Register step	201
The Decode step	202
The Attach step	202
<b>Implementing isomorphic handoff for the product-related pages</b>	203
Implementing the sort interface for the product model	203
Implementing isomorphic handoff for the products listing page	205
Implementing isomorphic handoff for the product detail page	209
<b>Implementing isomorphic handoff for the shopping cart</b>	213
Designing the shopping cart page	213
Implementing the shopping cart templates	215
The template data object	217
Modeling the shopping cart	218

<b>Shopping cart routes</b>	220
Fetching a list of items	221
Adding an item	221
Removing an item	221
<b>The session store</b>	222
<b>The server-side shopping cart handler function</b>	223
<b>Shopping cart endpoints</b>	226
The endpoint to get items in the shopping cart	226
The endpoint to add items to the shopping cart	227
The endpoint to remove items from the shopping cart	229
<b>Implementing the shopping cart functionality on the client side</b>	231
Rendering the shopping cart	231
Removing an item from the shopping cart	233
Adding an item to the shopping cart	235
<b>Verifying the shopping cart functionality</b>	236
<b>Summary</b>	239
<b>Chapter 7: The Isomorphic Web Form</b>	240
<b>Understanding the form flow</b>	242
<b>Designing the contact form</b>	244
Implementing the templates	246
<b>Validating email address syntax</b>	248
<b>The form interface</b>	249
<b>Implementing the contact form</b>	250
Registering the contact route	253
The contact route handler	254
The contact confirmation route handler	256
Processing the contact form submission	257
<b>The accessible contact form</b>	259
The contact form can function without JavaScript	266
<b>Client-side considerations</b>	272
<b>Contact form Rest API endpoint</b>	277
<b>Checking the client-side validation</b>	279
Tampering with the client-side validation result	284
<b>Summary</b>	287
<b>Chapter 8: Real-Time Web Application Functionality</b>	288
<b>The live chat feature</b>	289
Designing the live chatbox	290
Implementing the live chat box templates	291
<b>Implementing the live chat's server-side functionality</b>	292

The hub type	294
The client type	296
Activating the chat server	299
The agent's brain	300
Greeting the human	305
Replying to a human's question	306
Exposing the agent's information to the client	309
<b>Implementing the live chat's client-side functionality</b>	310
Creating the live chat client	310
Initializing the event listeners	313
The close chat control	314
Setting up event listeners for the WebSocket object	314
Handling a disconnection event	318
<b>Conversing with the agent</b>	319
<b>Summary</b>	324
<b>Chapter 9: Cogs – Reusable Components</b>	326
<b>Essential cog concepts</b>	327
The UX toolkit	328
The anatomy of a cog	331
The virtual DOM tree	335
The life cycle of a cog	338
<b>Implementing pure cogs</b>	339
The time ago cog	340
The live clock cog	347
<b>Implementing hybrid cogs</b>	354
The date picker cog	354
The carousel cog	363
The notify cog	371
<b>Summary</b>	378
<b>Chapter 10: Testing an Isomorphic Go Web Application</b>	379
<b>Testing the server-side functionality</b>	380
Go's testing framework	380
Verifying server-side routing and template rendering	381
Verifying the contact form's validation functionality	383
Verifying a successful contact form submission	386
<b>Testing the client-side functionality</b>	388
CasperJS	389
Verifying client-side routing and template rendering	391
Verifying the contact form	403

Verifying the shopping cart functionality	411
Verifying the live chat feature	416
Verifying the time ago cog	422
Verifying the live clock cog	424
Verifying the date picker cog	427
Verifying the carousel cog	429
Verifying the notify cog	432
<b>Summary</b>	436
<b>Chapter 11: Deploying an Isomorphic Go Web Application</b>	437
<b>How IGWEB operates in production mode</b>	438
GopherJS-produced JavaScript source file	439
Taming the GopherJS-produced JavaScript file size	440
Generating static assets	441
<b>Deploying an Isomorphic Go web application to a standalone server</b>	444
Provisioning the server	444
Setting up the Redis database instance	445
Setting up the NGINX reverse proxy	446
Enabling GZIP compression	448
Proxy settings	448
Setting up the IGWEB root folder	450
Cross compiling IGWEB	450
Preparing the deployment bundle	451
Deploying the bundle and starting IGWEB	453
Running IGWEB	453
Running IGWEB in the foreground	454
Running IGWEB in the background	455
Running IGWEB with systemd	455
<b>Deploying an Isomorphic Go web application using Docker</b>	458
Installing Docker	458
Dockerizing IGWEB	459
The Dockerfile	459
The Dockerfile for a closed source project	462
Docker compose	463
Running Docker compose	465
Setting up the dockerized IGWEB service	468
<b>Summary</b>	470
<b>Appendix: Debugging Isomorphic Go</b>	471
<b>Identifying compiler/transpiler errors</b>	471
<b>Examining panic stack traces</b>	472
<b>Tracing code to pinpoint the source of issues</b>	475



# Preface

In February 2017, I delivered a talk on Isomorphic Go at GopherCon India. Isomorphic Go is the methodology to create isomorphic web applications using the Go programming language. Isomorphic web application architecture promotes an enhanced user experience along with greater search engine discoverability. My talk on Isomorphic Go was well received and a group of full-stack engineers reached out to me afterwards.

The engineers I met with expressed frustration at the complexity involved in having to maintain full-stack web application codebases. They lamented on having to juggle between two distinctly different programming languages, Go on the back-end and JavaScript on the front-end. Whenever they had to develop solutions with JavaScript, they longed for the simplicity in syntax, the robust standard library, the concurrency constructs, and the type safety that comes with Go, out of the box.

The message that they were conveying to me was loud and clear. They yearned for the ability to create full-stack web applications exclusively in Go. The proposition of being able to have a single unified Go codebase where developers could code in Go on both the front-end and back-end was an alluring one indeed.

I created two action items for myself as a result of the feedback that I gathered from the conference. First, there needed to be a demonstration of the capabilities of an Isomorphic Go web application. Second, there needed to be a detailed explanation of all the major concepts behind Isomorphic Go.

The first action item, the demonstration of Isomorphic Go, became the genesis of the Isomorphic Go and UX toolkit open source projects—at the time of writing, these are the most advanced technologies for creating isomorphic web applications in Go. The first isomorphic web application to be created in Go is the IGWEB demo (available at <http://igweb.kamesh.com>), which is the web application that is featured in this book.

The second action item, the explanation of major Isomorphic Go concepts, manifested itself as this book. After watching my talk on Isomorphic Go, Packt Publishing reached out to me with an opportunity to write this book, which I gladly accepted. It was truly an exhilarating experience being able to write a book about an emerging technology that I was extremely passionate about.

Writing this book provided me the opportunity to present ideas and concepts that had never been covered previously in the realm of Go programming, such as in-memory template sets, end-to-end routing, isomorphic handoff, isomorphic web forms, real-time web application functionality, reusable components with Go, writing end-to-end automated tests that exercised client-side functionality, and the deployment of an isomorphic web application written in Go.

The extensive depth of this book's coverage ensured that I fulfilled an important responsibility to you, the reader, that I provided high value for your investment. This is especially significant since this happens to be the first and only book on Isomorphic Go.

This book focuses on teaching you how to create an Isomorphic Go web application from the ground up. This book is a journey starting out with presenting the advantages of creating an isomorphic web application with Go, and ending with the deployment of a multi-container Isomorphic Go web application to the cloud.

I hope you enjoy reading this book, and that it will be a valuable resource to you for many years to come.

## What this book covers

Chapter 1, *Isomorphic Web Applications with Go*, presents the benefits of the isomorphic web application architecture and explains why Go makes a compelling choice for the creation of isomorphic web applications.

Chapter 2, *The Isomorphic Go Toolchain*, guides you through the process of installing the Isomorphic Go toolchain, introduces you to the IGWEB project, walks you through the project file structure, provides the project roadmap, and highlights some coding techniques used over the course of the book.

Chapter 3, *Go on the Front-End with GopherJS*, introduces you to programming with Go on the front-end using GopherJS. We show you how to perform basic DOM operations, change the CSS style property of elements, call built-in JavaScript functionality, make XHR calls, encode data in the gob format, render inline templates, and create a local storage inspector.

Chapter 4, *Isomorphic Templates*, introduces the web template system consisting of the template engine, the template data object, and the template. It shows you how to organize templates based on their purpose. It explains the concept of isomorphic template rendering, and shows you how to create an in-memory template set that can be utilized across environments.

Chapter 5, *End-to-End Routing*, explains how to register server-side routes using the Gorilla Mux router, and also shows you how to register client-side routes using the isokit router. It also shows you how to create server-side Rest API endpoints that handle XHR requests.

Chapter 6, *Isomorphic Handoff*, introduces you to the concept of isomorphic handoff, the ability to pass state from the server to the client. It shows you how to implement isomorphic handoff using the ERDA (Encode-Register-Decode-Attach) strategy. It also shows you how to verify that isomorphic handoff has been implemented successfully.

Chapter 7, *The Isomorphic Web Form*, shows you how to build an accessible isomorphic web form, having the ability to share both form code as well as validation logic across environments. It shows you how to ensure that the form functions for web browsers that don't have a JavaScript runtime, and also shows you how to perform client-side form validation for web browsers that do.

Chapter 8, *Real-Time Web Application Functionality*, shows you how to implement real-time web application functionality in the form of a live chat feature. It shows you how to create a chat bot that can answer simple questions about Isomorphic Go. It also shows you how to gracefully shut down the feature if the internet connection is interrupted.

Chapter 9, *Cogs – Reusable Components*, introduces you to cogs, reusable components that can be implemented either exclusively in Go (pure cogs) or with Go and JavaScript (hybrid cogs). It explains the anatomy of a cog by exploring its file structure. It also explains how cogs utilize a virtual DOM to render their contents in an efficient manner.

Chapter 10, *Testing an Isomorphic Go Web Application*, shows you how to implement automated end-to-end testing to verify the functionality of an Isomorphic Go web application. It shows you how to verify server-side functionality by implementing tests using Go's testing package. It shows you how to verify client-side functionality by implementing CasperJS tests to test multiple user interaction scenarios on the client-side.

Chapter 11, *Deploying an Isomorphic Go Web Application*, explains how an Isomorphic Go web application behaves in production mode. It shows you how to deploy an Isomorphic Go web application to a standalone server in the cloud. It also shows you how to deploy an Isomorphic Go web application as a multi-container Docker application running in the cloud.

Appendix, *Debugging Isomorphic Go*, guides you through the process of debugging an Isomorphic Go web application. It shows you how to identify compiler/transpiler errors, examine panic stack traces, and trace code to pinpoint the source of issues.

## What you need for this book

To compile the code that comes with this book, you will need a computer running an operating system that has a Go distribution available for it. A list of supported operating systems along with system requirements can be found at <https://golang.org/doc/install#requirements>.

## Who this book is for

This book addresses readers who have had prior experience with the Go programming language and understand the language's essential concepts. It is also assumed that the reader has had prior experience with basic web development. No previous knowledge of isomorphic web application development is required. Since this book follows an idiomatic approach using Go, it is not necessary that the reader should have had prior experience using JavaScript or any tools or libraries within the JavaScript ecosystem.

## Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Let's examine the `products_page tmpl` source file."

A block of code is set as follows:

```
 {{ define "pagecontent" }}
{{template "products_content" . }}
{{end}}
{{template "layouts/webpage_layout" . }}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
func NewWidget() *Widget {
    w := &Widget{}
    w.SetCogType(cogType)
    return f
}
```

Any command-line input or output is written as follows:

```
$ go get -u github.com/ux toolkit/cog
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "This sends a mouse click event to the first **Add to Cart** button on the web page."

Warnings or important notes appear like this.



Tips and tricks appear like this.



## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—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 email [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book'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](http://www.packtpub.com/authors).

## Customer support

Now that you are the proud owner of a Packt book, 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 book 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 emailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your email 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 book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

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 book is also hosted on GitHub at <https://github.com/PacktPublishing/Isomorphic-Go>. We also have other code bundles from our rich catalog of books and videos 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 books-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 book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, 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 book 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](mailto: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 book, you can contact us at [questions@packtpub.com](mailto:questions@packtpub.com), and we will do our best to address the problem.

# 1

## Isomorphic Web Applications with Go

Isomorphic web applications, are applications where the web server and the web browser (the client), may share all, or some parts of the web application code. Isomorphic web applications allow us to reap maximum benefits from traditional web application architectures. They provide a better user experience, enhanced discoverability by search engines, and reduced operating costs by sharing parts of the web application code across environments.

Well-established businesses, such as Airbnb, Bloomberg, Capital One, Facebook, Google, Netflix, and Walmart have accepted isomorphic web application development, and with good reason—the financial bottom line.

A study by Walmart found that for every *1 second of improvement, they experienced up to a 2% increase in conversions*. In addition to that, they also found that for every *100 milliseconds of improvement, they grew incremental revenue by up to 1%*. Source: How Website Speed Affects Conversion Rates (<http://www.globaldots.com/how-website-speed-affects-conversion-rates/>).

Isomorphic Go is the methodology to create isomorphic web applications using the Go programming language. In this book, we will explore, in-depth, the process of creating an isomorphic web application in Go.

This chapter will cover the following topics:

- Why you should consider isomorphic Go for developing modern web applications
- An overview of the traditional web application architectures
- An introduction to the isomorphic web application architecture
- When to implement isomorphic web applications
- What you should know before learning Isomorphic Go

## Why Isomorphic Go?

There is no doubt that JavaScript is the current leading technology, in terms of market share and mindshare, for creating isomorphic web applications. On the client-side, JavaScript comes included with all the major web browsers. JavaScript can now, also exist on the server side, thanks to Node.js.

If this is the case, then why should we focus our attention on Go for creating isomorphic web applications? The answer to this question is manifold. Consider the list of answers provided here, as an initial list, which is the starting point for our discussion:

- Go comes with type checking
- Even the tech giants avoid Vanilla JavaScript
- Transpiling code to Vanilla JavaScript has become widely accepted
- Go comes with a lot of benefits for front-end web development

## Go comes with type checking

Go is a language that includes built-in static type checking. The immediate ramification of this fact is that many errors can be caught at compile time itself.

The single-most pain point for many JavaScript developers is the lack of static type checking in JavaScript. Having personally worked in JavaScript code bases that have spanned several hundred thousand lines of code, I have seen first hand, how the most trivial bug can arise from the lack of static type checking.

## Avoiding Vanilla JavaScript with transpilers

To avoid writing Vanilla JavaScript, tech giants Microsoft and Google, have created TypeScript and Dart, respectively, as languages and transpilers. A **transpiler** is a source code to source code compiler.

A compiler will turn human readable code, written in a programming language, into machine code. A transpiler is used to transform source code from one programming language into that of another language. The output may or may not be readable by a human, depending on the intent of the transpiler.

Languages, such as Typescript and Dart, get transpiled into Vanilla JavaScript code, so that they can be run in JavaScript-enabled web browsers. In the case of TypeScript, it's essentially a superset of JavaScript that introduces static type checking. The creators of the AngularJS Framework chose TypeScript over Vanilla JavaScript as the language of choice, to develop the next major version of their framework.

This approach to side stepping JavaScript, using an alternative programming language and a transpiler, creates a win-win situation for the developer. The developer is allowed to program in a programming language that is most productive for them and, at the end of the day, the code the developer created, is guaranteed to run in the web browser—thanks to the transpiler.

## Transpiling code

Transpiling code to JavaScript, has become a widely accepted practice, even within the JavaScript community itself. For instance, the Babel transpiler allows developers to code in yet-to-be-released future standards of the JavaScript language, which get transpiled into the widely accepted standard JavaScript code that's currently supported in the major web browsers.

Within this context, it is not outlandish, or far-fetched, to run Go programs, that get transpiled into JavaScript code, in the web browser. In fact, besides static type checking, there are many more benefits to be gained from being able to run Go on the front-end.

## Benefits of Go on the front-end

Having Go available on the front-end comes with many advantages, including the following:

- A robust standard library
- Code modularization is easy with Go packages
- Go comes with an implicit build system
- Go's concurrency constructs allow us to avoid callback hell
- The concept of concurrency comes built-in with Go
- Go can be utilized for isomorphic web application development

## Robust standard library

Go comes with a robust standard library that provides a lot of powerful functionality out of the box. For instance, in Go, we can render an inline client-side template without having to include any third-party template library or framework. We will consider an example of how to do this in [Chapter 3, Go on the Front-End with GopherJS](#).

## Promoting modularization using Go packages

Go has a powerful package implementation that promotes modularization, allowing for far greater code-reuse and maintainability. Also, the Go tool chain includes the `go get` command, which allows us to easily fetch official and third-party Go packages.



If you're coming from the JavaScript world, think of `go get` as a more simple, lightweight `npm` (`npm` is the Node Package Manager, a repository of third-party JavaScript packages).

## An implicit build system

In the JavaScript ecosystem, it is still popular, in modern times, for developers to spend their time manually creating and maintaining project build files. Being a modern programming language, Go ships with an implicit build system.

As long as you follow Go's prescribed conventions, and once you issue the `go build` command for your Go application, the implicit build system kicks in. It will build and compile the Go project automatically by examining the dependencies it finds, within the application's Go source code itself. This provides a major productivity boost for the developer.

## Avoiding callback hell

Perhaps the most compelling reason to consider Go for isomorphic web development is to avoid *callback hell*. JavaScript is a single threaded programming language. When we want to delay the execution of a particular task after an asynchronous call has been made, we would place the code for those tasks inside a callback function.

Soon enough, our list of tasks to delay for execution will grow, and the amount of nested callback functions will grow along with it. This situation is known as *callback hell*.

We can avoid callback hell in Go, using Go's built-in concurrency constructs.

## Concurrency

Go is a modern programming language designed to be relevant in an age of multicore processors and distributed systems. It was designed in a manner where the importance of concurrency was not treated as an afterthought.

In fact, concurrency was so important to the creators of Go, that they built concurrency right into the language itself. In Go, we can avoid callback hell, using Go's built-in concurrency constructs: goroutines and channels. **Goroutines** are cheap, lightweight threads. **Channels**, are the conduits that allow for communication between goroutines.

## Isomorphic web application development using Go

When it comes to isomorphic web application development, JavaScript is no longer the only game in town. Due to recent technological advancements, notably the creation of **GopherJS**, we can now use the Go programming language on the front-end; this allows us to create isomorphic web applications in Go.

**Isomorphic Go** is an emerging technology that provides us the essential ingredients needed to create isomorphic web applications, using the powerful and productive features that the Go programming language has to offer. In this book, we will use the functionality from Go's standard library and third-party libraries from the Go community to implement an isomorphic web application.

## An overview of web application architectures

In order to understand and fully appreciate the architecture of isomorphic web applications, it's insightful to have an understanding of the web application architectures that preceded it. We will cover the major web application architectures that have been prevalent in the industry over the past 25 years.

After all, we can't truly appreciate where we have arrived at, until we've fully acknowledged where we have been. With the monumental changes that have occurred in the web application architecture realm over the years, there is much to acknowledge.

Before we present the isomorphic web application architecture, let's devote some time to review the three traditional web application architectures that came before it:

- The classic web application architecture
- The AJAX web application architecture
- The **single-page application (SPA)** architecture

We'll identify the advantages and disadvantages for each of the three architectures considered. We will start a wish list of requirements, based on each disadvantage that we identify for a given architecture. After all, a shortcoming is actually an opportunity for improvement.

## The classic web application architecture

The **classic web application architecture** dates back to the early 1990s, when graphical web browsers started to gain traction. When the user interacts with a web server using a web browser, each user interaction, makes a request to a web server using HTTP. *Figure 1.1* depicts the classic web application architecture:

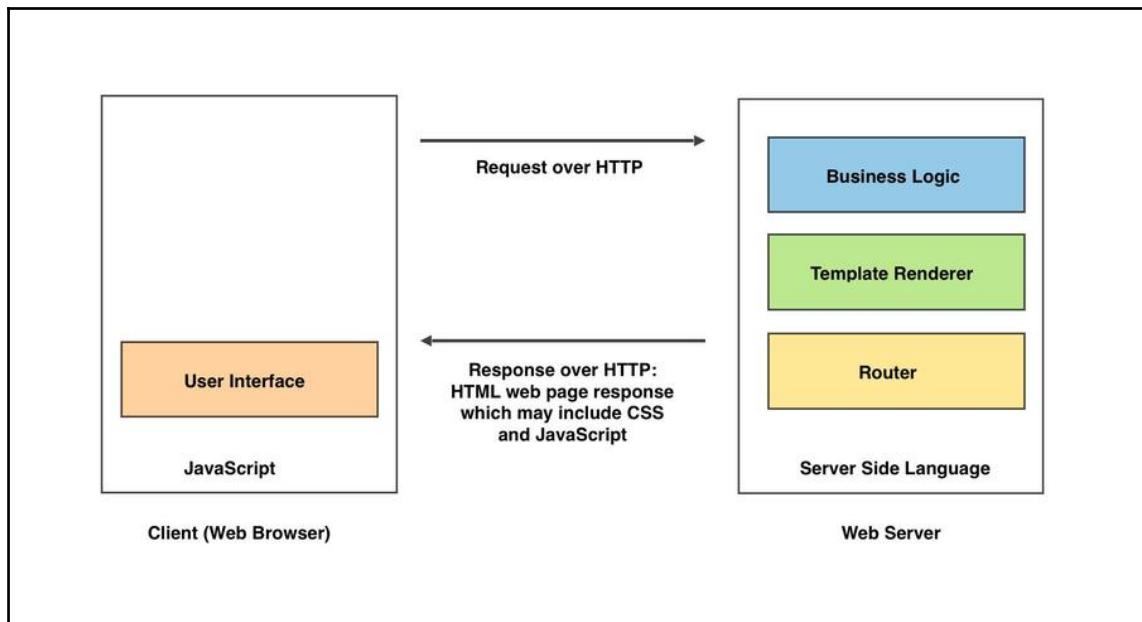


Figure 1.1: The Classic Web Application Architecture

The diagram also depicts an HTTP transaction, which consists of a request that's sent from the user's web browser to the web server. Once the web server accepts the request, it will return a corresponding response for that request.

Typically, the response is an HTML web page, which may either contain inline CSS and/or JavaScript, or call external CSS stylesheets and/or JavaScript source files.



There are two types of resources that the web server can return in the form of a response: a static resource and a dynamic resource.

A **static resource** is a file. For example, it could be an HTML, JPEG, PDF, or MP4 file that lives on the web server. The server will return the document specified by the request in its response body.

A **dynamic resource** is a resource that gets built by the server on the fly. An example of a dynamic resource is a search engine's search results page. Usually, the response body of a dynamic request will be formatted in HTML.

When it comes to web applications, we deal with dynamic resources. The web server is serving a web application, and usually the web application contains a controller with the logic that routes the user's request to a specific action to perform on the server. Once the web server is done processing the user's request, the server sends a response back to the client in the form of a web page response.

A server-side programming language (such as Go, Perl, PHP, Python, Ruby, and Java) is used to process the requests sent from the web browser. For example, let's consider we have a server side web application that is used for an e-commerce website.

The web application can route requests by making use of a server-side **route handler** (as shown in *Figure 1.1*); the `/product-detail/swiss-army-knife` route can be associated to a product detail controller, which will serve an HTML web page response containing the product profile page for the Swiss Army Knife product.

In a classic web application architecture, the code to render the web page lives on the server side, typically consolidated into template files. Rendering the web page response from a set of templates is performed by the **template renderer** that resides on the server (as shown in *Figure 1.1*).

Usually in this paradigm, JavaScript may be included in a rendered web page to enhance the user experience. In this type of web application architecture, the responsibility of implementing the web application is placed primarily on the server-side language, and JavaScript is placed on the sidelines of being used primarily for user interface controls or enhanced user interactivity for the website.

## Advantages

The classic web application architecture comes with two major advantages:

- Faster initial page loads
- Greater search engine discoverability

### Faster initial page loads

The first primary advantage of the classic web application architecture is that page loads are perceived to be fast by the user since the entire page is rendered at once. This is a result of the web server rendering the web page response, using a template renderer, on the server side itself.

The user does not perceive slowness, since they are delivered the rendered page from the server instantaneously.



Keep in mind that if there is high latency in the server's response time, then the user interaction will come to a grinding halt. In this scenario, the fast initial page load advantage is lost since the user has to stare at a blank screen—waiting for the server to finish processing. This waiting will end with either the web page response being delivered to the user, or the HTTP request timing out—whichever comes first.

## Greater search engine discoverability

The second primary advantage of the classic web application architecture is that this architecture is search engine friendly, since the web application serves up web page responses, in HTML, that can be readily consumed by search engine bots. In addition to this, the server-side route-handler allows for the creation of search engine friendly URLs, that can be associated with a specific server-side controller.

A key factor to making a website friendly to search engines is discoverability. Besides having great content, a search engine friendly website also needs permalinks – web links that are intended to remain in service permanently. Descriptive and well-named URLs can be registered as routes with the server-side's router. These routes end up serving as permalinks, which the search engine bot crawlers can easily index while crawling through the website.

The goal is to have pretty website URLs that can contain meaningful information, which can be easily indexed by a search engine's bot crawler, such as:

`http://igweb.kamesh.com/product-detail/swiss-army-knife.`

The aforementioned permalink is much more easily indexed by a search engine and understood by a human rather than the following one:

`http://igweb.kamesh.com/webapp?section=product-detail&product_id=052486.`

## The primary disadvantage

We'll be examining the primary disadvantage(s) for each of the traditional web application architectures considered in this chapter. *The isomorphic web application architecture* section in this chapter, will show us how the isomorphic web application architecture provides a solution for each disadvantage presented and also gather the benefits offered from each of the traditional web application architectures.

The primary disadvantage of the classic web application architecture is that all user interactions, even the most trivial, require a full page reload.

This means that the **Document Object Model (DOM)**, the tree data structure representing the current state of the web page, and the elements that comprise it, are completely wiped out, and recreated again upon each user interaction:

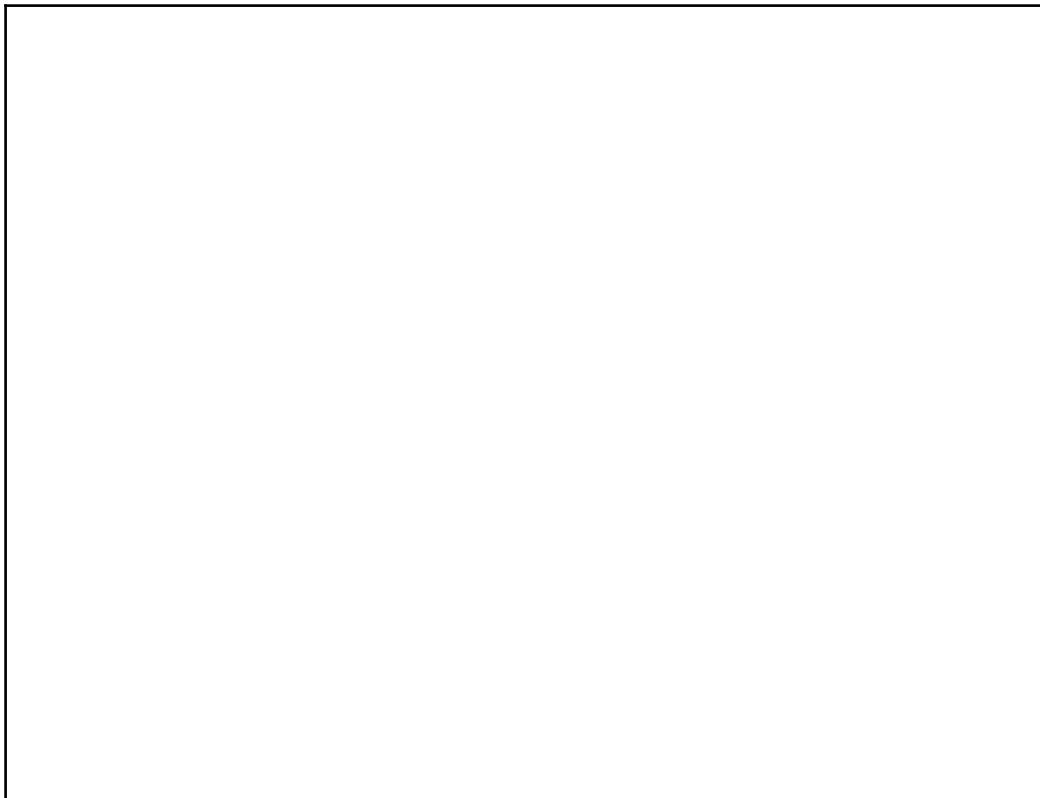


Figure 1.2: A layout diagram of a news website with a comments section and a wireframe depicting the comments section

For example, let's consider that we are reading an article on a news website. *Figure 1.2*, depicts a layout diagram of the news website (the illustration on the left), with the comments section of the website at the bottom of the web page. Other sections may exist on the news website in the negative (empty) space in the layout.

*Figure 1.2* also includes a wireframe design of the news comments section (the illustration on the right), which contains a few sample comments. The ellipses (...) denotes multiple website comments that are not listed for the sake of brevity.

Let's consider scenario where this particular news article has gone viral and it contains more than 10,000 comments. The comments are paginated, and there are 50 comments displayed per page:

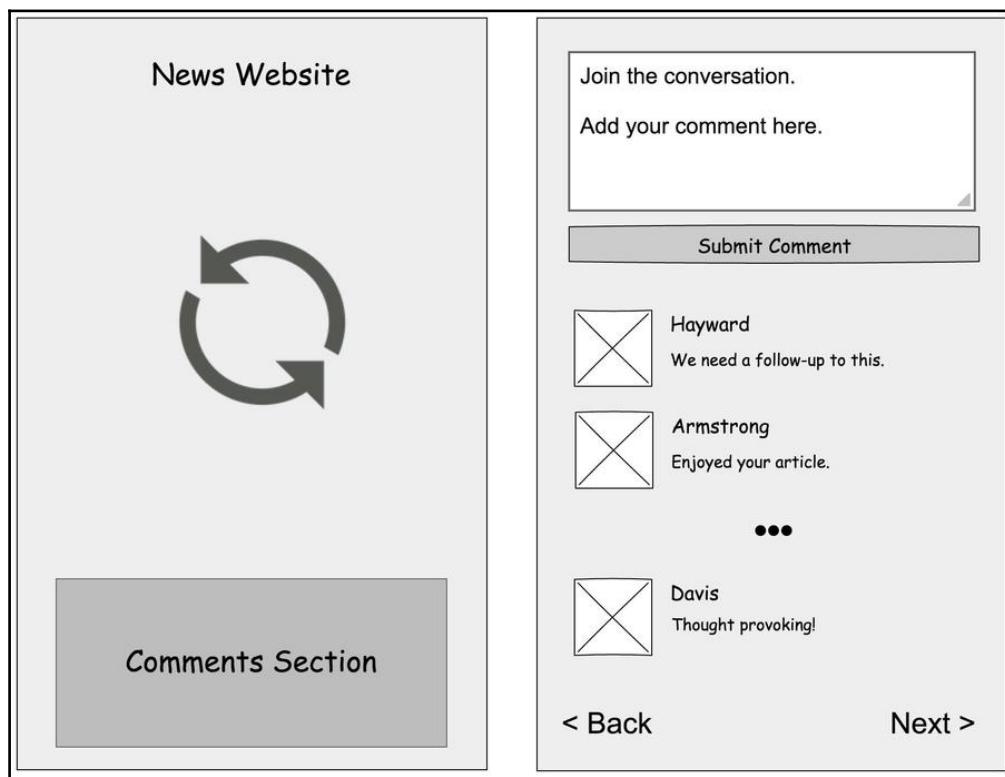


Figure 1.3: The entire web page needs to be refreshed to view the next set of comments

*Figure 1.3* depicts the web page for the news website being refreshed (the illustration on the left). Note that the user will perceive the refresh to be quick because the page will instantaneously load (considering that network latency is low). *Figure 1.3* also depicts the next batch of 50 articles (the illustration on the right) after the next link has been clicked.

If we were to click on the next link on the paginated navigation control, it would cause a full page reload, which would destroy the DOM and recreate it again. Since the comments are located at the bottom of the screen, upon a full page reload, the scroll position may also change back to the top of the web page, resulting in a poor user experience.

We only wanted to see the next set of comments at the bottom of the page. We didn't intend for the whole web page to reload, but it did, and that's the major limitation of the classic web application architecture.



**Wish list item #1:** To enhance the user experience, clicking on a link on the website should not cause a full page reload.

## The AJAX web application architecture

With the advent of the **XMLHttpRequest (XHR)** object, the **Asynchronous JavaScript And XML (AJAX)** era began. *Figure 1.4* illustrates the AJAX web application architecture.

After the client's initial request, the server sends back a web page response containing HTML, CSS, and JavaScript. Once the web page has finished loading, the JavaScript application on the client side may initiate asynchronous requests back to the web server over HTTP, using the XHR object.



Some observers have characterized the advent of AJAX as the *Web 2.0 era*, where websites became more interactive with more rich user experiences and the use of JavaScript libraries started to gain traction.

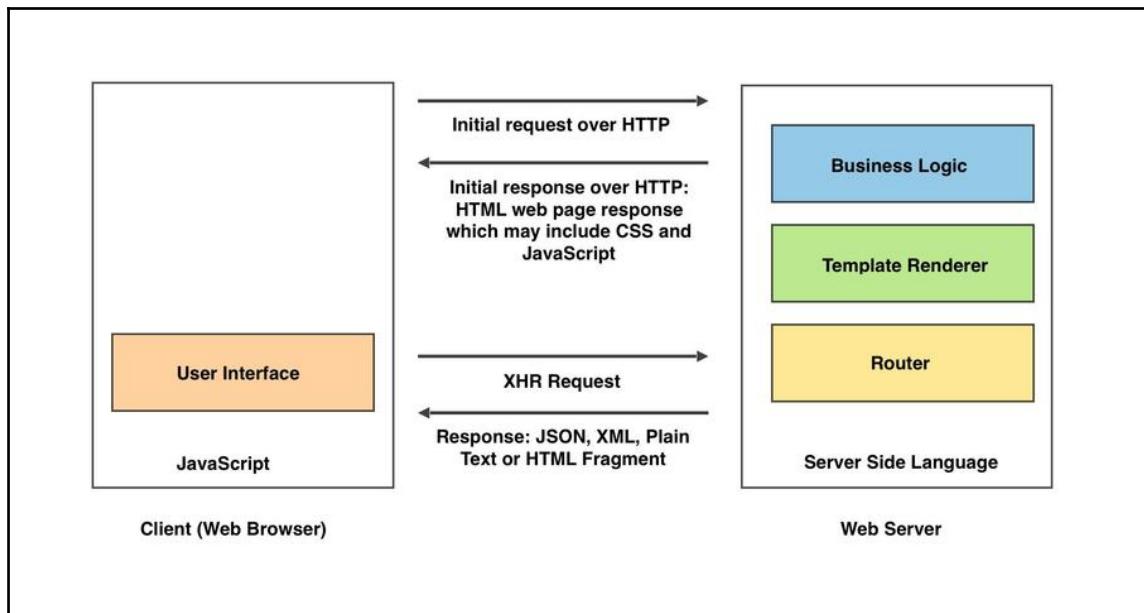


Figure 1.4: The AJAX web application architecture

Because the XHR calls are asynchronous in nature, they don't block the single threaded JavaScript application running in the web browser. Once a response is received from the server for a given XHR request, an action can be taken with the data that was returned from the server.

## The primary advantage

The primary advantage of the AJAX web application architecture is that it removes the need to perform a full page reload.

In the scenario that we considered with the news article web page that had 10,000+ comments, we can program the web application to initiate an XHR call when the **Next** button is pressed, then the server can send back an HTML fragment that contains the next set of comments to display. Once we get back the next set of comments, we can have JavaScript dynamically update the DOM, and completely avoid the need to perform a full page reload!

Figure 1.5 illustrates this approach. The left-most illustration depicts the comments in the comment section. The middle illustration depicts only the comments section being updated. Finally, the illustration on the right depicts the next batch of comments loaded in the comments section:

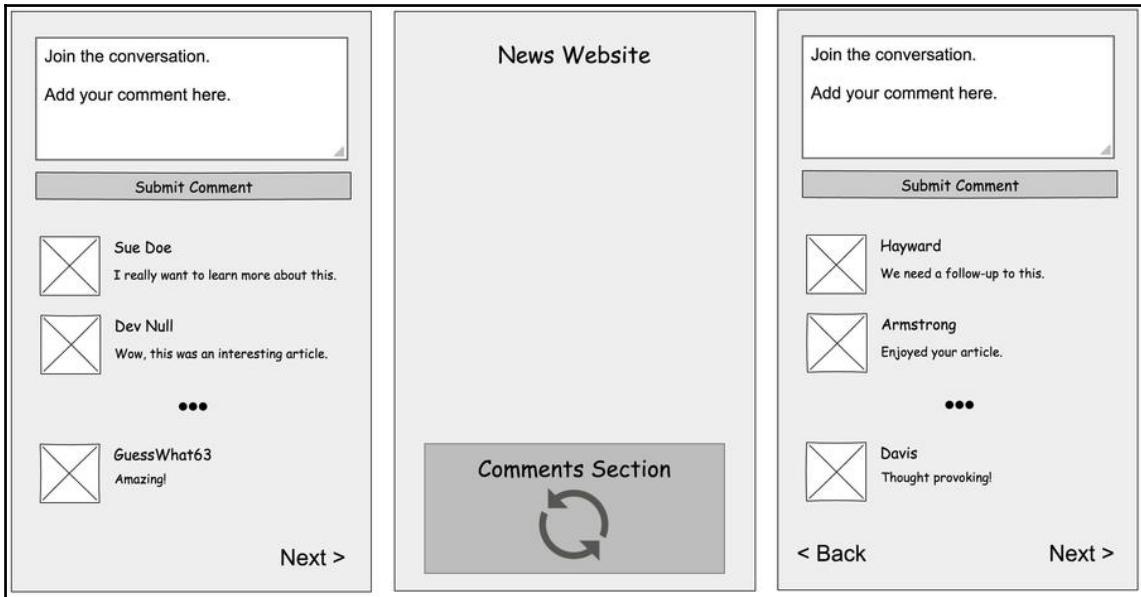


Figure 1.5: When the *Next* link is clicked, only the comments section of the news website is updated, avoiding a full page refresh

As you can see, the primary advantage of this approach is that we avoid the full page reload, which enhances the user experience. Keep in mind that in certain scenarios, such as navigating through different sections of the website, full page reloads may still occur.

## Disadvantages

The AJAX web application architecture comes with the following disadvantages:

- Handling the mental context switch between two programming languages
- The complexity introduced by performing piecemeal client-side rendering
- The duplication of efforts

## Mental context shifts

When it comes to developer productivity, we have now introduced a mental context shift (also known as a cognitive switch) assuming that the back-end server-side language is not JavaScript. For example, let's consider that our back-end application is implemented in Go and the front-end application is implemented in JavaScript. Now, the developer will have to be fluent in both the server-side language (Go) and the client-side language (JavaScript) which apart from syntactical differences may have a different set of guiding philosophies and idioms.

This causes a mental context shift for the full stack developer that is tasked with maintaining the client side and the server side of the codebase. One way for organizations to immediately address the issue of mental context shifts is to reach into the pocketbooks. If the organization can afford to do this, it could take the hit in increased operating costs and dedicate at least one developer to the front-end and one developer to the back-end.



**Wish list item #2:** To increase maintainability, there should be a single, unified, project codebase, which is implemented in a single programming language.

## Increased rendering complexity

In addition to introducing the mental context shift of handling two different programming languages, we have now increased the level of rendering complexity. In the classic web application architecture, the rendered web page that was received from the server response was never mutated. In fact, it was wiped out once a new page request was initiated.

Now, we are re-rendering portions of the web page in a piecemeal fashion from the client side, which requires us to implement more logic to make (and keep track of) subsequent updates to the web page.



**Wish list item #3:** To increase efficiency, there should be a mechanism to perform distributed template rendering.

## Duplication of efforts

The AJAX web application architecture introduces a duplication of efforts between the server side and the client side. Let's say that we wanted to add a new comment to the news article. Once we fill out the form, to add the new comment, we can initiate an XHR call, which will send the new comment, that is to be added, to the server. The server-side web application can then persist the new comment to the database, where all comments are stored. Instead of refreshing the entire web page, we can immediately update the comment section to include the new comment that was just added.

A basic tenet in computer programming, and especially in web programming, is to never trust user input. Let's consider the scenario where the user may have introduced a set of invalid characters into the comment box. We will have to implement some type of validation that checks the user's comment, both on the client side and on the server side. This means that we'll have to implement client-side form validation in JavaScript and server-side form validation in Go.

At this point, we have introduced a duplication of efforts in two programming languages spread across two different operating environments. Besides the example we just considered, there may be other scenarios that require duplication of efforts when going down this architectural path. This happens to be a major disadvantage of the AJAX web application architecture.



**Wish list item #4:** To increase productivity, there should be a means to share and reuse code across environments to avoid the duplication of efforts.

## The single page application (SPA) architecture

In 2004, the **World Wide Web Consortium (W3C)** started working on the new HTML standard, which was to be the precursor to HTML5. In 2010, HTML5 started to pick up speed, and features from the specification started to make their way into the major web browsers and the HTML5 functionality became very popular.

The major selling point for HTML5 was to introduce capabilities that would allow web applications to behave more like native applications. A new set of APIs that were accessible through JavaScript were introduced. These APIs included the functionality to store data locally on the user's device, better control of the forward and back button (using the web browser's History API), a 2D canvas for rendering graphics, and the second version of the XHR object that included greater capabilities than its predecessor:

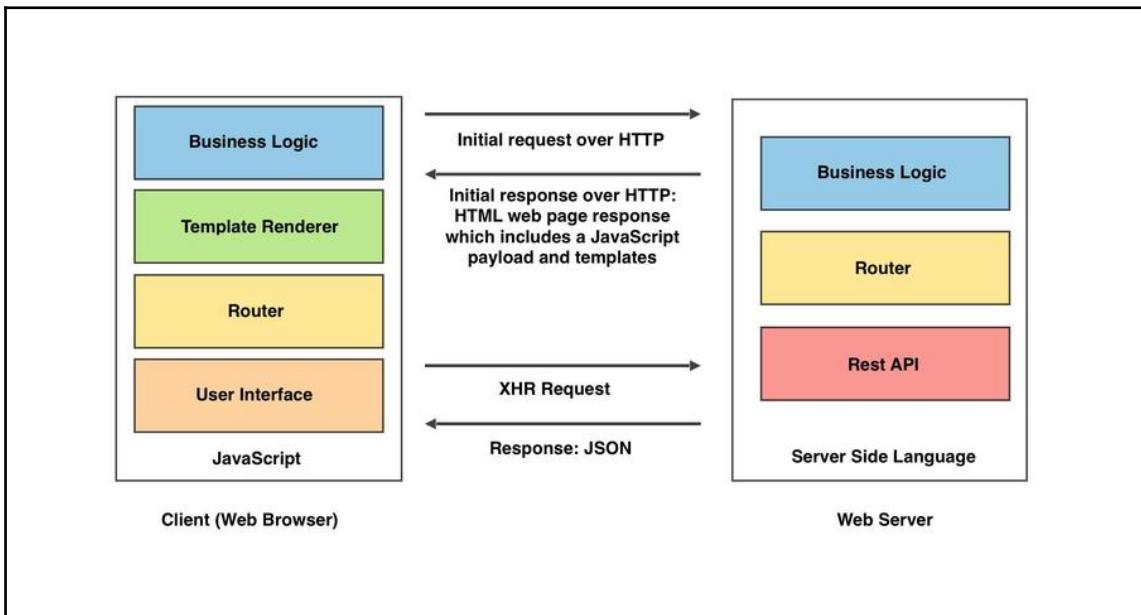


Figure 1.6: The Single Page Application (SPA) Architecture

In the early 2010s, JavaScript frameworks began to emerge, which facilitated in the development of a new type of architecture, the SPA architecture. This architecture, as depicted in *Figure 1.6*, focuses on a *fat client* and *thin server* strategy. The idea was to remove the responsibility of any type of template rendering from the server side and assign all **User Interface**(UI) rendering to the client side. In this architecture, there is a clear separation of concerns between the duties of the server and the client.

The SPA architecture removes the duplication of efforts for user interface responsibilities. It does so by consolidating all UI code to the client. Doing so eliminates the duplication of efforts on the server side in terms of the user interface. As depicted in *Figure 1.6*, the responsibility for the user interface rests solely with the client.

The server initially returns a payload containing JavaScript and client-side templates. The JavaScript payload could possibly be *aggregated*, which means that all JavaScript source files that comprise the web application can be combined into one JavaScript source file. In addition to that, the JavaScript payload might also be **minified**.



**Minification** is the process of removing any unnecessary characters from the source code, which may include renaming identifiers in the source code without changing the functionality of the source code, in order to reduce its storage footprint.

Once the web browser has fully downloaded the JavaScript payload, the first order of business for the JavaScript code is to bootstrap the JavaScript application, rendering the user interface on the client side.

## The primary advantage

The primary advantage of the SPA architecture is that it provides client-side routing, preventing full page reloads. Client-side routing involves intercepting the click event of hyperlinks on a given web page, so that they don't initiate a new HTTP request to the web server. The client-side router associates a given route with a client-side route handler that is responsible for servicing the route.

For example, let's consider an e-commerce website that has implemented client-side routing. When a user clicks on a link to the Swiss Army Knife product detail page, instead of initiating a full page reload, an XHR call is made to a REST API endpoint on the web server. The endpoint returns the profile data about the Swiss Army knife, in the **JavaScript Object Notation (JSON)** format, which is used by the client-side application to render the content for the Swiss Army knife product detail page.

The experience is seamless from the user's perspective, since the user will not experience the sudden white flash that is encountered on a full page reload.

## Disadvantages

The SPA architecture comes with the following disadvantages:

- The initial page loads are perceived to be slower
- Reduced search engine discoverability

## Slower initial page loads

The initial page load of an SPA-based web application can be perceived to be slow. The slowness can result from the time-consuming, initial download of the aggregated JavaScript payload.

The **Transmission Control Protocol (TCP)** has a slow start mechanism, where data is sent in segments. The JavaScript payload will require multiple round trips between the server and the client before it can be fully delivered to the web browser:

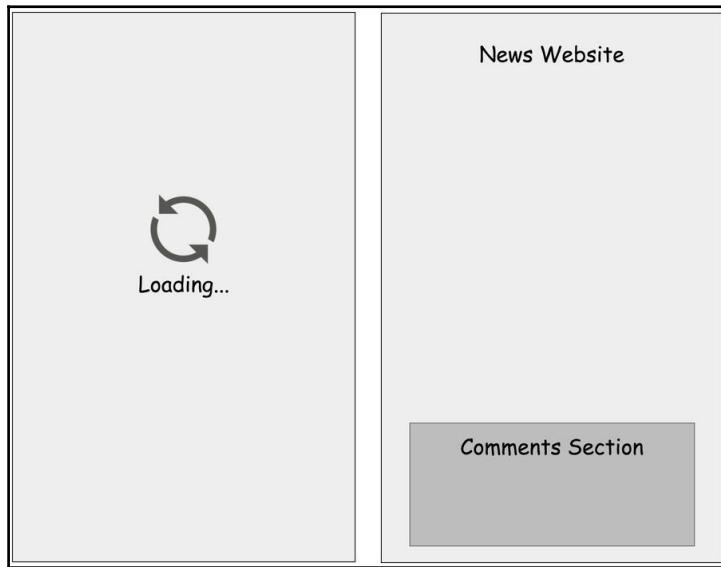


Figure 1.7: The initial page load is perceived to be slow since the user is greeted with a loading indicator instead of the rendered web page

A consequence of this is that users have to wait for the JavaScript payload to be completely fetched before the web page can be fully rendered. It is a common **user experience (UX)** practice to use a loading indicator (such as a spinning wheel) to let the user know that the user interface is still loading.

*Figure 1.7* includes an illustration (on the left) that depicts the loading indicator, and an illustration (on the right) that depicts the layout of the loaded web page. It is important to note that, depending on the SPA implementation, there may be more than one loading indicator spread across the individual sections that make up the web page.

I'm sure that, in your own web browsing travels, you have probably used web applications that have contained these loading spinners. We can agree, from the user's perspective, that ideally we would rather want to see the rendered output instead of the spinning wheel.



**Wish list item #5:** To make the best first impression, the website should readily display content to the user .

## Reduced search engine discoverability

The use of the SPA architecture may reduce search engine discoverability. Because of the dynamic nature of rendering content on the client side, some SPA implementations may not produce well-formed HTML content that can be easily consumed by search engine bot crawlers that are used to consuming an initial web page response only.

The search engine bot crawler may not have the capability to render the web page, since it may not be equipped with a JavaScript runtime. Without the fully rendered web page content, the bot crawler cannot effectively perform its duty of consuming the web page's content.

In addition to this, SPA implementations handle routes using fragment identifiers, strings that refer to a resource, after the hash mark (#) of a URL. This approach is not search engine friendly.

Let's return to our e-commerce web application example. In the classic and AJAX web application architectures, our web application could have the following URL:  
`http://igweb.kamesh.com/product-detail/swiss-army-knife.`

In the case of the SPA implementation, the URL, with a fragment identifier, could look like this:

`http://igweb.kamesh.com/#section=product_detail&product=swiss-army-knife`

This URL would be difficult for a search engine bot crawler to index because the fragment identifier (the characters after the hash symbol) is meant to specify a location within a given web page.

Fragment identifiers were designed to provide links within sections of an individual web page. The fragment identifier influences the web browser's history since we can tack on unique identifiers to the URL. This effectively, prevents the user from encountering a full page reload.

The shortcoming of this approach is that fragment identifiers are not included in the HTTP request, so from a web server's perspective, the URL, `http://igweb.kamesh.com/webapp#orange`, and the URL, `http://igweb.kamesh.com/webapp#apple`, are pointing to the same resource: `http://igweb.kamesh.com/webapp`.

The search engine bot crawler would have to be implemented in a more sophisticated manner to handle the complexity of indexing websites containing fragment identifiers. Although Google has made considerable progress on this problem, implementing URLs, without the fragment identifiers, is still the recommended best practice to ensure websites are easily indexed by search engines.

It is important to note is that in some cases, the SPA architecture may overcome this disadvantage, using more modern practices. For example, more recent SPA implementations avoid fragment identifiers altogether, using the web browser's History API to have more search engine friendly URLs.



**Wish list item #6:** To promote discoverability, the website should provide well-formed HTML content that is easily consumed by search engine bots. The website should also contain links that are easily indexed by search engine bots.

# The isomorphic web application architecture

The **isomorphic web application architecture** consists of implementing two web applications, one on the server side and one on the client side, using the same programming language and reusing code across the two environments:

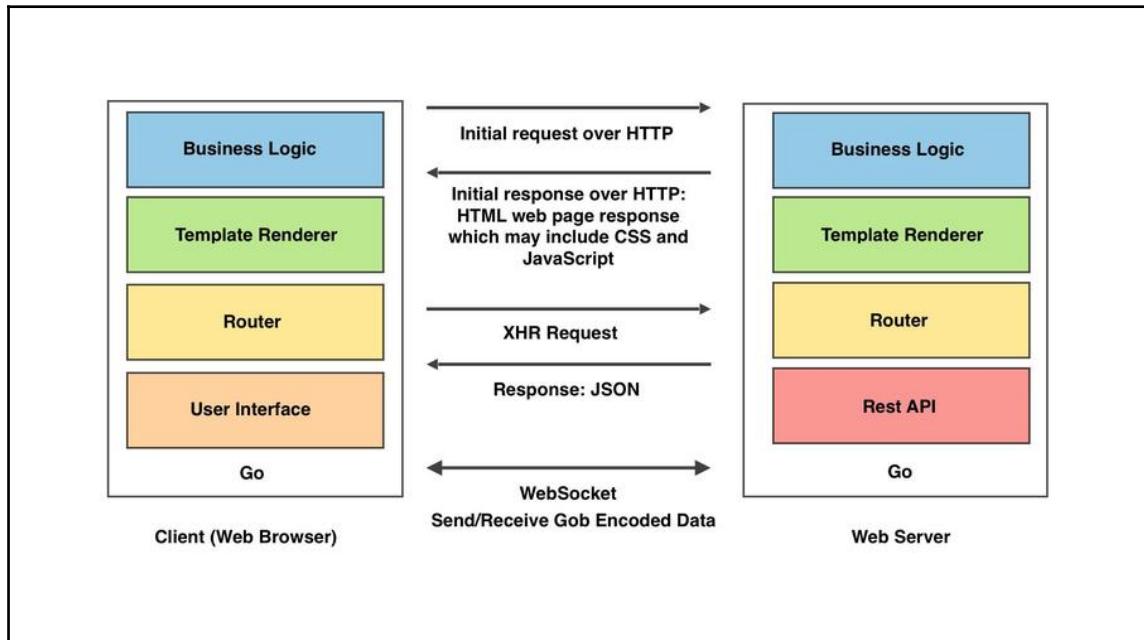


Figure 1.8: The Isomorphic Web Application Architecture

As depicted in *Figure 1.8*, business logic can be shared across environments. For example, if we had defined a `Product` struct to model a product for our e-commerce website, both the server-side and client-side applications can be made aware of it.

In addition to this, a template renderer exists on both the server side and the client side, so that templates can also be rendered across environments, making templates *isomorphic*.



The term *isomorphic* can be used to describe anything (business logic, templates, template functions, and validation logic) that can be shared across environments.

The server-side route handler is responsible for servicing routes on the server side and the client-side route handler is responsible for servicing routes on the client side. When a user initially accesses a website implemented using isomorphic web application architecture, the server-side route handler kicks in and generates a web page response using the server-side template renderer.

Subsequent user interactions with the website are performed in the SPA mode using client-side routing. The client-side route handler is responsible for servicing a given client-side route and rendering content to the web page (the user interface) using the client-side template renderer.

The client-side application can initiate a XHR request to a Rest API endpoint on the web server, retrieve data from the server's response, and render content on the web page using the client-side template renderer.

An Isomorphic Go web application may optionally utilize a WebSocket connection, as shown in *Figure 1.8*, for persistent, bidirectional communication between the web server and the web browser. Isomorphic Go web applications have the added benefit of sending and receiving data in the `gob` format—Go's format for binary encoded data. Encoding and decoding data to the `gob` format can be done using the `encoding/gob` package from the standard library.



Gob encoded data has a major advantage over JSON—it has a smaller data storage footprint.

The primary advantage of the `gob` format is its lower storage footprint. JSON data is in text format, and it's understood that data formatted as text requires a heavier storage footprint when compared with a binary encoded format. With smaller data payloads exchanged between the client and server, the web application can benefit with faster response times when transferring data.

## Wish list fulfilled

The Isomorphic Web Application Architecture offers a solution for all of the disadvantages found in the three traditional web application architectures. Let's take stock of the items that we've placed on our wish list:

1. To **enhance the user experience**, clicking a link on the website should not cause a full page reload.

2. To **increase maintainability**, there should be a single, unified, project codebase that is implemented in a single programming language.
3. To **increase efficiency**, there should be a mechanism to perform distributed template rendering.
4. To **increase productivity**, there should be a means to share and reuse code across environments, to avoid the duplication of efforts.
5. To **make the best first impression**, the website should readily display content to the user.
6. To **promote discoverability**, the website should provide well-formed HTML content that is easily consumed by search engine bots. The website should also contain links that are easily indexed by search engine bots.

Now, it's time to examine how the isomorphic web application architecture fulfills each item that has been placed on our wish list.

## 1. Enhancing the user experience

After the initial server-side rendered web page response, the isomorphic web application architecture enhances the user experience by running in the SPA mode. Client-side routing is used for subsequent user interactions with the website, preventing full page reloads and enhancing the user experience of the website.

## 2. Increasing the maintainability

Maintainability of the project codebase is strengthened by the isomorphic web application architecture due to the fact that a single programming language is used to implement both the client-side and server-side web applications. This prevents the mental context shifts that occur when dealing with two different programming languages across environments.

## 3. Increasing the efficiency

The isomorphic web application architecture increases the efficiency of rendering content by providing a mechanism for distributed template rendering—the isomorphic template renderer. With a template renderer present on both the server side and the client side, as depicted in *Figure 1.8*, templates can easily be reused across environments.

## 4. Increasing the productivity

The single unified codebase that is the hallmark of the isomorphic web application architecture provides many opportunities to share code across environments. For example, form validation logic can be shared across environments, allowing a web form to be validated, both on the client side and the server side using the same validation logic. It is also possible to share models and templates across the client and the server.

## 5. Making the best first impression

The isomorphic web application architecture's usage of server-side rendering for the initial web page response, guarantees that the user will see content immediately upon accessing the website. For the first encounter with the user, the isomorphic web application architecture takes a page out of the classic web application architecture's playbook for providing the initial web page response.

This is a welcome benefit to the user, since content is displayed to them instantly and the user will perceive a fast page load as a result of this. This is a sharp contrast to the SPA architecture, where the user would have to wait for the client-side application to bootstrap before seeing the web page's content appear on the screen.

## 6. Promoting discoverability

The isomorphic web application architecture promotes discoverability, since it can easily provide well-formed HTML content. Keep in mind that the rendered output of Go templates is HTML.

With an isomorphic template renderer, HTML content is easily rendered on the client side and the server side. This means that we can provide well-formed HTML content for traditional search engine bot crawlers that simply scrape web page content, as well as for modern search engine bot crawlers that may be equipped with a JavaScript runtime.

Another means by which the isomorphic web application architecture promotes discoverability is that well-formed URLs can be defined by the application's route handlers (both on the server side and client side) and these URLs can easily be indexed by search engine bot crawlers.

This is possible because the route handler implemented on the client side makes use of the web browser's History API to match the same routes that are defined on the server side. For example, the `/product-detail/swiss-army-knife` route for the Swiss Army Knife product detail page can be registered by both the server-side and the client-side routers.

## Live demo

Now it's time to see the isomorphic web application architecture in action. A live demo of IGWEB, the website that we will be implementing over the course of this book, is available at <http://igweb.kamesh.com>. *Figure 1.9* is a screenshot of the website's home page:

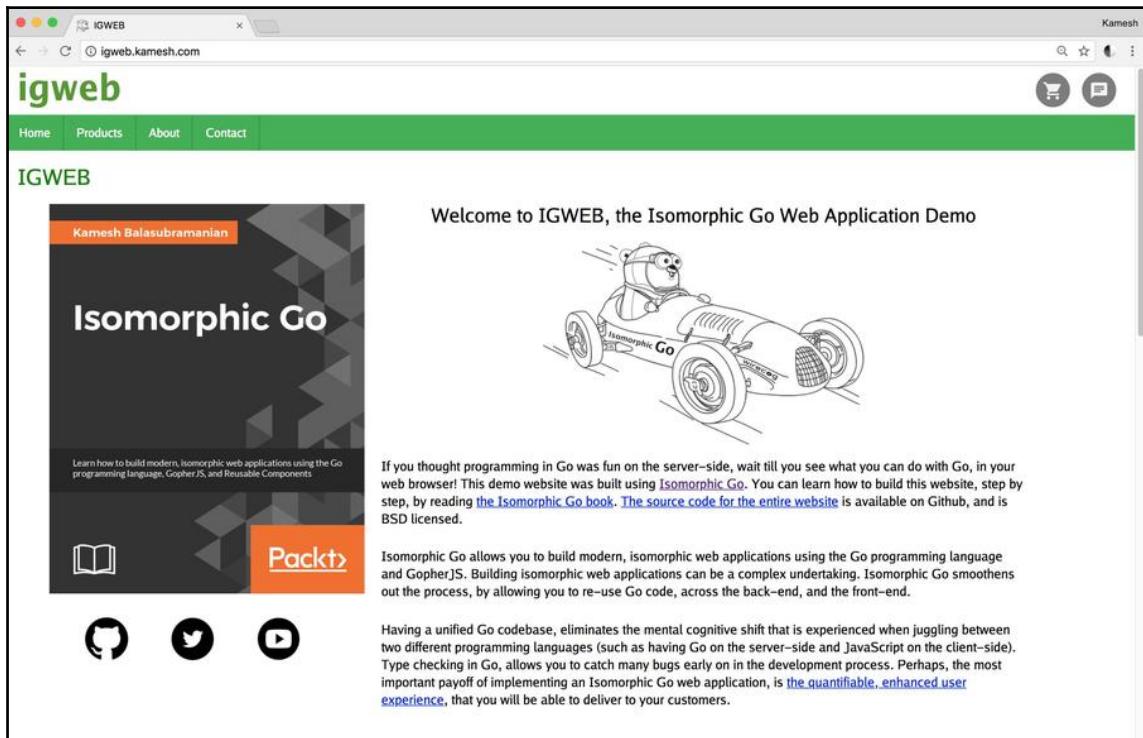


Figure 1.9: IGWEB: A website implemented with Isomorphic Go

Notice that the content in the *above the fold* area (the area that is visible in the browser window) is displayed instantly. Also, take note of the responsiveness of the website when navigating to different sections of the website by clicking on the links in the navigation menu. We'll provide you with a detailed introduction to the IGWEB project in the next chapter.



At the time of writing, IGWEB has been verified to function in the following web browsers: Google Chrome version 62.0, Apple Safari version 9.1.1, Mozilla Firefox 57.0, and Microsoft Edge 15.0. It is recommended that you use a web browser that has the same version, or above the version, provided in this list.

## Measurable benefits

The methodology to develop an isomorphic web application using Go, that is presented in this book, has proven, measurable benefits with regard to providing an enhanced user experience.

We can use the Google PageSpeed Insights tool (<https://developers.google.com/speed/pagespeed/insights/>) to evaluate the performance of IGWEB's home page. The tool measures how well a web page delivers a good user experience, on a scale of 0 to 100, based on various criteria, namely the organization of web page content, size of static assets, and time taken to render the web page:

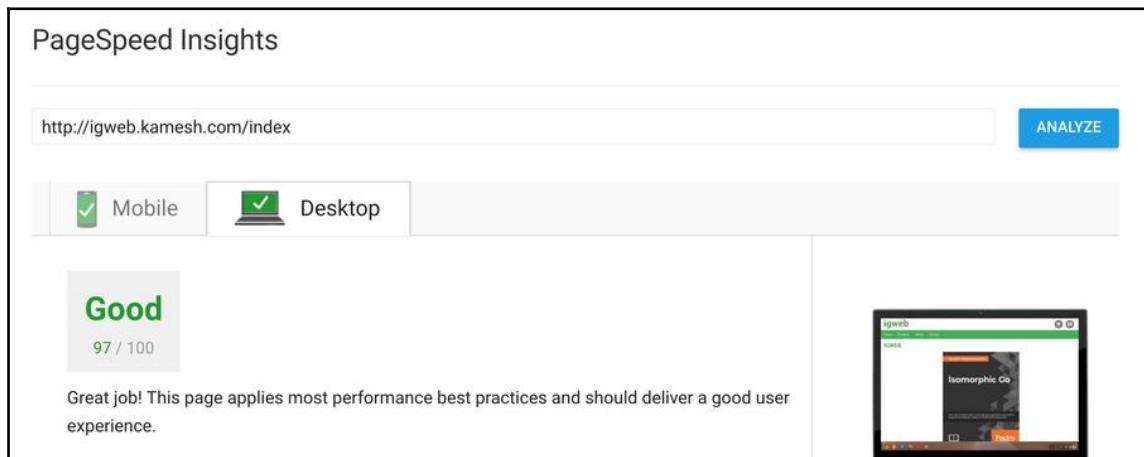


Figure 1.10: The result of running the IGWEB home page through the Google PageSpeed Insights tool

Figure 1.10 is a screenshot that shows the result of evaluating the desktop edition of IGWEB. At the time of writing, IGWEB scores 97/100 for the desktop browsing experience, and 91/100 for the mobile browsing experience. According to the tool, the 90+ score attained for both the desktop and mobile editions indicates that the IGWEB home page *applies most performance best practices and should deliver a good user experience*.

## Nomenclature

I used the term *Isomorphic Go* as the title for my introductory presentation on the subject of developing isomorphic web applications in Go at **GopherCon India**. The title of my presentation was inspired by the term *Isomorphic JavaScript*. The term *Isomorphic JavaScript* was coined by Charlie Robbins in his 2011 blog post (<https://blog.nodejitsu.com/scaling-isomorphic-javascript-code/>), *Scaling Isomorphic JavaScript Code*.



The word *isomorphism* comes from mathematics. In Greek, *iso* means equal and *morphosis* means to form or to shape.

A debate has existed within the JavaScript community on the usage of the term *isomorphic* to describe a web application that contains code that can run on either the client or on the server. Some members of the JavaScript community prefer using the term *universal* instead.

In my opinion, the term *isomorphic* is more appropriate, while the term *universal* introduces ambiguity. The ambiguity stems from the fact that the term *universal* carries some baggage along with it.

Apple has widely used the term *universal binary* to describe fat binaries that contain machine code for multiple processor architectures. Modern JavaScript code gets compiled into machine code by a just-in-time compiler.

Therefore, using the term *universal* is ambiguous, and requires extra detail, to determine the context in which it is used. For this reason, the preferred term that will be used in this book is *isomorphic*.

## Prerequisites

This book focuses on teaching you how to create an isomorphic web application using the Go programming language. Since we will be taking an idiomatic approach that focuses exclusively on Go, it is not necessary to have prior familiarity with libraries and tools from the JavaScript ecosystem.

We assume that the reader has some level of prior programming experience in Go, or some other server-side programming language.

If you have never programmed in Go, I would recommend that you refer to *A Tour of Go* available at: <https://tour.golang.org>.

For a more in-depth study of fundamental Go concepts, I would recommend that you take my video course, *Go Essentials For Full Stack Web Development*, *Packt Publishing*, available at <https://www.packtpub.com/web-development/go-essentials-full-stack-web-development-video>.

## Summary

In this chapter, we provided an introduction to Isomorphic Go. We covered the many advantages that the Go programming language provides, and why it makes a compelling choice for the creation of isomorphic web applications.

We reviewed the traditional web application architectures, which included the classic web application architecture, the AJAX application architecture, and the SPA architecture. We identified the advantages and disadvantages of each traditional architecture. We introduced the isomorphic web application architecture and presented how it solved all the shortcomings of the traditional architectures.

We presented a live demo of IGWEB, an Isomorphic Go website, and introduced you to the Google PageSpeed Insight tool to measure web page performance. Finally, we provided you with some background on the term *isomorphic* and the items that you need to know to make the most out of understanding the material covered in this book.

In Chapter 2, *The Isomorphic Go Toolchain*, we will introduce you to the key technologies used to develop Isomorphic Go web applications. We will also introduce you to IGWEB, the Isomorphic Go website, that we will be building over the course of this book.

# 2

## The Isomorphic Go Toolchain

In the previous chapter, we established the many benefits that the isomorphic web application architecture provides and the advantages of using the Go programming language to build isomorphic web applications. Now, it's time to explore the essential ingredients that are needed to make Isomorphic Go web applications possible.

In this chapter, we will introduce you to the *Isomorphic Go* toolchain. We will examine the key technologies that comprise the toolchain—Go, GopherJS, the Isomorphic Go toolkit, and the UX toolkit. Once we have established how to obtain and ready these tools, we will install the IGWEB demo—the Isomorphic Go web application that we will implement in this book. Later, we will dive into the anatomy of the IGWEB demo, examining the project structure and code organization.

We will also introduce you to some helpful and productive techniques that will be used throughout the book, such as implementing a custom datastore on the server side to serve our web application's data persistence needs and utilizing dependency injections to provide commonly used functionality throughout our web application. Finally, we will provide a project roadmap for the IGWEB application to map out our journey in building an Isomorphic Go web application.

In this chapter, we will cover the following topics:

- Installing the Isomorphic Go toolchain
- Setting up the IGWEB demo
- An introduction to the IGWEB demo
- Project structure and code organization

# Installing the Isomorphic Go toolchain

In this section, we will guide you through the process of installing and configuring the Isomorphic Go toolchain—the set of technologies that allow us to create Isomorphic Go web applications. Here are the key technologies that we will be covering:

- Go
- GopherJS
- The Isomorphic Go toolkit
- The UX toolkit

We will utilize **Go** as the server-side and client-side programming language for our web application. Go allows us to create reliable and efficient software using a simple and understandable syntax. It's a modern programming language that is designed in an age of multicore processors, networked systems, massive computation clusters, and the World Wide Web. Since Go is a general purpose programming language, it makes for an ideal technology to create isomorphic web applications.

**GopherJS** allows us to bring Go to the client side by transpiling Go code into pure JavaScript code that can run in all major web browsers. GopherJS bindings are available for common JavaScript functionality, including the DOM API, XHR, built-in JavaScript functions/operators, and the WebSocket API.

The **Isomorphic Go toolkit** provides us with the technology we need to build Isomorphic Go web applications. Using the tools available from this project, we can implement the common functionality required in an isomorphic web application, such as client-side routing, isomorphic template rendering, and creating isomorphic web forms.

The **UX toolkit** provides us with the capability to create reusable components in Go, which are known as **cogs**. You can think of them as self-contained user interface widgets that promote reusability. Cogs can be implemented as pure Go cogs or hybrid cogs that can tap into the existing JavaScript functionality. Cogs are registered on the server side, and deployed on the client side.

Figure 2.1 depicts the technology stack we'll use as a Venn diagram, clearly indicating the environment (or environments) the technology component will reside in:

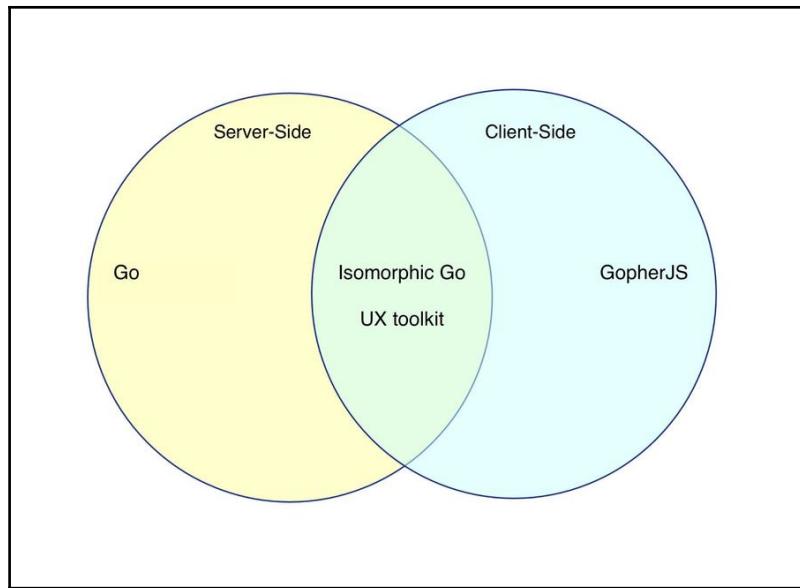


Figure 2.1: The Isomorphic Go toolchain: Go, GopherJS, the Isomorphic Go toolkit, and the UX toolkit

Now that we have identified the key components that comprise our technology stack, let's go ahead and install/configure them.

## Go

If you are new to Go, it is well worth your time to undertake tour of Go, available at <https://tour.golang.org>.

Before you can proceed further, you need to have Go installed on your system. In this section, we will provide a high-level overview of installing Go, and setting up your Go workspace. If you need further help, you can access the detailed instructions to install Go at <https://golang.org/doc/install>.

Let's make our way to the Go website, available at <https://golang.org>:

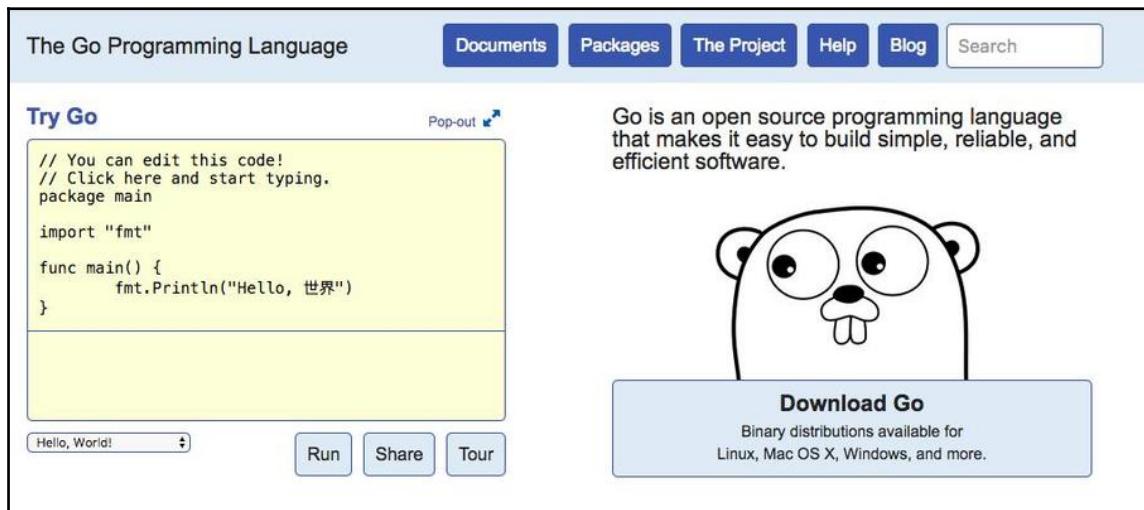


Figure 2.2: The Go Website

Click on the **Download Go** link, shown in *Figure 2.2*, to get to the Downloads page (<https://golang.org/dl/>), shown in *Figure 2.3*:



Figure 2.3: The Downloads page on the Go Website

As you can see, Go is available for all the major operating systems. We will be using a Mac while walking you through the installation and configuration process. Information on installing Go for other operating systems can be found in the *Getting Started* document available on the Go website, at <https://golang.org/doc/install>.

On the **Downloads** page, click on the link to download the distribution of Go for your operating system. I clicked on the link to download the **Apple macOS** installer.

Getting your system up and running with Go will consist of the following steps:

1. Installing Go
2. Setting up your Go workspace
3. Building and running programs

## Installing Go

After the Download is complete, go ahead and launch the installer. The Go installer is shown in *Figure 2.4*:

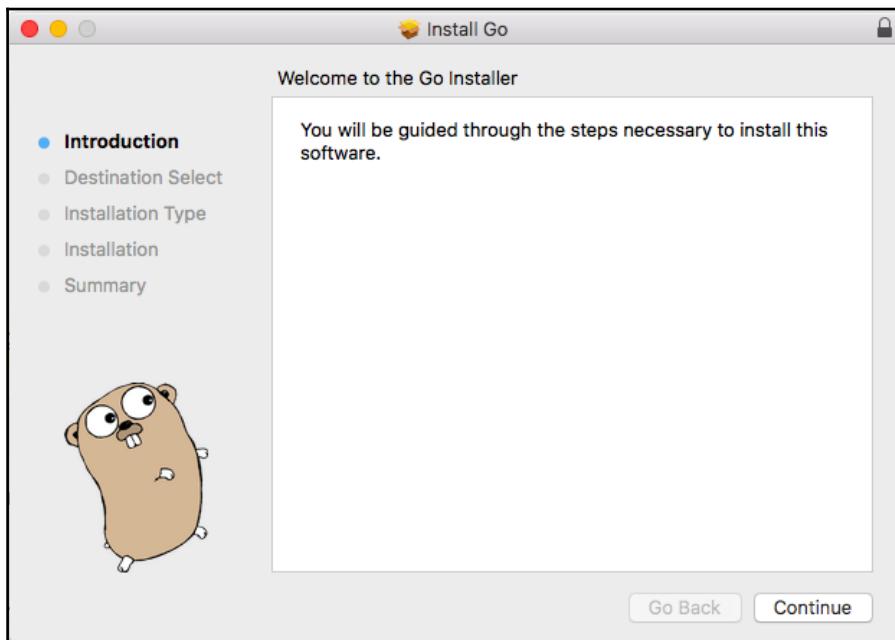


Figure 2.4: The Go installer

Follow the onscreen prompts of the installer, and if the installer asks you to make Go available for all users on the system, make sure that you choose to install Go for all users of the system. You may also be prompted for your system credentials (so that you may install Go for all users on the system). Again, go ahead and provide your system credentials.

Once the installer is complete, you should get the following confirmation from the Go installer:

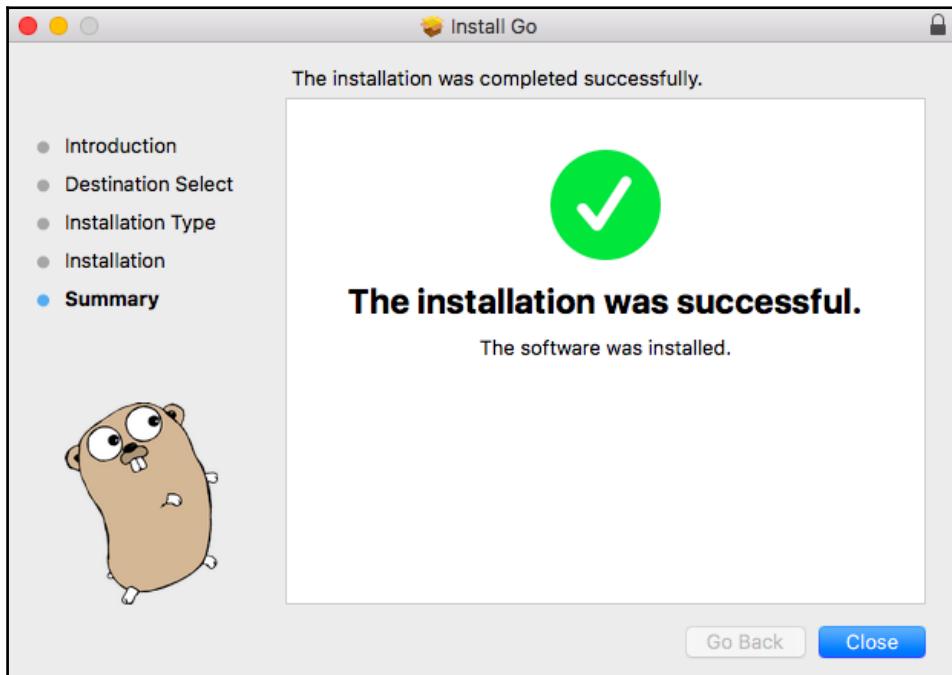


Figure 2.5: The Go installer reports a successful installation

Once the installer is done, let's open up Command Prompt and inspect where the installer installed the files:

```
$ which go  
/usr/local/go/bin/go
```

On a macOS system, the Go distribution gets installed to the `/usr/local/go` directory, and the binaries that come with the Go distribution are installed in the `/usr/local/go/bin` directory.

If you are new to the Go toolchain, you should use the `go help` command to get yourself acquainted with the various commands that come with Go:

```
$ go help
Go is a tool for managing Go source code.
```

**Usage:**

```
go command [arguments]
```

**The commands are:**

<code>build</code>	compile packages and dependencies
<code>clean</code>	remove object files
<code>doc</code>	show documentation for package or symbol
<code>env</code>	print Go environment information
<code>bug</code>	start a bug report
<code>fix</code>	run go tool fix on packages
<code>fmt</code>	run gofmt on package sources
<code>generate</code>	generate Go files by processing source
<code>get</code>	download and install packages and dependencies
<code>install</code>	compile and install packages and dependencies
<code>list</code>	list packages
<code>run</code>	compile and run Go program
<code>test</code>	test packages
<code>tool</code>	run specified go tool
<code>version</code>	print Go version
<code>vet</code>	run go tool vet on packages

`Use "go help [command]" for more information about a command.`

**Additional help topics:**

<code>c</code>	calling between Go and C
<code>buildmode</code>	description of build modes
<code>filetype</code>	file types
<code>gopath</code>	GOPATH environment variable
<code>environment</code>	environment variables
<code>importpath</code>	import path syntax
<code>packages</code>	description of package lists
<code>testflag</code>	description of testing flags
<code>testfunc</code>	description of testing functions

`Use "go help [topic]" for more information about that topic.`

To ascertain the version of Go installed on your system, you can use the `go version` command:

```
$ go version
go version go1.9.1 darwin/amd64
```



You should have the latest version of Go installed on your system, and you need to have a properly configured Go workspace before you can proceed further.

## Setting up your Go workspace

Now that you've successfully installed Go on your system, you need to have a properly configured Go workspace before you can proceed further. We will provide a high-level overview on setting up a Go workspace, and if you need further help, you may read the detailed instructions on setting up a Go workspace available at the Go website: <https://golang.org/doc/code.html>.

Use your favorite text editor to open up the `.profile` file in your home directory. If you are using Linux, you need to open up the `.bashrc` file found in your home directory.

We are going to add the following lines to the file to add some very important environment variables:

```
export GOROOT=/usr/local/go
export GOPATH=/Users/kamesh/go
export GOBIN=${GOPATH}/bin
export PATH=${PATH}:/usr/local/bin:${GOROOT}/bin:${GOBIN}
```



My username is `kamesh`, you will have to obviously replace this with your username.

`$GOROOT` is an environment variable used to specify where the Go distribution is installed on the system.

`$GOPATH` is an environment variable used to specify the top-level directory containing the source code for all our Go projects. This directory is known as our Go workspace. I have created my workspace in my home directory in the `go` folder: `/Users/kamesh/go`.

Let's go ahead and create our Go workspace along with three important directories inside of it:

```
$ mkdir go
$ mkdir go/src
$ mkdir go/pkg
$ mkdir go/bin
```

The `go/src` directory will contain the Go source files. The `go/pkg` directory will contain the compiled Go packages. Finally, the `go/bin` directory will contain compiled Go binaries.

`$GOBIN` is an environment variable used to specify the location where Go should install compiled binaries. When we run the `go install` command, Go compiles our source code and stores the newly created binary in the directory specified by `$GOBIN`.

We include two additional entries to the `$PATH` environment variable—the `$GOROOT/bin` and `$GOBIN` directories. This tells our shell environment where to look to find Go-related binaries. Tacking on `$GOROOT/bin` to the `$PATH` lets the shell environment know where binaries for the Go distribution are located. Tacking on `$GOBIN` tells the shell environment where the binaries for the Go programs we create will exist.

## Building and running Go programs

Let's create a simple "hello world" program to check our Go setup.

We start out by creating a directory for our new program inside the `src` directory of our Go workspace, as shown here:

```
$ cd $GOPATH/src
$ mkdir hellogopher
```

Now, using your favorite text editor, let's create a `hellogopher.go` source file in the `hellogopher` directory with the following contents:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello Gopher!")
}
```

To build and run this program in one step, you can issue the `go run` command:

```
$ go run hellogopher.go
Hello Gopher!
```

To produce a binary executable file that will exist in the current directory, you can issue the `go build` command:

```
$ go build
```

To build a binary executable and automatically move it to your `$GOBIN` directory, you can issue the `go install` command:

```
$ go install
```

After issuing the `go install` command, you simply have to type the following command to run it (provided that `$GOBIN` is specified in your `$PATH`):

```
$ hellogopher
Hello Gopher!
```

At this point, we have successfully installed, configured, and verified the Go installation. Now it's time to get the other tools up and running, starting with GopherJS.

## GopherJS

GopherJS is a transpiler that converts Go code into pure JavaScript code. Using GopherJS, we can write the front-end code in Go that will work across all major web browsers that support JavaScript. This technology allows us to unleash the power of Go inside the web browser, and without it, Isomorphic Go would not be possible.

In this chapter, we will show you how to install GopherJS. We will cover GopherJS in greater detail in [Chapter 3, Go on the Front-End with GopherJS](#).

Getting up and running with GopherJS consists of the following steps:

1. Installing GopherJS
2. Installing essential GopherJS bindings
3. Getting familiar with GopherJS on the command line

## Installing GopherJS

We can install GopherJS by issuing the following `go get` command:

```
$ go get -u github.com/gopherjs/gopherjs
```

To find out the current version of `gopherjs` installed on your system, use the `gopherjs version` command:

```
$ gopherjs version
GopherJS 1.9-1
```



The major versions of Go and GopherJS must match on your system. We will be using version 1.9.1 of Go and version 1.9-1 of GopherJS in this book.

You can type `gopherjs help` to get yourself acquainted with the various commands that come with GopherJS:

```
$ gopherjs
GopherJS is a tool for compiling Go source code to JavaScript.

Usage:
  gopherjs [command]

Available Commands:
  build compile packages and dependencies
  doc display documentation for the requested, package, method or symbol
  get download and install packages and dependencies
  install compile and install packages and dependencies
  run compile and run Go program
  serve compile on-the-fly and serve
  test test packages
  version print GopherJS compiler version

Flags:
  --color colored output (default true)
  --localmap use local paths for sourcemap
  -m, --minify minify generated code
  -q, --quiet suppress non-fatal warnings
  --tags string a list of build tags to consider satisfied during the
  build
  -v, --verbose print the names of packages as they are compiled
  -w, --watch watch for changes to the source files

Use "gopherjs [command] --help" for more information about a command.
```

## Installing essential GopherJS bindings

Now that we've installed GopherJS and confirmed that it is working, we need to obtain the following GopherJS bindings, which are required for our front-end web application development needs:

- `dom`
- `jsbuiltin`
- `xhr`
- `websocket`

### **dom**

The `dom` package provides us with GopherJS bindings for JavaScript's DOM APIs.

We can install the `dom` package by issuing the following command:

```
$ go get honnef.co/go/js/dom
```

### **jsbuiltin**

The `jsbuiltin` package provides bindings for common JavaScript operators and functions. We can install the `jsbuiltin` package by issuing the following command:

```
$ go get -u -d -tags=js github.com/gopherjs/jsbuiltin
```

### **xhr**

The `xhr` package provides bindings for the `XMLHttpRequest` object. We can install the `xhr` package by issuing the following command:

```
$ go get -u honnef.co/go/js/xhr
```

## websocket

The `websocket` package provides bindings for the web browser's WebSocket API. We can install the `websocket` package by issuing the following command:

```
$ go get -u github.com/gopherjs/websocket
```

## Getting familiar with GopherJS on the command line

The `gopherjs` command is very similar to the `go` command. For example, to transpile a Go program into its JavaScript representation, we issue the `gopherjs build` command like this:

```
$ gopherjs build
```

To build a GopherJS project and minify the produced JavaScript source file, we specify the `-m` flag along with the `gopherjs build` command:

```
$ gopherjs build -m
```

When we perform a build operation, GopherJS will create both a `.js` source file and a `.js.map` source file.



The `.js.map` files are called source maps. They help us map a minified JavaScript source file back to its unbuilt state. This feature comes in handy when we chase down errors using the web browser console.

The JavaScript source file, generated by GopherJS, can be imported as an external JavaScript source file into a web page using the `script` tag.

## The Isomorphic Go toolkit

The Isomorphic Go toolkit (<http://isomorphicgo.org>) provides us with the technology needed to implement Isomorphic Go web applications. We will be using the `isokit` package, from the Isomorphic Go toolkit, to implement an isomorphic web application:

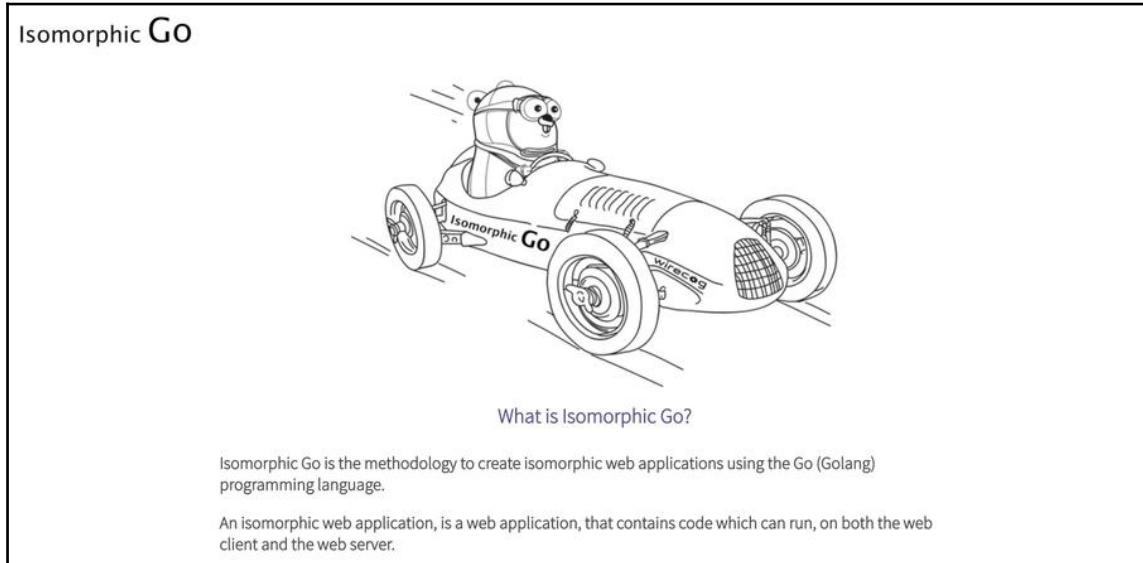


Figure 2.6: The Isomorphic Go website

## Installing isokit

The `isokit` package from the Isomorphic Go toolkit provides a common isomorphic functionality, which can be used either on the server side or on the client side. Some notable benefits that the package provides include isomorphic template rendering, client-side application routing, automatic static assets bundling, and the ability to create isomorphic web forms.

We can install the `isokit` package by issuing the following `go get` command:

```
$ go get -u github.com/isomorphicgo/isokit
```

## The UX toolkit

The UX toolkit (<http://uxtoolkit.io>) allows us to implement *cogs*, which are reusable components implemented in Go that can be used across the web pages that comprise IGWEB. We will cover reusable components in Chapter 9, *Cogs – Reusable Components*.

### Installing the cog package

We can install the `cog` package by issuing the following `go get` command:

```
$ go get -u github.com/uxtoolkit/cog
```

Now that we've installed the Isomorphic Go toolchain, it's time to set up the IGWEB demo, the isomorphic web application that we'll be building in this book.

## Setting up the IGWEB demo

You can get the source code examples for this book by issuing the following `go get` command:

```
$ go get -u github.com/EngineerKamesh/igb
```

The source code for the completed implementation of the IGWEB demo website resides in the `igb/igweb` folder. Source code listings for individual chapters can be found inside the `igb/individual` folder.

## Setting up the application root environment variable

The IGWEB demo relies on an application root environment variable, `$IGWEB_APP_ROOT`, being defined. This environment variable is used by the web application to declare where it resides. By doing so, the web application can determine where other resources, such as static assets (images, css, and javascript), are present.

You should set up the `$IGWEB_APP_ROOT` environment variable by adding the following entry in your bash profile:

```
export IGWEB_APP_ROOT=${GOPATH}/src/github.com/EngineerKamesh/igb/igweb
```

To verify that the `$IGWEB_APP_ROOT` environment variable exists in the environment, you can use the `echo` command:

```
$ echo $IGWEB_APP_ROOT  
/Users/kamesh/go/src/github.com/EngineerKamesh/igb/igweb
```

## Transpiling the client-side application

Now that we've set up the `$IGWEB_APP_ROOT` environment variable, we can access the `client` directory, where the client-side web application is located:

```
$ cd $IGWEB_APP_ROOT/client
```

We issue the following `go get` command to install any additional dependencies that may be required for the proper functioning of our client-side application:

```
$ go get ./..
```

Finally, we issue the `gopherjs build` command to transpile the IGWEB client-side web application:

```
$ gopherjs build
```

After running the command, two files should be generated—`client.js` and `client.js.map`. The `client.js` source file is the JavaScript representation of IGWEB's client-side Go program. The `client.js.map` file is the source map file that will be used in conjunction with `client.js` by the web browser to provide us detailed information in the web console, which comes in handy when debugging issues.

Now that we've transpiled the code for IGWEB's client-side application, the next logical step would be to build and run IGWEB's server-side application. Before we can do that, we must install and run a local Redis instance, which is what we'll do in the next section.

## Setting up Redis

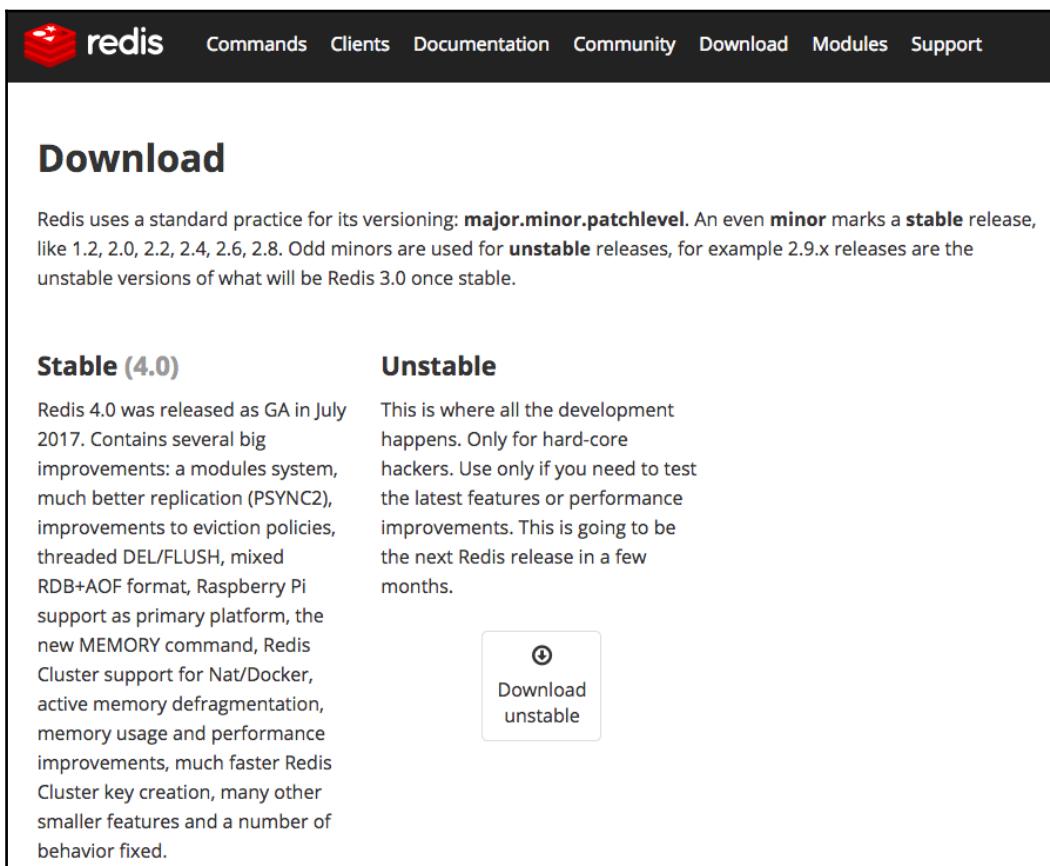
Redis is a popular NoSQL in-memory database. Since the entire database is present in the memory, database queries are blazingly fast. Redis is also known to offer support for a robust variety of data types, and it's a multipurpose tool that can be used as a database, a memory-cache, or even as a message broker.

In this book, we will use Redis for IGWEB's data persistence needs. We will be running our Redis instance on the default port of 6379.

We issue the following commands to download and install Redis:

```
$ wget http://download.redis.io/releases/redis-4.0.2.tar.gz
$ tar xzf redis-4.0.2.tar.gz
$ cd redis-4.0.2
$ make
$ sudo make install
```

An alternative to fetching Redis using the `wget` command is to obtain it from the Redis **Downloads** page, as shown in *Figure 2.7*, at <https://redis.io/download>:



The screenshot shows the Redis website's 'Download' section. The top navigation bar includes links for Commands, Clients, Documentation, Community, Download, Modules, and Support. The main content area has a heading 'Download' and a descriptive paragraph about Redis versioning. Below this, there are two sections: 'Stable (4.0)' and 'Unstable'. The 'Stable (4.0)' section contains a detailed list of improvements for Redis 4.0. The 'Unstable' section is described as a place for hard-core hackers to test latest features and includes a 'Download unstable' button with a circular arrow icon.

**Download**

Redis uses a standard practice for its versioning: **major.minor.patchlevel**. An even **minor** marks a **stable** release, like 1.2, 2.0, 2.2, 2.4, 2.6, 2.8. Odd minors are used for **unstable** releases, for example 2.9.x releases are the unstable versions of what will be Redis 3.0 once stable.

**Stable (4.0)**

Redis 4.0 was released as GA in July 2017. Contains several big improvements: a modules system, much better replication (PSYNC2), improvements to eviction policies, threaded DEL/FLUSH, mixed RDB+AOF format, Raspberry Pi support as primary platform, the new MEMORY command, Redis Cluster support for Nat/Docker, active memory defragmentation, memory usage and performance improvements, much faster Redis Cluster key creation, many other smaller features and a number of behavior fixed.

**Unstable**

This is where all the development happens. Only for hard-core hackers. Use only if you need to test the latest features or performance improvements. This is going to be the next Redis release in a few months.

Download unstable

Figure 2.7: The Downloads section on the Redis website

Once you have downloaded and installed Redis, you can start up the server by issuing the `redis-server` command:

```
$ redis-server
```

In another Terminal window, we can open up the Redis **command-line interface (CLI)**, to connect to the Redis server instance, using the `redis-cli` command:

```
$ redis-cli
```

We can set a `foo` key with the `bar` value using the `set` command:

```
127.0.0.1:6379> set foo bar
OK
```

We can get the value for the `foo` key using the `get` command:

```
127.0.0.1:6379> get foo
"bar"
```

You can learn more about Redis by visiting the documentation section of the Redis website at <https://redis.io/documentation>. Going through the Redis quick start document available at <https://redis.io/topics/quickstart> is also helpful. Now that we've installed our local Redis instance, it's time to build and run the IGWEB demo.

## Running the IGWEB demo

You can run the IGWEB web server instance by first changing the directory to the `$IGWEB_APP_ROOT` directory, and then issuing the `go run` command:

```
$ cd $IGWEB_APP_ROOT
$ go run igweb.go
```

You can access the IGWEB website by visiting the `http://localhost:8080/index` link from your web browser. You should be able to see the home page of the website, as shown in *Figure 2.8*:

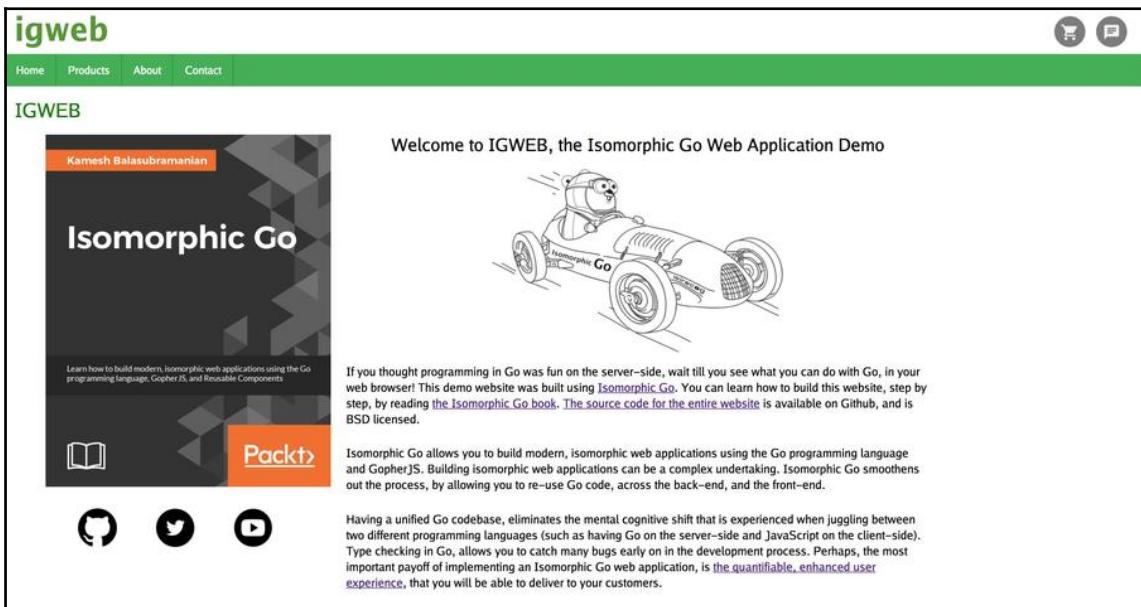


Figure 2.8: The IGWEB Home Page

The final step of our installation procedure is to load the local Redis instance with the sample dataset.

## Loading the sample dataset

The sample dataset provided is used to populate data for the **Products** listing and **About** pages. You can visit the **Products** listing page in your browser by accessing `http://localhost:8080/products`, and you should see the screen shown in *Figure 2.9*:

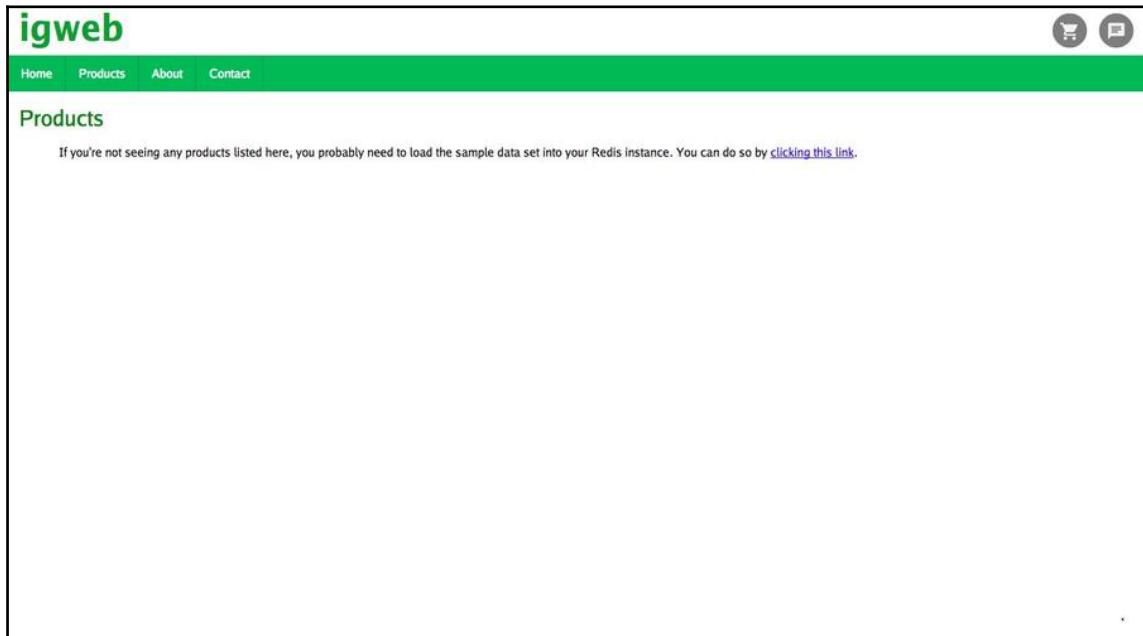


Figure 2.9: The Empty Products section with a message to load the sample dataset

Go ahead and click on the link displayed on the web page to load the sample dataset. When you click on the link, you should see the screen shown in *Figure 2.10*:

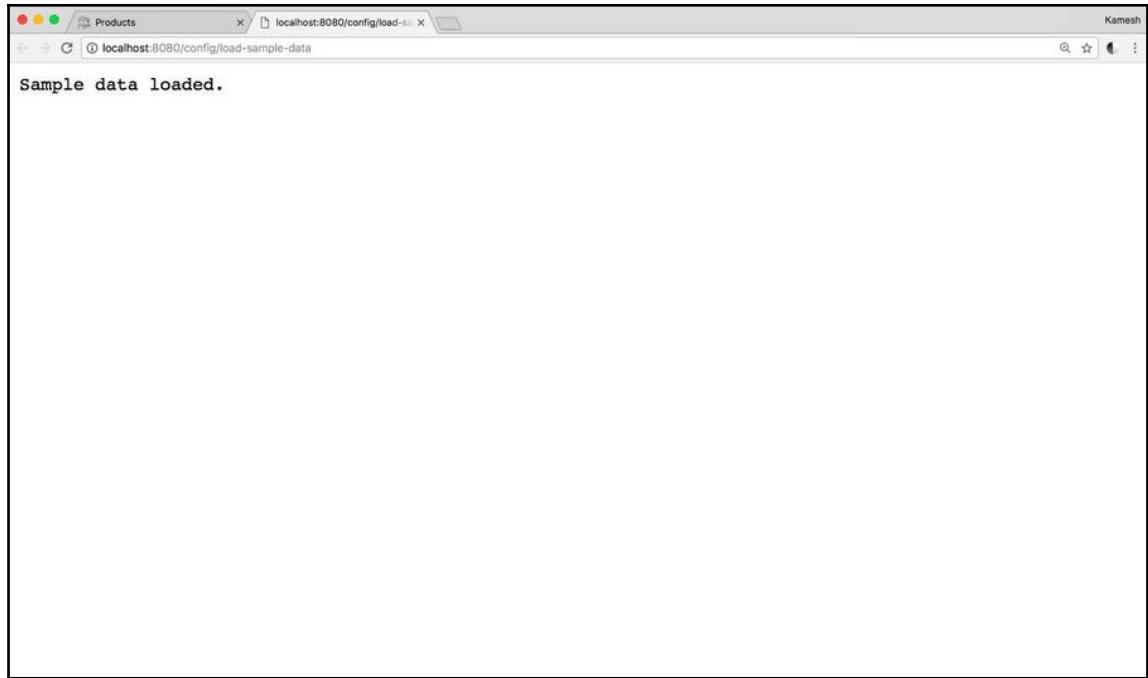


Figure 2.10: The confirmation that the sample dataset has been loaded

Now if you go back to the **Products** listing page, you should see the products displayed on the page, as shown in *Figure 2.11*:

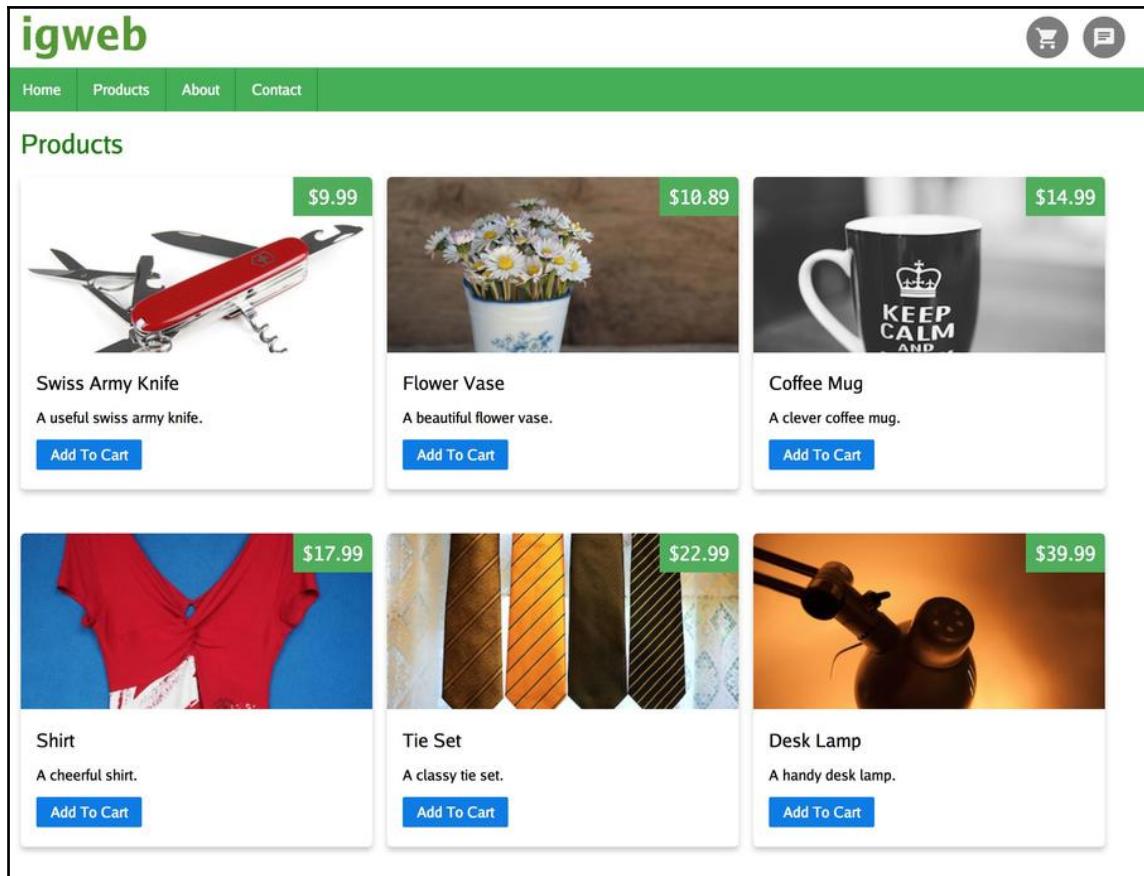


Figure 2.11: The Products Section, Populated With Products

We now have the IGWEB demo up and running!

Each time we want to make a change to our server-side Go application, we need to issue a `go build` command and restart the web server instance. Similarly, each time we make a change to our client-side Go application, we have to issue a `gopherjs build` command. Constantly issuing these commands, while we're deep in development, can be tedious and inefficient. The `kick` command provides us with a means to be more productive.

## Using kick

The `kick` command is a lightweight mechanism that provides an *instant kickstart* to a Go web server instance. The *instant kickstart* happens when a Go source file within the application's project directory (or any of its subdirectories) gets changed.

The `kick` command provides us a means to automate our development workflow, by recompiling our Go code and restarting the web server, anytime we make a change to a Go source file.

The workflow provided by `kick` is similar to developing web applications using a dynamic, scripting language such as PHP, where anytime a change is made to the PHP source file, the change is instantly reflected when the web page is refreshed in the browser.

What sets `kick` apart from other Go-based solutions in this problem space is that it takes both the `go` and `gopherjs` commands into consideration while performing the *instant kickstart*. It also takes changes made to template files into consideration as well, making it a handy tool for isomorphic web application development.

## Installing kick

To install `kick`, we simply issue the following `go get` command:

```
$ go get -u github.com/isomorphicgo/kick
```

## Running kick

To learn how to use `kick`, you can issue the `help` command line flag like this:

```
$ kick --help
```

The `--appPath` flag specifies the path to the Go application project. The `--gopherjsAppPath` flag specifies the path to your GopherJS project. The `--mainSourceFile` flag specifies the name of the Go source file that contains the `main` function implementation in the Go application project directory. If you're still running IGWEB using the `go run` command in the Terminal window, it's time to exit the program and run it using `kick` instead.

To run the IGWEB demo with `kick`, we issue the following command:

```
$ kick --appPath=$IGWEB_APP_ROOT --gopherjsAppPath=$IGWEB_APP_ROOT/client --mainSourceFile=igweb.go
```

## Verify that kick is working

Let's open up the **About** page (`http://localhost:8080/about`) along with the web inspector. Take note of the message that says **IGWEB Client Application** in the web console, as shown in *Figure 2.12*:

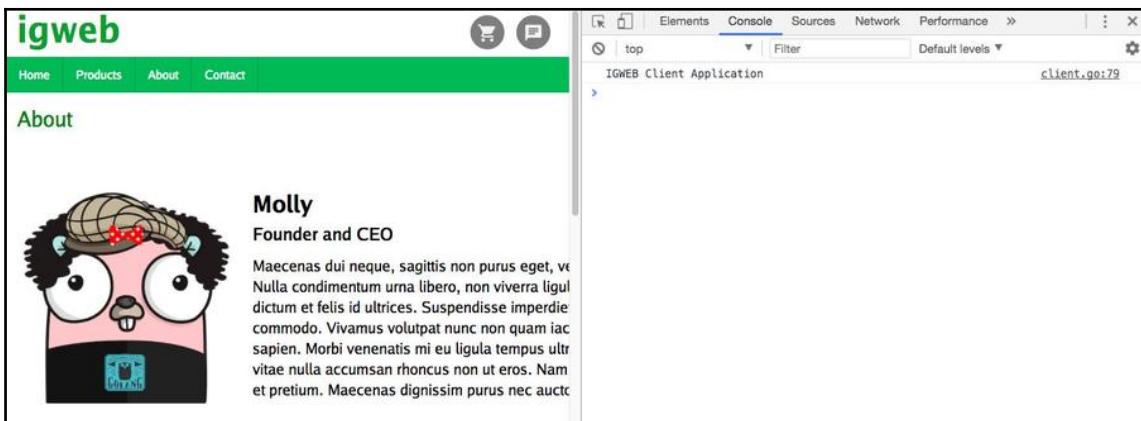


Figure 2.12: The message printed in the web console

Let's open up the `client.go` source file located in the `client` directory. Let's replace the first line in the `run` function with the following one:

```
println("IGWEB Client Application - Kamesh just made an update.")
```

Save the file and look at the Terminal window, where you're running `kick`, and you should be able to see the following message appear:

```
Instant KickStart Applied! (Recompiling and restarting project.)
```

This is a confirmation from `kick` that it has detected the change to the file, and that it has performed an *instant kickStart*. Now, let's reload the web page, and you should be able to see the updated message, as shown in *Figure 2.13*:



Figure 2.13: The modified message is printed in the web console

Now that you have the IGWEB demo running successfully on your machine using `kick`, an introduction to the project is in order.

## An introduction to the IGWEB demo

IGWEB is a fictitious tech startup created by three imaginary gophers who want to use Isomorphic Go to build a simple storefront demo on the web. The idea of these enterprising gophers is to take common, secondhand products sold at garage/yard sales and sell them online instead. This team of gophers have chosen to implement the IGWEB demo in Isomorphic Go to not only provide an enhanced user experience but also to gain greater search engine discoverability. If you haven't guessed already, IGWEB simply stands for *Isomorphic Go web application*.

## Building IGWEB from the ground up

In order to understand the underlying concepts that are involved with building an isomorphic web application, we will follow an idiomatic Go approach while creating IGWEB. We will make use of the functionality found from packages in the standard library as well as from third-party packages.

If you are experienced in developing web applications using a web framework, you may be wondering why we are taking this approach. At the time of writing, there is no Go-based web framework that provides functionality, out of the box, to create a web application that conforms to the isomorphic web application architecture that was presented in the previous chapter.

In addition to this, web frameworks often involve following a particular set of rules and conventions that may be framework-specific. Our focus is conceptual and is not tied to a particular web framework. Therefore, our attention will be focused on the underlying concepts involved in the creation of an isomorphic web application.

## The IGWEB roadmap

In the process of building each section and feature of the IGWEB demo website, we will learn more about Isomorphic Go. Here's a roadmap of the major sections/features of IGWEB along with the corresponding chapter in the book, where we implement that particular section or feature.

### The home page

Besides containing the image carousel of featured products and the multiple live clocks, the IGWEB home page also contains a section with links to standalone front-end coding examples.

The standalone examples include various front-end programming examples, an example of inline template rendering using GopherJS, and a local storage inspector. These examples will be covered in [Chapter 3, Go on the Front-End with GopherJS](#). The image carousel and the live clocks will be covered in [Chapter 9, Cogs – Reusable Components](#).

Location of the home page: <http://localhost:8080/index>.

## The about page

Our team of gophers want to be visible to the world, by being featured on IGWEB's **About** page. In the process of making this happen, we will learn isomorphic template rendering and the ability to share templates, template data, and template functions across environments.

The **About** page will be covered in [Chapter 4, Isomorphic Templates](#).

Location of the **About** page: <http://localhost:8080/about>.

## The products page

The **Products** listing page features the available products to be sold on the IGWEB website. Each product comes with a product title, an image thumbnail preview, the price of the product, and a short description. Clicking on the product image will take the user to the product detail page, where the user can learn more about that particular product. By implementing the product listing and product detail pages, we will learn about end-to-end application routing in Isomorphic Go.

The **Products** page will be covered in [Chapter 5, End-to-End Routing](#).

Location of the **Products** page: <http://localhost:8080/products>.

## The shopping cart feature

Each product card displayed in the **Products** page will contain an **Add To Cart** button. The button will also be available on the product's detail page. We will learn how to maintain the state of the shopping cart when performing add and remove operations on the shopping cart.

The shopping cart feature will be covered in [Chapter 6, Isomorphic Handoff](#).

Location: <http://localhost:8080/shopping-cart>.

## The contact page

The **Contact** page will provide the means to contact IGWEB's team of gophers. In the process of implementing the contact form, we will understand how to implement an isomorphic web form that shares validation logic, across environments. In addition to this, we will also learn how the web form can work resiliently, even in situations where JavaScript is disabled in the web browser.

The **Contact** page will be covered in [Chapter 7, The Isomorphic Web Form](#). The date picker cog for the contact form's time sensitivity input field will be covered in [Chapter 9, Cogs – Reusable Components](#).

Location of the **Contact** page: <http://localhost:8080/contact>.

## The live chat feature

In situations where greater user interactivity is required, website users can engage with a live chat bot. In the process of building the live chat feature, we will learn about the real-time web application functionality. The live chat feature will be covered in [Chapter 8, Real-Time Web Application Functionality](#).

The live chat feature can be activated by clicking on the live chat icon located at the top-right corner of the web page.

## Reusable components

We will return to the home page by implementing a variety of reusable components, such as a live clock, and an image carousel, which features products that are available on IGWEB. We'll also build a date picker cog for the **Contact** page, and a time ago cog for the **About** page. The time ago cog will represent time in a human readable format. We'll also take a look at implementing a notify cog, which is used to display notification messages to the user.

Reusable components will be covered in [Chapter 9, Cogs – Reusable Components](#).

# Project structure and code organization

The code for the IGWEB project can be found in the `igweb` folder, and it is organized into the following folders (listed in alphabetical order):

- bot
- chat
- client
  - carsdemo
  - chat
  - common
  - gopherjsprimer
  - handlers
  - localstoragedemo
  - tests
- common
  - datastore
- endpoints
- handlers
- scripts
- shared
  - cogs
  - forms
  - models
  - templates
  - templatedata
  - templatefuncs
  - validate
- static
  - css
  - fonts
  - images
  - js
  - templates
- submissions
- tests

The `bot` folder contains the source files that implement the chat bot for the live chat feature.

The `chat` folder contains the server-side code that implements the chat server for the live chat feature.

The `client` folder contains the client-side Go program that will be transpiled into JavaScript using GopherJS.

The `client/carsdemo` contains a standalone example that demonstrates inline template rendering using GopherJS. This example will be covered in [Chapter 3, Go on the Front-End with GopherJS](#).

The `client/chat` folder contains the client-side code that implements the chat client.

The `client/common` folder contains the client-side code that implements the common functionality used throughout the client-side application.

The `client/gopherjsprimer` contains the standalone GopherJS examples that will be covered in [Chapter 3, Go on the Front-End with GopherJS](#).

The `client/handlers` folder contains client-side, route/page handlers. These handlers are responsible for handling the routes of pages on the client side, preventing a full page reload. They are also responsible for handling all client-side user interactions that occur for a given web page.

The `client/localstoragedemo` contains an implementation of a local storage inspector, which will be covered in [Chapter 3, Go on the Front-End with GopherJS](#).

The `client/tests` folder contains end-to-end tests that exercise the client-side functionality. This folder consists of these three folders: `client/tests/go`, `client/tests/js`, and `client/tests/screenshots`. The `go` subfolder contains CasperJS tests, which are automated tests that simulate user interactions with the website implemented in Go. Running the `build_casper_tests.sh` bash script, found in the `scripts` folder, will transpile each Go source file into its equivalent JavaScript representation, which will be stored in the `js` subfolder. Upon running the CasperJS tests, screenshots are generated and saved in the `screenshots` subfolder.

The `common` folder contains the server-side code that implements the common functionality used throughout the server-side application.

The `common/datastore` folder contains the server-side code that implements a Redis datastore to satisfy the application's data persistence needs.

The `endpoints` folder contains the server-side code for the Rest API endpoints that are responsible for servicing XHR calls made from the web client.

The `handlers` folder contains the server-side code for the server-side route handler functions responsible for servicing a particular route. The primary responsibility of these handler functions is to send a web page response back to the client. They are used for the initial web page load, where the web page response is rendered on the server-side using the classic web application architecture.

The `scripts` folder contains handy bash shell scripts to be run on the command line.

The `shared` folder contains the isomorphic code that is shared across the server and the client. Taking a look into this folder gives us an understanding of all the Go code that can be shared across environments.

The `shared/cogs` folder contains reusable components (cogs), which are registered on the server side and deployed on the client side.

The `shared/forms` folder contains isomorphic web forms.

The `shared/models` folder contains the isomorphic types (structs) that we use to model data in our isomorphic web application.

The `shared/templates` folder contains isomorphic templates that can be rendered across environments.

The `shared/templatedata` folder contains isomorphic data objects that are to be supplied to isomorphic templates at the time of rendering.

The `shared/templatefuncs` folder contains isomorphic template functions that can be used across environments.

The `shared/validate` folder contains common, isomorphic validation logic, which can be utilized by web forms across environments.

The `static` folder contains all the static assets for the isomorphic web application.

The `static/css` folder contains the CSS stylesheet source files.

The `static/fonts` folder contains the custom fonts used by the web application.

The `static/images` folder contains the images used by the web application.

The `static/js` folder contains the JavaScript source code for the web application.

The `submissions` folder exists for illustration purposes. The folder houses the `submissions` package, which contains logic that is to be invoked after a web form has successfully cleared the web form validation process.

The `tests` folder contains end-to-end tests that exercise the server-side functionality.

## The MVC pattern

IGWEB's project codebase can be conceptualized as following the **Model-View-Controller (MVC)** pattern. The MVC pattern is heavily used in the creation of web applications, and it is depicted in *Figure 2.14*:

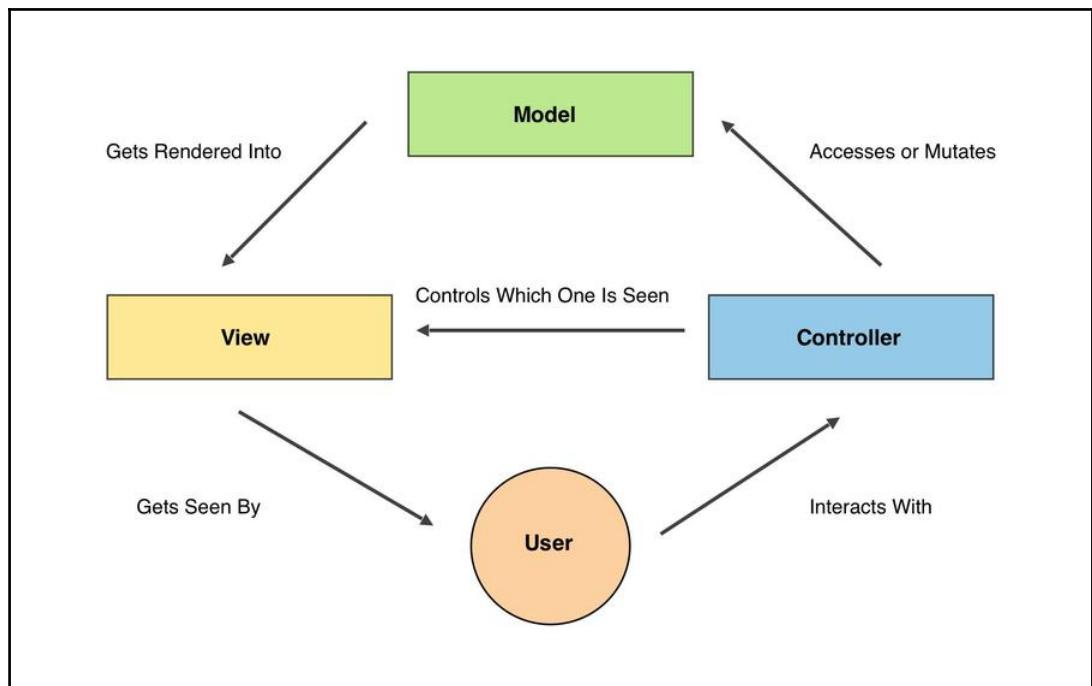


Figure 2.14: The Model View Controller pattern

There are three primary components in a MVC-based application—the model, the view, and the controller. The primary purpose of models are to supply data and business rules to the application. Think of the models as the gatekeepers of the application's data needs. The models for IGWEB can be found in the `shared/models` folder.

Views are responsible for the output that the user sees. The focus of views is on presentation, and the rendering of the models into the user interface, in a manner that makes sense to the user. The views in IGWEB exist as the templates found in the `shared/templates` folder.

Controllers implement the application logic of the system, and they basically tell the application how it should behave. You can conceptualize the controller as a broker between the model and the views of an application. The controller accepts user input from the view, and it can access or mutate the state of the model. The controller may also change what the view is currently presenting. The server-side controllers in IGWEB are the route handlers found in the `handlers` folder. The client-side controllers in IGWEB are the route/page handlers found in the `client/handlers` directory.



As you go through the examples in this book, take note that all folders mentioned in a relative manner are relative to the `igweb` folder.

Now that we have established how the code for the IGWEB project is organized, we can begin our journey to implement the individual sections and features that comprise our Isomorphic Go web application.

## The custom datastore

A custom datastore has been implemented for the IGWEB demo website. Although we will be using Redis as the exclusive database for this book, the truth is, you are free to use just about any database your heart desires—as long as you create a custom datastore that implements the `Datastore` interface.

Let's examine the section where we've defined the `Datastore` interface in the `datastore.go` source file found in the `common/datastore` folder:

```
type Datastore interface {
    CreateGopherTeam(team []*models.Gopher) error
    GetGopherTeam() []*models.Gopher
    CreateProduct(product *models.Product) error
    CreateProductRegistry(products []string) error
```

```

GetProducts() []*models.Product
GetProductDetail(productTitle string) *models.Product
GetProductsInShoppingCart(cart *models.ShoppingCart) []*models.Product
CreateContactRequest(contactRequest *models.ContactRequest) error
Close()
}

```

We will be going over the individual methods of the `Datastore` interface in their respective chapters that deal with the particular sections or features, where the method is used. Note that the final method required to implement the `Datastore` interface is the `Close` method (shown in bold). The `Close` method determines how the datastore closes its connection (or drains its connection pool).



Examining the `RedisDatastore` implementation in the `redis.go` source file, found in the `common/datastore` folder, will provide an idea of what goes into creating a custom datastore that implements the `Datastore` interface.

Moving further along in the `datastore.go` source file, we have defined the `NewDatastore` function, which is responsible for returning a new datastore:

```

const (
    REDIS = iota
)

func NewDatastore(datastoreType int, dbConnectionString string) (Datastore, error) {
    switch datastoreType {
        case REDIS:
            return NewRedisDatastore(dbConnectionString)
        default:
            return nil, errors.New("Unrecognized Datastore!")
    }
}

```

Our datastore solution is flexible since we can swap out the Redis datastore with any other database, so long as our new custom datastore implements the `Datastore` interface. Notice that we have defined the `REDIS` constant in the constant grouping using the `iota` enumerator (shown in bold). Examine the `NewDatastore` function, and note that a new `RedisDatastore` instance is returned when the `REDIS` case is encountered (shown in bold), inside the `switch` block on the `datastoreType`.

If we want to add support for another database, such as MongoDB, we will simply add a new constant entry, `MONGODB`, to the constant grouping. In addition to that, we will add an additional `case` statement for the `switch` block in the `NewDatastore` function, for MongoDB, which returns a `NewMongoDataStore` instance, providing the connection string to the MongoDB instance as an input argument to the function. The `NewMongoDBDatastore` function will return an instance of our custom datastore type, `MongoDBDataStore`, which will implement the `Datastore` interface.

A great benefit of implementing the custom datastore in this manner is that we can prevent littering our web application with database driver-specific calls for a particular database. With the custom datastore, our web application becomes agnostic to the database and provides us with greater flexibility in handling our data access and data storage needs.



The GopherFace web application, from the web programming with Go video series, implements a custom datastore for MySQL, MongoDB, and Redis. An example of the custom datastore using these databases is available at <https://github.com/EngineerKamesh/gofullstack/tree/master/volume2/section5/gopherfacedb/common/datastore>.

## Dependency injections

The server-side application's primary entry point is the `main` function defined in the `igweb.go` source file. The client-side application's primary entry point is the `main` function defined in the `client/client.go` source file. In both of these primary entry points, we utilize a dependency injection technique to share a common functionality throughout the web application. By doing so, we avoid having to utilize package-level global variables.

On both the server side, and on the client side, we implement a custom `Env` type in the `common` package. You may consider that `Env` stands for the common functionality that is to be accessed from the *application environment*.

Here's the declaration of the `Env` struct on the server side, found in the `common/common.go` source file:

```
package common

import (
    "github.com/EngineerKamesh/igb/igweb/common/datastore"
    "github.com/gorilla/sessions"
    "github.com/isomorphicgo/isokit"
)
```

```
type Env struct {
    DB datastore.Datastore
    TemplateSet *isokit.TemplateSet
}
```

The `DB` field will be used to store the custom datastore object.

The `TemplateSet` field is a pointer to a `TemplateSet` object. A template set allows us to render templates in a flexible manner across environments, and we'll be going over them in detail in [Chapter 4, Isomorphic Templates](#).

The `Store` field is a pointer to a `sessions.FilesystemStore` object. We will be using the `sessions` package from the Gorilla toolkit for session management.

Inside the `main` function in the `igweb.go` source file, we will declare a `env` variable, an object of the `common.Env` type:

```
env := common.Env{}
```

We assign the `DB` and `TemplateSet` fields of the `env` object with a newly created `RedisDatastore` instance and a newly created `TemplateSet` instance, respectively (the assignments are shown in bold). For illustration purposes, we have omitted some code and we have shown a partial code listing here:

```
db, err := datastore.NewDatastore(datastore.REDIS, "localhost:6379")
ts := isokit.NewTemplateSet()

env.TemplateSet = ts
env.DB = db
```

We will use the Gorilla Mux router for our server-side routing needs. Notice that we pass in a reference to the `env` object as an input argument (shown in bold) to the `registerRoutes` function:

```
func registerRoutes(env *common.Env, r *mux.Router) {
```

We propagate the `env` object to our request handler functions by including a reference to the `env` object as an input argument to the route handler function that we register for a particular route, as shown here:

```
r.Handle("/index", handlers.IndexHandler(env)).Methods("GET")
```

By calling the Gorilla Mux router's `Handle` method, we have registered the `/index` route, and we have associated the `IndexHandler` function from the `handlers` package as the function that will service this route. We have supplied the reference to the `env` object as the sole input argument to this function (shown in bold). At this point, we have successfully propagated the `RedisDatastore` and `TemplateSet` instances, and we have made them available to the `IndexHandler` function.

Let's examine the source code of the `IndexHandler` function defined in the `handlers/index.go` source file:

```
package handlers

import (
    "net/http"

    "github.com/EngineerKamesh/igb/igweb/common"
    "github.com/EngineerKamesh/igb/igweb/shared/templatedata"
    "github.com/isomorphicgo/isokit"
)

func IndexHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        templateData := templatedata.Index{PageTitle: "IGWEB"}
        env.TemplateSet.Render("index_page", &isokit.RenderParams{Writer: w,
        Data: templateData})
    })
}
```

Notice that the handler logic for the `handler` function is placed into a closure, and we have closed over the `env` variable. This allows us to satisfy the requirement that the handler function should return a `http.Handler`, and at the same time, we can provide access to the `env` object to the handler function.

The benefit of this approach, instead of using package-level global variables, is that we can explicitly see that this handler function requires the `env` object to properly function by examining the input arguments to the function (shown in bold).

We follow a similar dependency injection strategy on the client side as well. Here's the declaration of the `Env` type on the client side, found in the `client/common/common.go` source file:

```
package common

import (
    "github.com/isomorphicgo/isokit"
```

```
"honnef.co/go/js/dom"
)

type Env struct {
    TemplateSet *isokit.TemplateSet
    Router *isokit.Router
    Window dom.Window
    Document dom.Document
    PrimaryContent dom.Element
    Location *dom.Location
}
```

The `Env` type that we have declared on the client side is different from the one that we declared on the server side. This is understandable, since there's a different set of common functionality that we want to have access to on the client side. For example, there is no `RedisDatastore` that lives on the client side.

We have declared the `TemplateSet` field in the same manner that we did on the server side. Because the `*isokit.TemplateSet` type is isomorphic, it can exist on both the server side and the client side.

The `Router` field is a pointer to the client-side `isokit.Router` instance.

The `Window` field is the `Window` object, and the `Document` field is the `Document` object.

The `PrimaryContent` field represents the `div` container that we will render page content to, on the client side. We will be covering the roles of these fields in more detail in [Chapter 4, Isomorphic Templates](#).

The `Location` field is for the `Window` object's `Location` object.

Inside the `registerRoutes` function defined in the `client.go` source file, we use the `isokit.Router` to handle our client-side routing needs. We propagate the `env` object to the client-side handler function as follows:

```
r := isokit.NewRouter()
r.Handle("/index", handlers.IndexHandler(env))
```

Let's examine the source code of the `IndexHandler` function on the client side, defined in the `client/handlers/index.go` source file:

```
func IndexHandler(env *common.Env) isokit.Handler {
    return isokit.HandlerFunc(func(ctx context.Context) {
        templateData := templatedata.Index{PageTitle: "IGWEB"}
        env.TemplateSet.Render("index_content", &isokit.RenderParams{Data:
```

```
templateData, Disposition: isokit.PlacementReplaceInnerContents, Element:  
env.PrimaryContent, PageTitle: templateData.PageTitle})  
})  
}
```

The means by which we have provided access to the `env` object (shown in bold) to this handler function is identical to the way that we had done so on the server side. The handler logic for the handler function is placed into a closure, and we have closed over the `env` variable. This allows us to satisfy the requirement that the client-side handler function should return an `isokit.Handler`, and at the same time, we can provide the handler function access to the `env` object.



The dependency injection technique that we have utilized here, was inspired by the technique illustrated in Alex Edwards' blog post on organizing database access:

<http://www.alexedwards.net/blog/organising-database-access>.

## Summary

In this chapter, we walked you through the process of installing the Isomorphic Go toolchain. We introduced you to the IGWEB project, the isomorphic web application that we will implement in this book. We also examined the project structure and code organization of the IGWEB codebase.

We showed you how to set up the datastore and load the sample dataset into a Redis instance. We demonstrated how to use `kick` to perform an *instant kickstart* to speed up web application development cycles. We also provided a roadmap for the implementation of features and the functionality for the IGWEB project, and included the respective chapters where they will be covered. Finally, we demonstrated the dependency injection technique to share common functionality throughout the web application, both on the server side and on the client side.

Now that we have our tools in place, we need to establish a good understanding of using Go in the web browser. In *Chapter 3, Go on the Front-End with GopherJS*, we will explore GopherJS in further detail, and learn how to perform common DOM operations using GopherJS.

# 3

## Go on the Front-End with GopherJS

Ever since its creation, JavaScript has been the de facto programming language of the web browser. Accordingly, it has had a monopoly over front-end web development for a very long period of time. It's been the only game in town that comes with the capability to manipulate a web page's **Document Object Model (DOM)** and access various **application programming interfaces (APIs)** implemented in modern web browsers.

Due to this exclusivity, JavaScript has been the only viable option for isomorphic web application development. With the introduction of GopherJS, we now have the ability to create Go programs in the web browser, which also makes it possible to develop isomorphic web applications using Go.

GopherJS allows us to write programs, in Go, that get converted into an equivalent JavaScript representation, which is suitable to run in any JavaScript-enabled web browser. GopherJS provides us with a viable and attractive alternative to using JavaScript, especially if we are using Go on the server side. With Go on both ends of the spectrum (front-end and back-end), we have new opportunities to share code and eliminate the mental context shifts that come with having to juggle different programming languages across environments.

In this chapter, we will cover the following topics:

- The Document Object Model
- Basic DOM operations
- GopherJS overview
- GopherJS examples
- Inline template rendering
- Local storage

# The Document Object Model

Before we dive deeper into GopherJS, it's important for us to get an appreciation of what JavaScript, and by extension—GopherJS, does for us. One of the major capabilities that JavaScript has is its ability to access and manipulate the **DOM** (short for **Document Object Model**). The DOM is a tree data structure that represents the structure of a web page and all of the nodes (elements) that exist within it.

The significance of the DOM is that it acts as a programming interface for HTML documents, whereby programs that have access to the DOM can change a web page's style, structure, and content. Since each node in a DOM tree is an object, the DOM can be considered the object-oriented representation of a given web page. As such, the objects and their given properties can be accessed and changed using JavaScript.

*Figure 3.1* depicts the DOM hierarchy for a given web page. All elements on the web page are children of the **html** node, which is represented by the `<html>` tag in the web page's HTML source code:

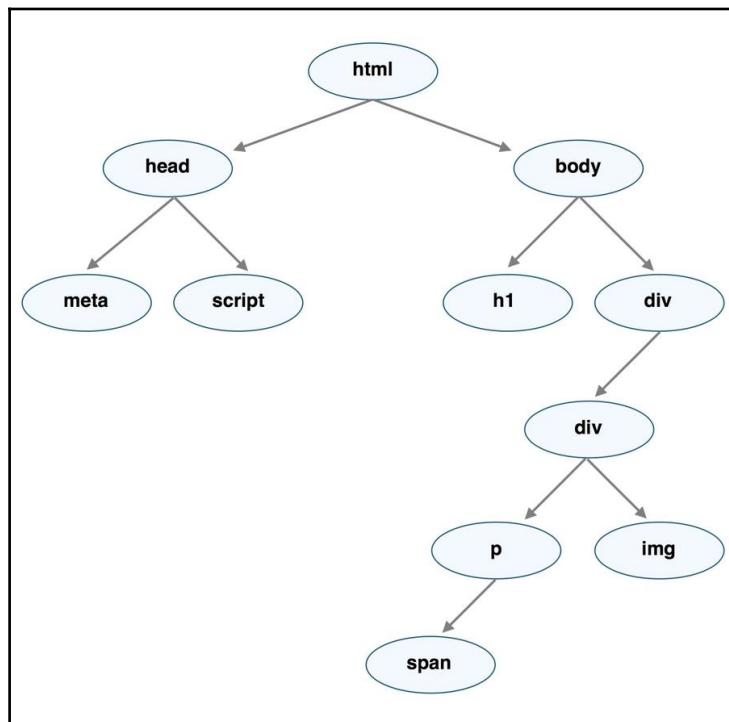


Figure 3.1: A web page's DOM hierarchy

The **head** node is a child of the **html** node and contains two children—meta (defined, in HTML, using the `<meta>` tag) and a script node (for an external CSS or JavaScript source file). At the same level of the head node exists the body node, which is defined using the `<body>` tag.

The body node contains all the elements that are to be rendered on the web page. Directly under the body node, we have a child, which is a heading node (defined using the `<h1>` tag), that is, the title of the web page. This node has no child elements.

At the same level of the heading node, we also have a div node (defined using a `<div>` tag). This node contains a div child node, which has two children—a paragraph node (defined using the `<p>` tag) and at the same level of this node exists an image node (defined using the `<img>` tag).

The image node has no child elements, and the paragraph node has one child element—a span node (defined using the `<span>` tag).

The JavaScript runtime included in the web browser provides us with the functionality to access the various nodes in the DOM tree along with their respective values. Using the JavaScript runtime, we can access both an individual node, and, if the given node contains children, we can access a collection of all the parent's child nodes.

Since web pages are represented as a collection of objects, using the DOM we can access the events, methods, and properties of any given DOM object. In fact, the `document` object represents the web page document itself.



Here's a helpful introduction to the DOM from the MDN website:  
[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model/Introduction](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction).

## Accessing and manipulating the DOM

As noted previously, we can use JavaScript to access and manipulate the DOM for a given web page. Since GopherJS transpiles to JavaScript, we now have the capability to access and manipulate the DOM within the confines of Go. *Figure 3.2* depicts a JavaScript Program accessing/manipulating the DOM along with a Go program also accessing/manipulating the DOM:

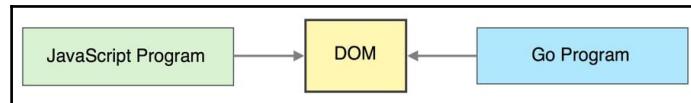


Figure 3.2: The DOM can be accessed and manipulated by a JavaScript program and/or a Go program (using GopherJS)

Now, let's take a look at a few simple programming snippets, where we can access the JavaScript functionality using Go, followed by some basic DOM operations using JavaScript and their equivalent instruction in GopherJS. For the time being, let's get a preview of what coding with GopherJS looks like. These concepts will be explained in further detail, as fully fleshed out examples, later in this chapter.

## Basic DOM operations

In this section, we'll look at a collection of some basic DOM operations. Each DOM operation presented includes the equivalent operation performed in JavaScript, GopherJS, and using the DOM binding.

### Displaying an alert message

#### JavaScript

```
alert("Hello Isomorphic Go!");
```

#### GopherJS

```
js.Global.Call("alert", "Hello Isomorphic Go!")
```

## DOM Binding

```
dom.GetWindow().Alert("Hello Isomorphic Go!")
```

One of the most basic operations we can perform is to show an `alert` message in a modal dialog. In JavaScript, we can display the `alert` message using the built-in `alert` function:

```
alert("Hello Isomorphic Go!");
```

This line of code will print out the message `Hello Isomorphic Go!` in a modal window dialog. The `alert` function blocks further execution until the user dismisses the `alert` dialog.

When we make a call to the `alert` method, we are actually calling it in this manner:

```
window.alert("Hello Isomorphic Go!");
```

The `window` object is a global object, representing an open window in the web browser. The JavaScript implementation allows us to directly call the `alert` function along with the other built-in functions, without explicitly referencing them as methods of the `window` object as a means of convenience.

We use the `js` package to access the JavaScript functionality through Go, using GopherJS. We can import the package into our Go program as follows:

```
import "github.com/gopherjs/gopherjs/js"
```

The `js` package provides us with functionality to interact with native JavaScript APIs. Calls to functions in the `js` package are translated directly to their equivalent JavaScript syntax.

We can display an `alert` message dialog using Go, with GopherJS, in the following manner:

```
js.Global.Call("alert", "Hello Isomorphic Go!")
```

In the preceding code snippet, we used the `Call` method that is available to the `js.Global` object. The `js.Global` object provides us with JavaScript's global object (the `window` object).

Here's what the `Call` method signature looks like:

```
func (o *Object) Call(name string, args ...interface{}) *Object
```

The `Call` method will call the global object's method with the provided name. The first parameter provided to the method is the name of the method to call. The second parameter is a list of arguments that are to be passed on to the global object's method. The `Call` method is known as a variadic function, since it can take in a variable number of parameters of the `interface{}` type.



You can learn more about the `Call` method by viewing the GopherJS documentation at <https://godoc.org/github.com/gopherjs/gopherjs/js#Object.Call>.

Now that we've seen how to display the `alert` dialog window using the `Call` method of the `js.Global` object, let's take a look at the DOM bindings.

The `dom` package provides us with convenient GopherJS bindings to the JavaScript DOM API. The idea behind using this package, as opposed to performing all operations using the `js.Global` object, is that the DOM bindings provides us with an idiomatic way to call the common DOM API functionality.

If you are already familiar with the JavaScript APIs used to access and manipulate the DOM, then using the `dom` package will feel second nature to you. We can access the global `window` object using the `GetWindow` function, like this:

```
dom.GetWindow()
```

Using the `dom` package, we can display the `alert` dialog message with the following code:

```
dom.GetWindow().Alert("Hello Isomorphic Go!")
```

A cursory view of this code snippet shows that this feels closer to the JavaScript way of calling the `alert` dialog:

```
window.alert("Hello Isomorphic Go!")
```

Due to this similarity, it's a good idea to be well-versed in the JavaScript DOM APIs, since it will provide you with the ability to be familiar with equivalent function calls, using the `dom` package.



You can learn more about the `dom` package by viewing the documentation for the package at <https://godoc.org/honnef.co/go/js/dom>.

## Getting a DOM element by ID

We can use the `document` object's `getElementById` method to access an element for a given `id`. In these examples, we access the primary content `div` container, which has `id` of "primaryContent".

### JavaScript

```
element = document.getElementById("primaryContent");
```

### GopherJS

```
element := js.Global.Get("document").Call("getElementById",  
"primaryContent")
```

### DOM Binding

```
element := dom.GetWindow().Document().GetElementByID("primaryContent")
```

Although the `dom` package's method calls are very similar to the JavaScript method calls, subtle differences can occur.

For example, take note of the capitalization in the `getElementById` method call on the `document` object using JavaScript, and compare it with the capitalization of the `GetElementByID` method call when using the DOM binding.

In order to export the `GetElementByID` method in Go, we must capitalize the first letter, here, `G`. Also, notice the subtle difference in the capitalization of the substring `Id` when using the JavaScript way, compared with the capitalization of `ID` when using the DOM binding.

## Query selector

The `querySelector` method of the `document` object provides us with a means to access a DOM element using a CSS query selector, in a manner similar to the jQuery library. We can access the `h2` element containing the welcome message, on the IGWEB homepage, using the `querySelector` method of the `document` object.

### JavaScript

```
element = document.querySelector(".welcomeHeading");
```

## GopherJS

```
element := js.Global.Get("document").Call("querySelector",  
  ".welcomeHeading")
```

## DOMBinding

```
element := dom.GetWindow().Document().QuerySelector(".welcomeHeading")
```

# Changing the CSS style property of an element

In the previous code snippets that we covered, we only considered examples where we accessed the DOM element. Now, let's consider an example where we change an element's CSS style property. We will hide the content inside the primary content `div` container by changing the value of the `div` element's `display` property.

We can save ourselves some typing by aliasing calls to `js.Global` and the `dom` package like this:



For GopherJS:

```
JS := js.Global
```

For the `dom` package:

```
D := dom.GetWindow().Document()
```

In order to change the `display` property of the primary content `div` container, we will first need to access the `div` element, and then change its `display` property to the `none` value.

## JavaScript

```
element = document.getElementById("primaryContent");  
element.style.display = "none"
```

## GopherJS

```
js := js.Global  
element := js.Get("document").Call("getElementById", "primaryContent")  
element.Get("style").Set("display", "none")
```

## DOM Binding

```
d := dom.GetWindow().Document()  
element := d.GetElementByID("welcomeMessage")  
element.Style().SetProperty("display", "none", "")
```



You can get a feel for working with GopherJS, using the GopherJS Playground at <https://gopherjs.github.io/playground/>.

## GopherJS overview

Now that we've seen a preview of using GopherJS, let's consider a high-level overview on how GopherJS works. *Figure 3.3* depicts an Isomorphic Go application that consists of a Go front-end web application (using GopherJS) and a Go back-end web application:

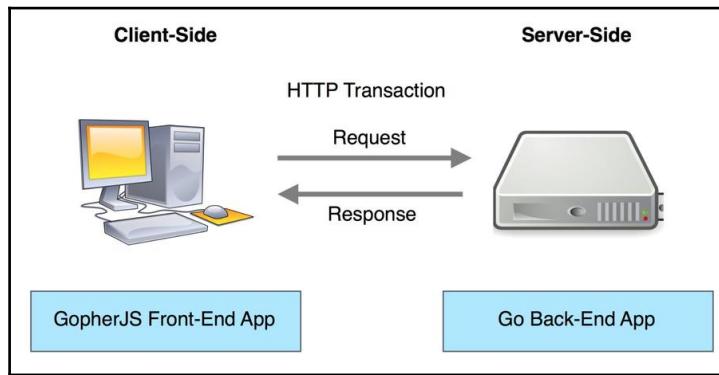


Figure 3.3: An Isomorphic Go web application consists of a Go front-end web application (using GopherJS) and a Go back-end web application

In *Figure 3.3*, we illustrated the means of communication as an HTTP transaction, but it's important to note that this is not the only means by which the client and web server can communicate. We can also establish a persistent connection using the web browser's WebSocket API, which we will cover in [Chapter 8, Real-Time Web Application Functionality](#).

In the microexamples that we covered in the previous section, we were introduced to the GopherJS DOM bindings, which provide us access to the DOM API—a JavaScript API implemented in the web browser. In addition to the DOM API, there are other APIs such as the XHR (to create and send XMLHttpRequests) API and WebSocket API (to create a bidirectional, persistent connection with the web server). There are GopherJS bindings available for the XHR and WebSocket APIs as well.

Figure 3.4 depicts common JavaScript APIs on the left, and their equivalent GopherJS binding on the right. With a GopherJS binding available, we can access JavaScript API functionality from within the Go programming language itself:

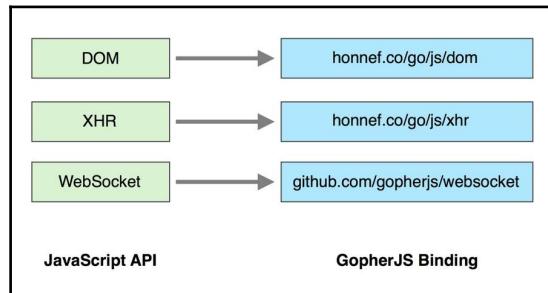


Figure 3.4: Common JavaScript APIs and their Equivalent GopherJS bindings

## The GopherJS transpiler

We use the GopherJS transpiler to convert a Go program into a JavaScript program. Figure 3.5 depicts a Go program that not only makes use of the functionality from the Go standard library but also uses the functionality from various JavaScript APIs using the equivalent GopherJS bindings package:

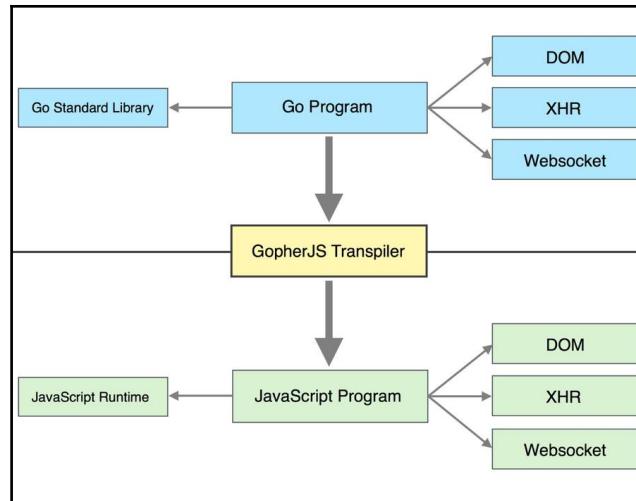


Figure 3.5: A Go program that makes use of the standard library and GopherJS bindings transpiled to an equivalent JavaScript program

We use the `gopherjs build` command to transpile the Go program into its equivalent JavaScript representation. The produced JavaScript source code is not meant to be modified by humans. The JavaScript program has access to the underlying JavaScript runtime embedded in the web browser, along with access to common JavaScript APIs.



To get an idea of how types are converted from Go to JavaScript, take a look at the table available at <https://godoc.org/github.com/gopherjs/gopherjs/js>.

With regard to IGWEB, we have organized the front-end Go web application project code inside of the `client` folder. This allows us to neatly separate the front-end web application from the back-end web application.

*Figure 3.6* depicts the client project folder containing numerous Go source files:

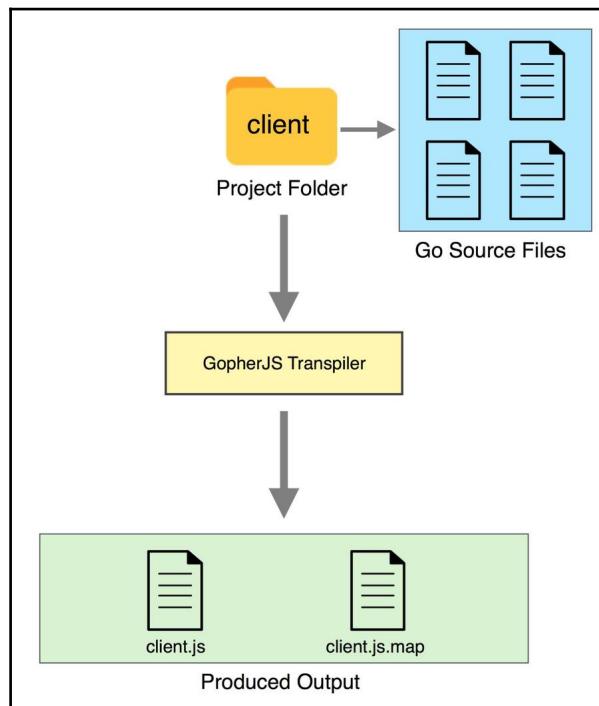


Figure 3.6: The client Folder Houses the Go source files that comprise the Front-End Go Web application. The GopherJS transpiler produces a JavaScript program (client.js) and a source map (client.js.map)

Upon running the GopherJS transpiler on the Go source files inside the `client` folder, by issuing the `gopherjs build` command, two output files are created. The first output file is the `client.js` file, which represents the equivalent JavaScript program. The second output file is the `client.js.map` file, which is a source map that's used for debugging purposes. This source map helps us when we are chasing down bugs using the web browser's console by providing us detailed information on produced errors.



*The Appendix: Debugging Isomorphic Go*, contains guidance and advice on debugging an isomorphic web application implemented in Go.

The `gopherjs build` command is synonymous in behavior with its `go build` counterpart. The client project folder can contain any number of subfolders, which may also contain Go source files. When we execute the `gopherjs build` command, a single JavaScript source program is created along with a source map file. This is analogous to the single static binary file that gets created when issuing a `go build` command.

Code that is shared between the server and the client, outside of the client folder, may be shared by specifying the proper path to the shared package in the `import` statement. The shared folder will contain code that is to be shared across environments, such as models and templates.

We can include the GopherJS produced JavaScript source file as an external `javascript` source file in our web page using the `<script>` tag, as shown here:

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

Keep in mind that when we issue a `gopherjs build` command, we are not just creating a JavaScript equivalent of the program we are writing, but we are also bringing along any packages from the standard library or third-party packages that our program relies on. So in addition to including our front-end Go program, GopherJS also includes any dependent packages that our program is dependent on.



Not all packages from the Go standard library work inside the web browser. You can reference the GopherJS compatibility table to view a list of supported packages from the Go standard library at <https://github.com/gopherjs/gopherjs/blob/master/doc/packages.md>.

The ramifications of this fact is that the produced JavaScript source code's file size will grow proportionately to the amount of dependencies we introduce in our Go program. The other ramifications of this fact is that it doesn't make sense to include multiple GopherJS produced JavaScript files on the same web page as depicted in *Figure 3.7*, since dependent packages (such as common packages from the standard library) will be included multiple times, unnecessarily bulking up our total script payload and offering no value in return:



Figure 3.7: Do not import multiple GopherJS produced source files in a single web page

A web page should therefore, at maximum, include only one GopherJS produced source file, as depicted in *Figure 3.8*:



Figure 3.8: Only a single GopherJS produced source file should be included in a web page

## GopherJS examples

Earlier in this chapter, we got a preview of what coding with GopherJS looks like. Now we will take a look at some fully fleshed out examples to solidify our understanding of some basic concepts.

As mentioned previously, the source code for the front-end web application can be found within the `client` folder.

If you want to manually transpile the Go code in the client directory, you can issue the `gopherjs build` command inside the `client` folder:

```
$ gopherjs build
```

As mentioned earlier, two source files will be produced—the `client.js` JavaScript source file, and the `client.js.map` source map file.

To run the web server manually, you can go into the `igweb` folder and run the following command:

```
$ go run igweb.go
```

A more convenient alternative is to compile the Go code and the GopherJS code using `kick`, with the following command:

```
$ kick --appPath=$IGWEB_APP_ROOT --gopherjsAppPath=$IGWEB_APP_ROOT/client --mainSourceFile=igweb.go
```

The advantage of using `kick` is that it will automatically watch for changes made either to the Go back-end web application or the GopherJS front-end web application. As noted in the previous chapter, `kick` will perform an *instant kickstart* when a change is detected, which will speed up your iterative development cycles.

Once you have the `igweb` program running, you may access the GopherJS examples at the following URL:

<http://localhost:8080/front-end-examples-demo>

The front-end examples demo will contain some basic GopherJS examples. Let's open up the `igweb.go` source file in the `igweb` folder to see how everything works.

Inside the `registerRoutes` function, we register the following routes:

```
r.Handle("/front-end-examples-demo",
  handlers.FrontEndExamplesHandler(env)).Methods("GET")
r.Handle("/lowercase-text",
  handlers.LowercaseTextTransformHandler(env)).Methods("POST")
```

The `/front-end-examples-demo` route is used to display our front-end examples web page. The `/lowercase-text` route is used to transform text into lowercase. We will be covering the second route in more detail later on; first, let's take a look at the handler function (found in the `handlers/frontendexamples.go` source file) that handles the `/front-end-examples-demo` route:

```
package handlers

import (
    "net/http"
    "github.com/EngineerKamesh/igb/igweb/common"
    "github.com/isomorphicgo/isokit"
)

func FrontEndExamplesHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        env.TemplateSet.Render("frontend_examples_page",
        &isokit.RenderParams{Writer: w, Data: nil})
    })
}
```

Here, we have defined our handler function, `FrontEndExamplesHandler`, which accepts a pointer to an `env` object as an input argument, and returns an `http.Handler` function. We have defined a closure to return the `http.HandlerFunc`, which accepts `http.ResponseWriter` and `*http.Request` as input arguments.

We call the `Render` method on the `TemplateSet` object to render the front-end examples page. The first input argument to the method is the template's name, which is `frontend_examples_page`. The second input argument is the render parameters that are to be used. Since we are rendering the template from the server side, we pass `w`, `http.ResponseWriter`, which is responsible for writing out the web page response (the rendered template). Since we are not passing any data to the template, we assign a value of `nil` to the `Data` field of the `RenderParams` struct.

In Chapter 4, *Isomorphic Templates*, we will explain how template sets work, and how we can use the isomorphic template renderer, provided by the `isokit` package, to render templates on both the server-side and the client-side.

Inside the partial source code listing of the `initializePage` function, found in the `client.go` source file, we have included the following line of code to initialize the GopherJS code examples (shown in bold):

```
func initializePage(env *common.Env) {  
  
    l := strings.Split(env.Window.Location().Pathname, "/")  
    routeName := l[1]  
  
    if routeName == "" {  
        routeName = "index"  
    }  
  
    if strings.Contains(routeName, "-demo") == false {  
        handlers.InitializePageLayoutControls(env)  
    }  
  
    switch routeName {  
  
        case "front-end-examples-demo":  
            gopherjsprimer.InitializePage()  
    }  
}
```

The `gopherjsprimer.InitializePage` function is responsible for adding event listeners to elements found on the front-end examples web page. Before we register any events, we first check to see if the page with the `/front-end-examples` route has been accessed. If the user is accessing a page with a different route, such as `/index`, there is no need to set up the event handlers for the front-end examples page. If the user has accessed the `/front-end-examples` route, then the flow of control will reach the `case` statement specifying the `"front-end-examples-demo"` value, and we will set up all the event handlers for the UI elements on the web page by calling the `gopherjsprimer.InitializePage` function.

Let's take a closer look at the `InitializePage` function found in the `client/gopherjsprimer/initpage.go` source file:

```
func InitializePage() {  
  
    d := dom.GetWindow().Document()  
  
    messageInput := d.GetElementByID("messageInput").(*dom.HTMLInputElement)  
  
    alertButtonJS :=  
    d.GetElementByID("alertMessageJSGlobal").(*dom.HTMLButtonElement)  
    alertButtonJS.AddListener("click", false, func(event dom.Event) {  
        DisplayAlertMessageJSGlobal(messageInput.Value)  
    })  
}
```

```
alertButtonDOM :=
d.GetElementByID("alertMessageDOM").(*dom.HTMLButtonElement)
alertButtonDOM.AddEventListener("click", false, func(event dom.Event) {
    DisplayAlertMessageDOM(messageInput.Value)
})

showGopherButton :=
d.GetElementByID("showGopher").(*dom.HTMLButtonElement)
showGopherButton.AddEventListener("click", false, func(event dom.Event) {
    ShowIsomorphicGopher()
})

hideGopherButton :=
d.GetElementByID("hideGopher").(*dom.HTMLButtonElement)
hideGopherButton.AddEventListener("click", false, func(event dom.Event) {
    HideIsomorphicGopher()
})

builtinDemoButton :=
d.GetElementByID("builtinDemoButton").(*dom.HTMLButtonElement)
builtinDemoButton.AddEventListener("click", false, func(event dom.Event) {
    builtinDemo(event.Target())
})

lowercaseTransformButton :=
d.GetElementByID("lowercaseTransformButton").(*dom.HTMLButtonElement)
lowercaseTransformButton.AddEventListener("click", false, func(event
dom.Event) {
    go lowercaseTextTransformer()
})

}
```

The `InitializePage` function is responsible for adding the event listeners to elements found in the front-end examples web page using the element's `AddEventListener` method (shown in bold).

## Displaying an alert message

Let's start off with an example to display an alert dialog. Earlier in this chapter, we saw how we could accomplish displaying the alert dialog using the `call` method of the `js.Global` object and the GopherJS DOM bindings. *Figure 3.9* depicts the user interface of our first example:

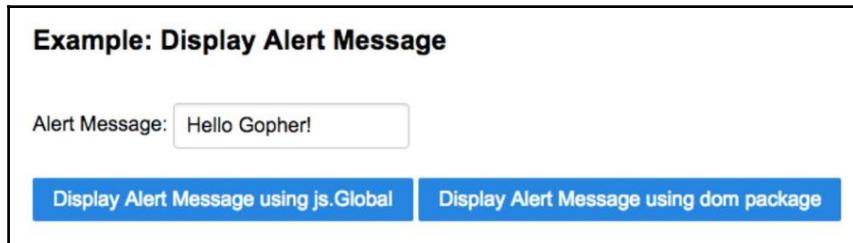


Figure 3.9: The Display Alert message example

The user interface consists of an input text field, where the user can enter a custom message to display in the alert dialog. Following the text field are two buttons:

- The first button will display the alert dialog using the `call` method on the `js.Global` object
- The second button will display the alert dialog using the GopherJS DOM Bindings



The HTML markup for the front-end examples can be found in the template file located at `shared/templates/frontend_examples_page tmpl`.

Here's the HTML markup for the alert message example:

```
<div class="example">
<form class="pure-form">
  <fieldset class="pure-group">
    <h2>Example: Display Alert Message</h2>
  </fieldset>
  <fieldset class="pure-control-group">
    <label for="messageInput">Alert Message: </label>
    <input id="messageInput" type="text" value="Hello Gopher!" />
  </fieldset>
  <fieldset class="pure-group">
    <button id="alertMessageJSGlobal" type="button" class="pure-button pure-
```

```
button-primary">Display Alert Message using js.Global</button>
  <button id="alertMessageDOM" type="button" class="pure-button pure-
button-primary">Display Alert Message using dom package</button>
</fieldset>
</form>
</div>
```

Here, we declared two buttons (shown in bold) and assigned unique ids to them. The button that will display the alert dialog using the `js.Global.Call` functionality has an `id` of `alertMessageJSGlobal`. The button that will display the alert dialog using the GopherJS DOM bindings has an `id` of `alertMessageDOM`.

The following code snippet from the `InitializePage` function, defined in the `initpage.go` source file, is responsible for setting up the event handlers for the `Display Alert Message` buttons that will be displayed in the example:

```
alertButtonJS :=
d.GetElementByID("alertMessageJSGlobal").(*dom.HTMLButtonElement)
  alertButtonJS.AddEventListener("click", false, func(event dom.Event) {
    DisplayAlertMessageJSGlobal(messageInput.Value)
  })

alertButtonDOM :=
d.GetElementByID("alertMessageDOM").(*dom.HTMLButtonElement)
  alertButtonDOM.AddEventListener("click", false, func(event dom.Event) {
    DisplayAlertMessageDOM(messageInput.Value)
  })
```

We fetch the first button by making a call to the `GetElementByID` function on the `document` object, passing in the `id` of the button as the `input` argument to the function. We then call the `AddEventListener` method on the button to create a new event listener, which will listen for a `click` event. We call the `DisplayAlertMessagesJSGlobal` function when the first button has been clicked, and pass in the value of the `messageInput` text field, which contains the custom **Alert Message** that the user can enter.

We set up the event listener for the second button in a similar fashion, except the function we call when a `click` event is detected on the button is `DisplayAlertMessageDOM`, which calls the function to show the alert dialog using the GopherJS DOM bindings. Again, we pass in the value of the `messageInput` text field to the function.

Now, if you were to click on either button, you should be able to see the alert dialog. Change the alert message to something different, and notice that the change you make to the **Alert Message** text field will be reflected in the alert dialog. *Figure 3.10* depicts the alert dialog with a custom message of **Hello Isomorphic Gopher!**:

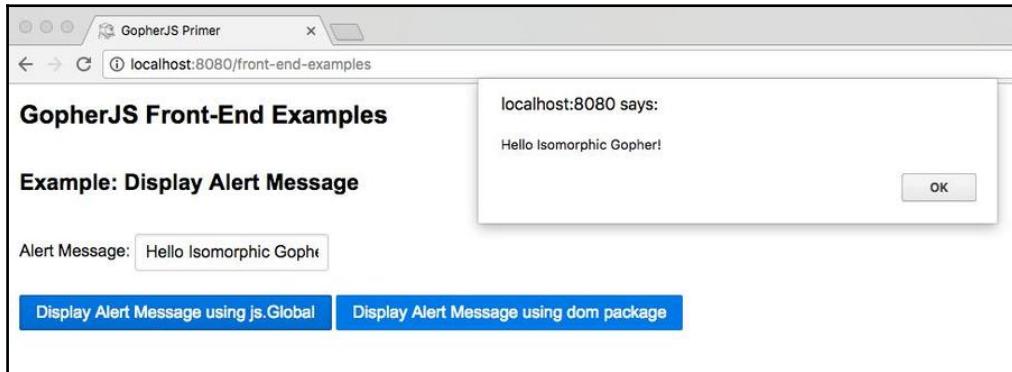


Figure 3.10: The example that displays the alert dialog with a custom alert message

## Changing an element's CSS style property

Now we will take a look at an example where we actually manipulate the DOM by changing an element's CSS style property. The user interface of this example consists of the image of the Isomorphic Gopher, and right below it are two buttons, as shown in *Figure 3.11*. The first button, when clicked, will show the Isomorphic Gopher image, if it is hidden. The second button, when clicked, will hide the Isomorphic Gopher image, if it is shown. *Figure 3.11* shows the Isomorphic Gopher when it is visible:

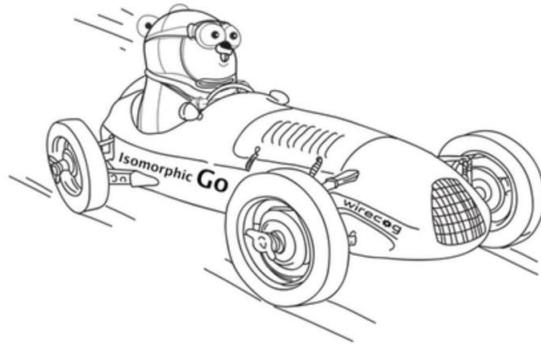
**Example: Change An Element's CSS Style Property****Show Isomorphic Gopher****Hide Isomorphic Gopher**

Figure 3.11: The user interface when the Isomorphic Gopher image is visible

Figure 3.12 depicts the user interface when the Isomorphic Gopher image is not visible:

**Example: Change An Element's CSS Style Property****Show Isomorphic Gopher****Hide Isomorphic Gopher**

Figure 3.12: The user interface when the Isomorphic Gopher image is not visible

Here's the HTML markup that generates the user interface for this example:

```
<div class="example">
  <form class="pure-form">
    <fieldset class="pure-group">
      <h2>Example: Change An Element's CSS Style Property</h2>
    </fieldset>
    <fieldset class="pure-group">
      <div id="igRacer">
        
      </div>
    </fieldset>
    <fieldset class="pure-group">
      <button id="showGopher" type="button" class="pure-button pure-button-primary">Show Isomorphic Gopher</button>
      <button id="hideGopher" type="button" class="pure-button pure-button-primary">Hide Isomorphic Gopher</button>
    </fieldset>
  </form>
</div>
```

Here, we declare an image tag that represents the Isomorphic Go image and assign it an `id` of `isomorphicGopher`. We declare two buttons (shown in bold):

- The first button, having an `id` of `showGopher`, will show the Isomorphic Gopher image, when clicked
- The second button, having an `id` of `hideGopher`, will hide the Isomorphic Gopher image, when clicked

The following code snippet from the `InitializePage` function is responsible for setting up the event handlers for the two buttons that show and hide the Isomorphic Gopher image:

```
showGopherButton :=
d.GetElementByID("showGopher").(*dom.HTMLButtonElement)
showGopherButton.AddEventListener("click", false, func(event dom.Event) {
  ShowIsomorphicGopher()
})

hideGopherButton :=
d.GetElementByID("hideGopher").(*dom.HTMLButtonElement)
hideGopherButton.AddEventListener("click", false, func(event dom.Event) {
  HideIsomorphicGopher()
})
```

If the **Show Isomorphic Gopher** button is clicked, we call the `ShowIsomorphicGopher` function. If the **Hide Isomorphic Gopher** button is clicked, we call the `HideIsomorphicGopher` function.

Let's examine the `ShowIsomorphicGopher` and `HideIsomorphicGopher` functions defined in the `client/gopherjsprimer/cssexample.go` source file:

```
package gopherjsprimer

import "honnef.co/go/js/dom"

func toggleIsomorphicGopher(isVisible bool) {

    d := dom.GetWindow().Document()
    isomorphicGopherImage :=
        d.GetElementByID("isomorphicGopher").(*dom.HTMLImageElement)

    if isVisible == true {
        isomorphicGopherImage.Style().SetProperty("display", "inline", "")
    } else {
        isomorphicGopherImage.Style().SetProperty("display", "none", "")
    }
}

func ShowIsomorphicGopher() {
    toggleIsomorphicGopher(true)
}

func HideIsomorphicGopher() {
    toggleIsomorphicGopher(false)
}
```

Both the `ShowIsomorphicGopher` and `HideIsomorphicGopher` functions call the `toggleIsomorphicGopher` function. The only difference is that the `ShowIsomorphicGopher` function calls the `toggleIsomorphicGopher` function with an input parameter of `true`, and the `HideIsomorphicGopher` function calls the `toggleIsomorphicGopher` function with an input parameter of `false`.

The `toggleIsomorphicGopher` function takes in a single argument, which is a Boolean variable indicating whether or not the `IsomorphicGopher` image should be shown, or not.

If we pass in a value of `true` to the function, then the Isomorphic Gopher image should be displayed, as shown in *Figure 3.11*. If we pass in a value of `false` to the function, then the Isomorphic Gopher image should not be displayed, as shown in *Figure 3.12*. We assign the value of the `Document` object to the `d` variable. We make a call to the `GetElementByID` method of the `Document` object to get the Isomorphic Gopher image. Notice that we have performed a type assertion (shown in bold) to assert that the value returned by `d.GetElementByID("isomorphicGopher")` has a concrete type of `*dom.HTMLImageElement`.

We declared an `if` conditional block that checks if the value of the `isVisible` Boolean variable is `true`, and if it is, we set the `display` property of the image element's `Style` object to be `inline`. This will cause the Isomorphic Gopher image to appear, as shown in *Figure 3.11*.

If the value of the `isVisible` Boolean variable is `false`, we reach the `else` block, and we set the `display` property of the image element's `Style` object to be `none`, which will prevent the Isomorphic Gopher image from being displayed, as shown in *Figure 3.12*.

## The JavaScript `typeof` operator functionality

The JavaScript `typeof` operator is used to return the type of a given operand. For example, let's consider the following JavaScript code:

```
typeof 108 === "number"
```

This expression will evaluate to the Boolean value of `true`. On a similar note, now consider this JavaScript code:

```
typeof "JavaScript" === "string"
```

This expression will also evaluate to the Boolean value of `true`.

So you might be wondering, how can we make use of the JavaScript `typeof` operator using Go? The answer is, we will need the `jsbuiltin` package, the GopherJS bindings for built-in JavaScript functionality, which includes the `typeof` operator.

In this example, we will make use of JavaScript's `typeof` operator using the `jsbuiltin` package. *Figure 3.13* depicts the user interface for this example:

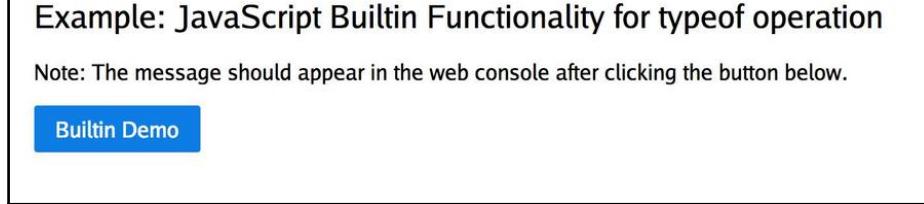


Figure 3.13: The user interface for the JavaScript typeof example

Here's the HTML markup that implements the user interface for this example:

```
<div class="example">
  <h2>Example: JavaScript Builtin Functionality for typeof operation</h2>
  <p>Note: The message should appear in the web console after clicking the
  button below.</p>
  <button id="builtinDemoButton" type="button" class="pure-button pure-
  button-primary">Builtin Demo</button>
</div>
```

We have declared a button with an `id` of `builtinDemoButton`. Now, let's set up an event listener for the **Builtin Demo** button, inside the `InitializePage` function, to handle the `click` event:

```
builtinDemoButton :=  
d.GetElementByID("builtinDemoButton").(*dom.HTMLButtonElement)  
builtinDemoButton.AddEventListener("click", false, func(event dom.Event)  
{  
    builtinDemo(event.Target())  
})
```

We get the `button` element by calling the `GetElementID` method on the `Document` object, `d`. We assign the returned `button` element to the `builtinDemoButton` variable. We then add an event listener to the `button` element to detect when it's clicked. If a click event is detected, we call the `builtinDemo` function and pass in the value of the `button` element, which happens to be the event target.

Let's examine the `builtindemo.go` source file found in the `client/gopherjsprimer` folder:

```
package gopherjsprimer  
  
import (  
  "github.com/gopherjs/jsbuiltin"  
  "honnef.co/go/js/dom"
```

```
)  
  
func builtinDemo(element dom.Element) {  
  
    if jsbuiltin.TypeOf(element) == "object" {  
        println("Using the typeof operator, we can see that the element that  
        was clicked, is an object.")  
    }  
  
}
```

The `bulitindemo` function accepts an input argument of the `dom.Element` type. Inside this function, we perform a JavaScript `typeof` operation on the element that's passed into the function by calling the `TypeOf` function from the `jsbuiltin` package (shown in bold). We check to see if the element passed in is an object. If it is an object, we print out a message to the web console, confirming that the element passed into the function is an object. *Figure 3.14* depicts the message printed on the web console:

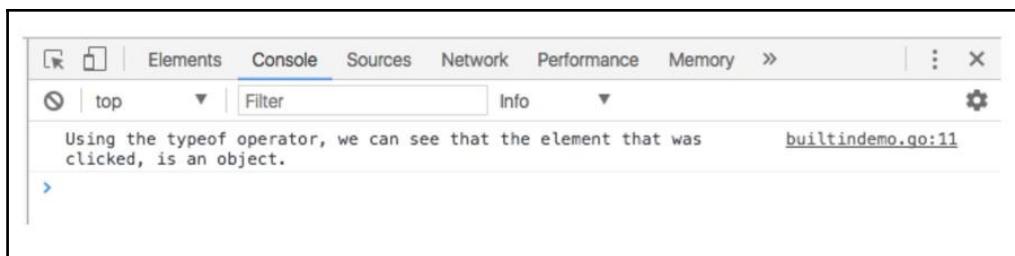


Figure 3.14: The message printed on the web console after the built in demo button is clicked

On the surface, this is a pretty trivial example. However, it highlights a very important concept—from within the confines of Go, we can still access the built-in JavaScript functionality.

## Transforming text to lowercase using an XHR post

Now we will create a simple lowercase text transformer. Any text the user enters will be converted to lowercase. The user interface for our lowercase text transformer solution is depicted in *Figure 3.15*. In the image, the input text is **GopherJS**. When the user clicks on the **Lowercase It!** button, the text in the text field will be transformed to its lowercase equivalent, which is **gopherjs**:

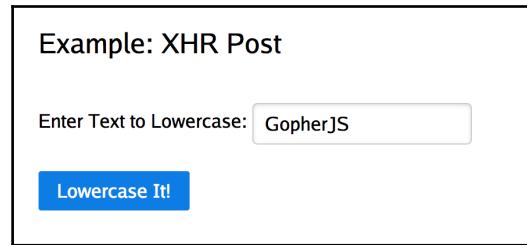


Figure 3.15: The lowercase Text Transformer example

In fact, we can apply the text transformation on the client side; however, it would be more interesting to see an example where we send the input text over to the web server in the form of an `XHR Post`, and then perform the lowercase transformation on the server side. Once the server is done transforming the text to lowercase, the input is sent back to the client, and the text field is updated with the lowercase version of the input text.

Here's what the HTML markup for the user interface looks like:

```
<div class="example">
  <form class="pure-form">
    <fieldset class="pure-group">
      <h2>Example: XHR Post</h2>
    </fieldset>
    <fieldset class="pure-control-group">
      <label for="textToLowercase">Enter Text to Lowercase: </label>
      <input id="textToLowercase" type="text" placeholder="Enter some text
here to lowercase." value="GopherJS" />
    </fieldset>
    <fieldset class="pure-group">
      <button id="lowercaseTransformButton" type="button" class="pure-button
pure-button-primary">Lowercase It!</button>
    </fieldset>
  </form>
</div>
```

We declare an `input` text field, where the user can enter text that they want to convert to lowercase. We assign an `id` of `textToLowercase` for the `input` text field. We then declare a button with an `id` of `lowercaseTransformButton`. When this button is clicked, we will initiate an `XHR Post` to the server. The server will convert the text to lowercase and send back the lowercase version of the entered text.

Here's the code from the `InitializePage` function, that is used to set up the event listener for the button:

```
lowercaseTransformButton :=  
d.GetElementByID("lowercaseTransformButton").(*dom.HTMLButtonElement)  
lowercaseTransformButton.AddEventListener("click", false, func(event  
dom.Event) {  
    go lowercaseTextTransformer()  
})
```

We assign the button element to the `lowercaseTransformButton` variable. We then call the `AddEventListener` method on the button element to detect a click event. When a click event is detected, we call the `lowercaseTextTransformer` function.

Here's the `lowercaseTextTransformer` function defined in the `client/gopherjsprimer/xhrpost.go` source file:

```
func lowercaseTextTransformer() {  
    d := dom.GetWindow().Document()  
    textToLowercase :=  
d.GetElementByID("textToLowercase").(*dom.HTMLInputElement)  
  
    textBytes, err := json.Marshal(textToLowercase.Value)  
    if err != nil {  
        println("Encountered error while attempting to marshal JSON: ", err)  
        println(err)  
    }  
  
    data, err := xhr.Send("POST", "/lowercase-text", textBytes)  
    if err != nil {  
        println("Encountered error while attempting to submit POST request via  
XHR: ", err)  
        println(err)  
    }  
  
    var s string  
    err = json.Unmarshal(data, &s)  
  
    if err != nil {  
        println("Encountered error while attempting to umarshal JSON data: ",  
err)  
    }  
    textToLowercase.Set("value", s)  
}
```

We first start out by fetching the text input element and assigning it to the `textToLowercase` variable. We then marshal the text value entered into the text input element to its JSON representation, using the `Marshal` function from the `json` package. We assign the marshaled value to the `textBytes` variable.

We use the GopherJS XHR bindings to send the `XHR Post` to the web server. The XHR bindings are made available to us through the `xhr` package. We call the `Send` function from the `xhr` package to submit the XHR Post. The first argument to the function is the HTTP method that we are going to use to submit the data. Here we have specified `POST` as the HTTP method. The second input argument is the path to `POST` the data to. Here we have specified the `/lowercase-text` route, which we had set up in the `igweb.go` source file. The third and last argument is the data that is to be sent through the XHR Post, which is `textBytes`—the JSON marshaled data.

The server response from the `XHR Post` will be stored in the `data` variable. We call the `Unmarshal` function in the `json` package to unmarshal the server's response and assign the unmarshaled value to the `s` variable of the `string` type. We then set the value of the text input element to the value of the `s` variable, using the `Set` method of the `textToLowercase` object.

Now, let's take a look at the server-side handler that's responsible for the lowercase transformation in the `handlers/lowercasetext.go` source file:

```
package handlers

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "strings"

    "github.com/EngineerKamesh/igb/igweb/common"
)

func LowercaseTextTransformHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        var s string

        reqBody, err := ioutil.ReadAll(r.Body)
        if err != nil {
            log.Print("Encountered error when attempting to read the request
body: ", err)
        }
    })
}
```

```
}

reqBodyString := string(reqBody)

err = json.Unmarshal([]byte(reqBodyString), &s)
if err != nil {
    log.Println("Encountered error when attempting to unmarshal JSON: ",
err)
}

textBytes, err := json.Marshal(strings.ToLower(s))
if err != nil {
    log.Println("Encountered error when attempting to marshal JSON: ", err)
}
fmt.Println("textBytes string: ", string(textBytes))
w.Write(textBytes)

})
}
```

In the `LowercaseTextTransformHandler` function, we make a call to the `ReadAll` function in the `ioutil` package to read the request body. We save the string value of `reqBody` to the `reqBodyString` variable. We then JSON unmarshal this string, and store the unmarshaled value to the `s` variable, which is of the `string` type.

We use the `ToLower` function from the `strings` package to transform the value of the `s` string variable to lowercase, and marshal the value into its JSON representation. We then call the `Write` method on `http.ResponseWriter`, `w`, to write out the JSON marshaled value of the string in lowercase.

When we click on the **Lowercase It!** button in the user interface, the string **GopherJS** gets transformed into its lowercase representation **gopherjs**, as shown in *Figure 3.16*:



Figure 3.16: The text "GopherJS" is converted to lowercase "gopherjs" once the button is clicked

## Inline template rendering

In this section, you will learn how to perform client-side template rendering in Go using GopherJS. We can render templates directly within the web browser using the `html/template` package. We will render the individual rows of a table of cars, using an inline template.

## The cars listing demo

In the cars listing demo, we will populate a table with rows that are rendered from an inline client-side Go template. In our example, the table will be a list of cars, and we will obtain the cars to be shown in the table from a slice of cars. We will then encode the slice of cars using `gob` encoding and transmit the data to the web server instance over an XHR call.

Client-side template rendering has many benefits:

- CPU usage on the web server is rendered, which is caused by server-side template rendering
- Full page reloads are not required to render the client-side template
- Bandwidth consumption is reduced by rendering the template on the client-side

Let's open up the `cars.html` source file in the `shared/templates/carsdemo_page tmpl` directory:

```
 {{ define "pagecontent" }}  
 <table class="mdl-data-table mdl-js-data-table mdl-shadow--2dp">  
   <thead>  
     <tr>  
       <th class="mdl-data-table__cell--non-numeric">Model Name</th>  
       <th class="mdl-data-table__cell--non-numeric">Color</th>  
       <th class="mdl-data-table__cell--non-numeric">Manufacturer</th>  
     </tr>  
   </thead>  
   <tbody id="autoTableBody">  
   </tbody>  
 </table>  
 {{end}}  
 {{template "layouts/carsdemolayout" . }}
```

This HTML source file contains the web page content for our example, a table of cars, where we will be rendering each row of the table using an inline template.

We have declared the table that will be displayed on the web page using the `table` tag. We have declared the headers for each column. Since we will be displaying a table of cars, we have three columns for each car; we have a column for the model name, a column for the color, and a column for the manufacturer.

Each new row that we will be adding to the table will be appended to the `tbody` element (shown in bold).

Notice that we use the `carsdemolayout tmpl` layout template, to layout the cars demo page. Let's open this file located in the `shared/templates/layouts` directory:

```
<html>
  {{ template "partials/carsdemohandler" }}
<body>
  <div class="pageContent" id="primaryContent">
    {{ template "pagecontent" . }}
  </div>
  <script src="/js/client.js"></script>
</body>
</html>
```

The layout template is responsible for not only rendering the `pagecontent` template but also the header template, `carsdemohandler tmpl`, which is located in the `templates/shared/partials` directory. The layout template is also responsible for importing the `client.js` external JavaScript source file that was produced by GopherJS.

Let's take a look at the `carsdemohandler tmpl` source file:

```
<head>
  <link rel="icon" type="image/png"
  href="/static/images/isomorphic_go_icon.png">
  <link rel="stylesheet"
  href="https://fonts.googleapis.com/icon?family=Material+Icons">
  <link rel="stylesheet"
  href="https://code.getmdl.io/1.3.0/material.indigo-pink.min.css">
  <script defer
  src="https://code.getmdl.io/1.3.0/material.min.js"></script>
</head>
```

In this header template file, we import the CSS stylesheet and the JavaScript source file for Material Design Library. We'll use Material Design Library to make our table look pretty with the default material design styles.

Inside the `initializePage` function of the `client.go` source file, we included the following line of code to initialize the cars demo code example, upon landing on the cars demo web page:

```
carsdemo.InitializePage()
```

Inside the `cars.go` source file in the `client/carsdemo` directory, we have declared the inline template used to render the information for a given car:

```
const CarItemTemplate = `<td class="mdl-data-table__cell--non-numeric">{{.ModelName}}</td>
<td class="mdl-data-table__cell--non-numeric">{{.Color}}</td>
<td class="mdl-data-table__cell--non-numeric">{{.Manufacturer}}</td>`
```

We declared the `CarItemTemplate` constant, which is a multi-line string that comprises our inline template. In the first line of our template, we render the column containing the model name. In the second line of our template, we render the color of the car. Finally, in the third line of our template, we render the manufacturer of the car.

We declared and initialized the `D` variable with the `Document` object, as shown here:

```
var D = dom.GetWindow().Document()
```

The `InitializePage` function (found in the `client/carsdemo/cars.go` source file) is responsible for calling the `cars` function:

```
func InitializePage() {
    cars()
}
```

Inside the `cars` function, we create a `nano`, an `ambassador`, and an `omni`—three instances of the `Car` type. Right after this, we use the `car` objects to populate the `cars` slice:

```
nano := models.Car{ModelName: "Nano", Color: "Yellow", Manufacturer: "Tata"}
ambassador := models.Car{ModelName: "Ambassador", Color: "White", Manufacturer: "HM"}
omni := models.Car{ModelName: "Omni", Color: "Red", Manufacturer: "Maruti Suzuki"}
cars := []models.Car{nano, ambassador, omni}
```

Now that we have a slice of `cars` to populate the table with, it's time to generate each row of the table with the following code:

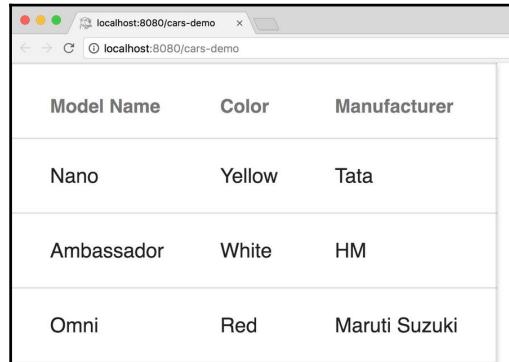
```
autoTableBody := D.GetElementByID("autoTableBody")
for i := 0; i < len(cars); i++ {
    trElement := D.CreateElement("tr")
    tpl := template.New("template")
    tpl.Parse(CarItemTemplate)
    var buff bytes.Buffer
    tpl.Execute(&buff, cars[i])
    trElement.SetInnerHTML(buff.String())
    autoTableBody.AppendChild(trElement)
}
```

Here, we have declared and initialized the `autoTableBody` variable, which is the `tbody` element of the table. This is the element we will use to append new rows to the table. We loop through the `cars` slice, and for each `Car` struct, we dynamically create a `tr` element, using the `CreateElement` method of the `Document` object. We then create a new template, and parse the contents of the car item template.

We declare a buffer variable named `buff`, to hold the result of the executed template. We call the `Execute` function on the template object, `tpl`, passing in `buff`, and the current `Car` record at the `i` index of the `cars` slice, which will be the data object that is fed to the inline template.

We then call the `SetInnerHTML` method on the `tr` element object and pass in the string value of the `buff` variable, which will contain our rendered template contents.

This is what the `cars` table looks like with all the rows populated:



A screenshot of a web browser window titled "localhost:8080/cars-demo". The browser shows a simple table with three rows of data. The table has three columns: "Model Name", "Color", and "Manufacturer". The data rows are: Row 1: Nano, Yellow, Tata; Row 2: Ambassador, White, HM; Row 3: Omni, Red, Maruti Suzuki.

Model Name	Color	Manufacturer
Nano	Yellow	Tata
Ambassador	White	HM
Omni	Red	Maruti Suzuki

Figure 3.17: The cars table

This example was useful for illustration purposes, however, it is not very practical in a real-world scenario. Mixing inline templates, written in HTML, inside Go source files can become an unmaintainable mess, as the project codebase scales. In addition to this, it would be nice if we had a means to access all the templates for user-facing web pages that the server had access to, on the client side. In fact, we can, and that will be our focus in [Chapter 4, Isomorphic Templates](#).

Now that we've seen how to render an inline template, let's consider how we can transmit the `cars` slice to the server as binary data, encoded in the `gob` format.

## Transmitting gob encoded data

The `encoding/gob` package provides us with the functionality to manage streams of gobs, which are binary values exchanged between an encoder and a decoder. You use the encoder to encode a value into `gob` encoded data and you use the decoder to decode `gob` encoded data.

With Go on the server side and on the client-side, we have created a Go-specific environment, as shown in *Figure 3.18*. This is an ideal environment to use the `encoding/gob` package, as a means for data exchange between the client and the server:

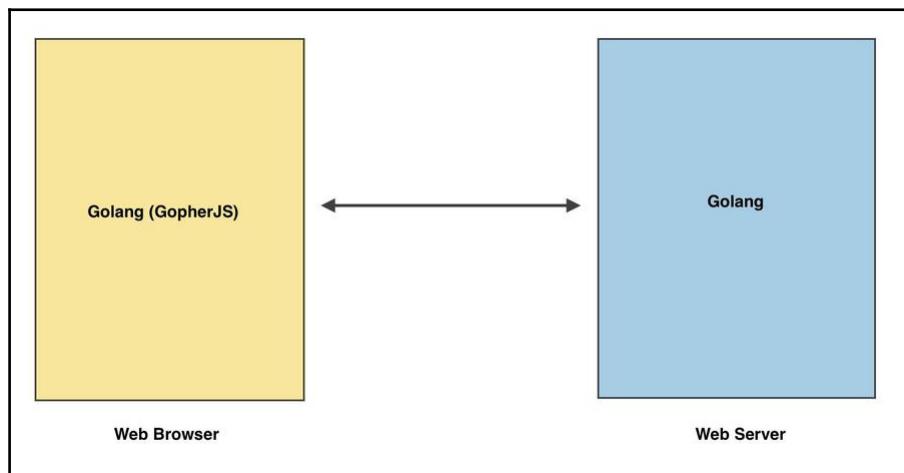


Figure 3.18: A Go-specific environment

The data that we will be transmitting consists of the `cars` slice. The `Car` struct can be considered isomorphic, since we can use the `Car` struct on both the client side and the server side.

Notice that in the `cars.go` source file, we have included the `encoding/gob` package (shown in bold) in our import groupings:

```
import (
    "bytes"
    "encoding/gob"
    "html/template"
    "github.com/EngineerKamesh/igb/igweb/shared/models"
    "honnef.co/go/js/dom"
    "honnef.co/go/js/xhr"
)
```

We encode the `cars` slice to the `gob` format using the following code:

```
var carsDataBuffer bytes.Buffer
enc := gob.NewEncoder(&carsDataBuffer)
enc.Encode(cars)
```

Here we have declared a `bytes` buffer called `carsDataBuffer` that will contain the `gob` encoded data. We created a new `gob` encoder, and specified that we want to store the encoded data into `carsDataBuffer`. We then called the `Encode` method on our `gob` encoder object, and passed in the `cars` slice. At this point, we have encoded the `cars` slice into `carsDataBuffer`.

Now that we have encoded the `cars` slice into the `gob` format, we can transmit the `gob` encoded data to the server over an XHR call using the `HTTP POST` method:

```
xhrResponse, err := xhr.Send("POST", "/cars-data",
carsDataBuffer.Bytes())

if err != nil {
    println(err)
}

println("xhrResponse: ", string(xhrResponse))
```

We call the `Send` function in the `xhr` package, and specify that we want to use the `POST` method, and will be sending the data to the `/cars-data` URL. We call the `Bytes` method on the `carsDataBuffer` to get the representation of the buffer as a byte slice. It is this byte slice that we will send off to the server, and it is the `gob` encoded `car` slice.

The response from the server will be stored in the `xhrResponse` variable, and we will print this variable out in the web console.

Now that we've seen the client-side of our program, it's time to take a look at the server-side handler function that services the `/cars-data` route. Let's examine the `CarsDataHandler` function defined in the `carsdata.go` source file found in the `handlers` directory:

```
func CarsDataHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        var cars []models.Car
        var carsDataBuffer bytes.Buffer

        dec := gob.NewDecoder(&carsDataBuffer)
        body, err := ioutil.ReadAll(r.Body)
        carsDataBuffer = *bytes.NewBuffer(body)
        err = dec.Decode(&cars)

        w.Header().Set("Content-Type", "text/plain")

        if err != nil {
            log.Println(err)
            w.Write([]byte("Something went wrong, look into it"))
        } else {
            fmt.Printf("Cars Data: %#v\n", cars)
            w.Write([]byte("Thanks, I got the slice of cars you sent me!"))
        }
    })
}
```

Inside the `CarsDataHandler` function, we declare the `cars` variable, which is a slice of `Car` objects. Right below this, we have `carsDataBuffer`, which will contain the `gob` encoded data that we receive from the XHR call that was sent from the client-side web application.

We create a new `gob` decoder and we specify that the `gob` data will be stored in `carsDataBuffer`. We then use the `ReadAll` function from the `ioutil` package to read the request body and to save all the contents to the `body` variable.

We then create a new bytes buffer and pass in the `body` variable as the input argument to the `NewBuffer` function. The `carsDataBuffer` now contains the `gob` encoded data that was transmitted over the XHR call. Finally, we make a call to the `Decode` function of the `dec` object to convert the `gob` encoded data back into a slice of the `Car` objects.

If we didn't receive any errors, we print out the `cars` slice to standard out:

```
Cars Data: []models.Car{models.Car{ModelName:"Nano", Color:"Yellow",  
Manufacturer:"Tata"}, models.Car{ModelName:"Ambassador", Color:"White",  
Manufacturer:"HM"}, models.Car{ModelName:"Omni", Color:"Red",  
Manufacturer:"Maruti Suzuki"}}
```

In addition to printing the `cars` slice to standard out, we write a response back to the web client indicating that the slice of `cars` has been received successfully. We can view this message in the web browser console:

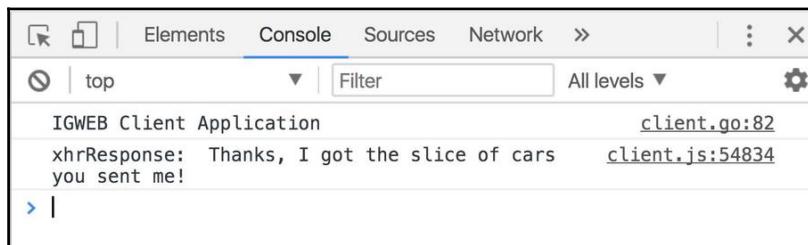


Figure 3.19: The server response to the web client

## Local storage

Did you know that the web browser comes with a built-in key-value database? The name of this database is local storage, and in JavaScript, we can access the `localStorage` object as a property of the `window` object. Local storage allows us to store data locally within the web browser. Local storage is per domain and protocol, meaning that pages from the same origin can access and modify shared data.

Here are some benefits of local storage:

- It provides secure data storage
- It has a far greater storage limit than cookies (at least 5 MB)
- It provides low latency data access

- It is helpful for web applications that need to operate offline (internet connection not required)
- It can be utilized as a local cache

## Common local storage operations

We will be showing you how to perform some common operations on the `localStorage` object using JavaScript code. These operations include the following:

1. Setting a key-value pair
2. Getting a value for a given key
3. Getting all key-value pairs
4. Clearing all entries

In the next section, we will show you how to perform the same operations using GopherJS, in a fully fleshed out example.

## Setting a key-value pair

To store an item into local storage, we call the `setItem` method of the `localStorage` object, and pass in the key and value as parameters to the method:

```
localStorage.setItem("foo", "bar");
```

Here we have provided a `"foo"` key, with a `"bar"` value.

## Getting a value for a given key

To get an item from local storage, we call the `getItem` method of the `localStorage` object and pass in the key as the single parameter to the method:

```
var x = localStorage.getItem("foo");
```

Here we have provided the `"foo"` key, and we expect that the value of the `x` variable will be equal to `"bar"`.

## Getting all key value pairs

We can retrieve all key-value pairs from local storage using a `for` loop and accessing the values of the key and value, using the `key` and `getItem` methods of the `localStorage` object:

```
for (var i = 0; i < localStorage.length; i++) {  
  console.log(localStorage.key(i)); // prints the key  
  console.log(localStorage.getItem(localStorage.key(i))); // prints the  
  value  
}
```

We use the `key` method on the `localStorage` object, passing in the numeric index, `i`, to get the `ith` key in the storage. Similarly, we pass in the `i` numeric index to the `key` method of the `localStorage` object, in order to get the name of the key at the `i`th place in the storage. Note that the name of the key is obtained by the `localStorage.key(i)` method call and passed to the `getItem` method to retrieve the value for the given key.

## Clearing all entries

We can easily remove all the entries in local storage by calling the `clear` method on the `localStorage` object:

```
localStorage.clear();
```

## Building a local storage inspector

With the information presented in the previous section on how to utilize the `localStorage` object, let's go ahead and build a local storage inspector. The local storage inspector, will allow us to perform the following operations:

- Viewing all the key-value pairs that are currently stored in local storage
- Adding a new key-value pair to local storage
- Clearing all key-value pairs in local storage

Figure 3.20 depicts the user interface for the local storage inspector:

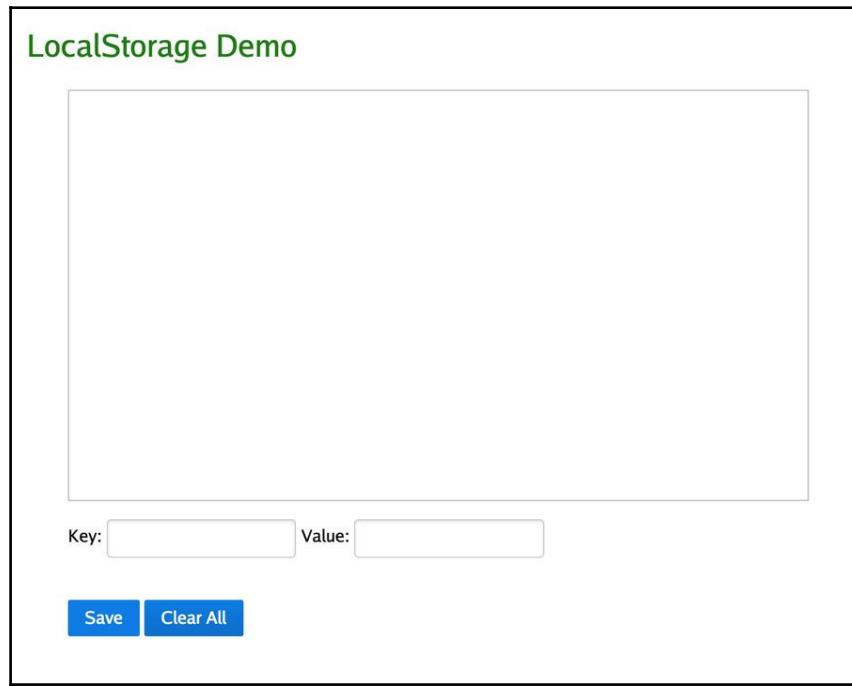


Figure 3.20: The Local Storage Demo user interface

The box directly under the **LocalStorage Demo** title is a `div` container that is responsible for holding the list of key-value pairs that are currently stored in local storage. The **Key** input text field is where the user enters the key for the key-value pair. The **Value** input text field is where the user enters the value for the key-value pair. Clicking on the **Save** button will save the new key-value entry into local storage. Clicking on the **Clear All** button, will clear all key-value entries in local storage.

## Creating the user interface

We've defined the layout for the local storage demo page inside the `localStorage_layout tmpl` source file found in the `shared/templates/layouts` folder:

```
<!doctype html>
<html>
  {{ template "partials/localstorageheader_partial" }}
```

```
<body>
  <div class="pageContent" id="primaryContent">
    {{ template "pagecontent" . }}
  </div>

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

</body>
</html>
```

This layout template defines the layout of the local storage demo web page. We use template actions (shown in bold) to render the `partials/localstorageheader_partial` header template, and the `pagecontent` page content template.

Notice that at the bottom of the web page, we include the JavaScript source file, `client.js`, which was produced by GopherJS, using the `script` tag (shown in bold).

We've defined the header template for the local storage demo page inside the `localStorageheader_partial.tpl` source file found in the `shared/templates/partials` folder:

```
<head>
  <title>LocalStorage Demo</title>
  <link rel="icon" type="image/png"
    href="/static/images/isomorphic_go_icon.png">
  <link rel="stylesheet"
    href="https://unpkg.com/purecss@1.0.0/build/pure-min.css"
    integrity="sha384-nn4HPE81THyVtfCBi5yW9d20FjT8BJwUXyWZT9InLYax14RDjBj46LmSztkmNP9w"
    crossorigin="anonymous">
    <link rel="stylesheet" type="text/css" href="/static/css/igweb.css">
    <link rel="stylesheet" type="text/css"
      href="/static/css/localstoragedemo.css">
</head>
```

This header template is meant to render the `head` tag, where we include external CSS stylesheets using the `link` tags (shown in bold).

We've defined the HTML markup for the local storage demo's user interface in the `localStorage_example_page.tpl` source file found in the `shared/templates` folder:

```
{{ define "pagecontent" }}

<h1>LocalStorage Demo</h1>
```

```
<div id="inputFormContainer">
  <form class="pure-form">
    <fieldset class="pure-group" style="min-height: 272px">
      <div id="storageContents">
        <dl id="itemList">
        </dl>
      </div>
    </fieldset>

    <fieldset class="pure-control-group">
      <label for="messageInput">Key: </label>
      <input id="itemKey" type="text" value="" />
      <label for="messageInput">Value: </label>
      <input id="itemValue" type="text" value="" />
    </fieldset>

    <fieldset class="pure-control-group">
    </fieldset>

    <fieldset class="pure-group">
      <button id="saveButton" type="button" class="pure-button pure-
button-primary">Save</button>
      <button id="clearAllButton" type="button" class="pure-button pure-
button-primary">Clear All</button>
    </fieldset>
  </form>
</div>

{{end}}
{{template "layouts/localstorage_layout" . }}
```

The div element with the id of "storageContents" will be used to store the list of item entries in the local storage database. In fact, we will use the dl (description list) element with id of "itemList" to display all the key-value pairs.

We have defined an input text field for the user to enter the key, and we have also defined an input text field for the user to enter the value. We've also defined the markup for the Save button, and directly under that, we've defined the markup for the Clear All button.

## Setting up the server-side route

We've registered the `/localStorage-demo` route inside the `registerRoutes` function found in the `igweb.go` source file:

```
r.Handle("/localStorage-demo",
  handlers.LocalStorageDemoHandler(env)).Methods("GET")
```

We've defined the `LocalStorageDemoHandler` server-side handler function to service the `/localStorage-demo` server-side route in the `localstoragedemo.go` source file found in the `handlers` folder:

```
package handlers

import (
  "net/http"

  "github.com/EngineerKamesh/igb/igweb/common"
  "github.com/isomorphicgo/isokit"
)

func LocalStorageDemoHandler(env *common.Env) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    env.TemplateSet.Render("localStorage_example_page",
    &isokit.RenderParams{Writer: w, Data: nil})
  })
}
```

The `LocalStorageDemoHandler` function is responsible for writing the web page response to the client. It calls the `Render` method of the application's `TemplateSet` object, to render the `localStorage_example_page` template. You will learn more about rendering isomorphic templates, in Chapter 4, *Isomorphic Templates*.

## Implementing the client-side functionality

Implementing the client-side functionality of the local storage inspector consists of these steps:

1. Initializing the local storage inspector web page
2. Implementing the local storage inspector

## Initializing the local storage inspector web page

In order to initialize the event handlers on the local storage inspector web page, we need to add the following line of code in the `localStorageDemo` case, inside the `initializePage` function found in the `client.go` source file:

```
localStorageDemo.InitializePage()
```

Calling the `InitializePage` function, defined in the `localStorageDemo` package, will add the event listeners for the **Save** and **Clear All** buttons.

## Implementing the local storage inspector

The implementation of the local storage inspector can be found in the `localStorage.go` source file in the `client/localStorageDemo` directory.

In the `import` groupings we include the `js` and `dom` packages (shown in bold):

```
package localStorageDemo

import (
    "github.com/gopherjs/gopherjs/js"
    "honnef.co/go/js/dom"
)
```

We've defined the `localStorage` variable and we've assigned it the value of the `localStorage` object that is attached to the `window` object:

```
var localStorage = js.Global.Get("localStorage")
```

As usual, we've aliased the `Document` object with the `D` variable to save us some typing:

```
var D = dom.GetWindow().Document().(dom.HTMLDocument)
```

The `InitializePage` function is responsible for setting up the event listeners for the **Save** and **Clear All** buttons:

```
func InitializePage() {
    saveButton := D.GetElementByID("saveButton").(*dom.HTMLButtonElement)
    saveButton.AddEventListener("click", false, func(event dom.Event) {
        Save()
    })

    clearAllButton :=
        D.GetElementByID("clearAllButton").(*dom.HTMLButtonElement)
```

```
clearAllButton.addEventListener("click", false, func(event dom.Event) {
  ClearAll()
}

DisplayStorageContents()
}
```

We fetch the `saveButton` element by calling the `GetElementByID` method of the `Document` object and providing the `id`, `"saveButton"`, as the sole input parameter to the method. Right below this, we add an event listener on the `click` event to call the `Save` function. Calling the `Save` function will save a new key-value pair entry.

We also fetch the `clearAllButton` element by calling the `GetElementByID` method of the `Document` object and providing the `id`, `"clearAllButton"`, as the sole input parameter to the method. Right below this, we add an event listener on the `click` event to call the `ClearAll` function. Calling the `ClearAll` function will clear all key-value pairs that are currently stored in local storage.

The `Save` function is responsible for saving the key-value pair into the web browser's local storage:

```
func Save() {

  itemKey := D.GetElementByID("itemKey").(*dom.HTMLInputElement)
  itemValue := D.GetElementByID("itemValue").(*dom.HTMLInputElement)

  if itemKey.Value == "" {
    return
  }

  SetKeyValuePair(itemKey.Value, itemValue.Value)
  itemKey.Value = ""
  itemValue.Value = ""
  DisplayStorageContents()
}
```

We get the text input fields for the key and the value (shown in bold) using the `GetElementByID` method of the `Document` object. In the `if` conditional block, we check to see if the user has not entered a value for the `Key` input text field. If they have not entered a value, we return from the function.

If the user has entered a value into the `Key` input text field, we continue forward. We call the `SetKeyValuePair` function and provide the values for `itemKey` and `itemValue` as input parameters to the function.

We then set the `value` property of both `itemKey` and `itemValue` to an empty string, to clear the input text field, so that the user can easily add new entries later without having to manually clear the text in these fields.

Finally we call the `DisplayStorageContents` function, which is responsible for displaying all the current entries in local storage.

Let's take a look at the `SetKeyValuePair` function:

```
func SetKeyValuePair(itemKey string, itemValue string) {
    localStorage.Call("setItem", itemKey, itemValue)
}
```

Inside this function, we simply call the `setItem` method of the `localStorage` object, passing in the `itemKey` and `itemValue` as input parameters to the function. At this point, the key-value pair entry will be saved to the web browser's local storage.

The `DisplayStorageContents` function is responsible for displaying all the key-value pairs that are in local storage inside the `itemList` element, which is a `dl` (description list) element:

```
func DisplayStorageContents() {
    itemList := D.GetElementByID("itemList")
    itemList.SetInnerHTML("")

    for i := 0; i < localStorage.Length(); i++ {

        itemKey := localStorage.Call("key", i)
        itemValue := localStorage.Call("getItem", itemKey)

        dtElement := D.CreateElement("dt")
        dtElement.SetInnerHTML(itemKey.String())

        ddElement := D.CreateElement("dd")
        ddElement.SetInnerHTML(itemValue.String())

        itemList.AppendChild(dtElement)
        itemList.AppendChild(ddElement)
    }
}
```

We call the `SetInnerHTML` method with an input value of empty string to clear the contents of the list.

We iterate through all the entries in local storage using a `for` loop. For each key-value pair present, we get `itemKey` and `itemValue` by calling the `localStorage` object's `key` and `getItem` methods, respectively.

We use a `dt` element (`dtElement`) to display the key. A `dt` element is used to define a term in a description list. We use a `dd` element (`ddElement`) to display the value. A `dd` element is used to describe a term in a description list. Using the description list and its associated elements to display key-value pairs, we are using a semantic friendly approach to displaying the key-value pairs on the web page. We append the `dt` and `dd` elements to the `itemList` object by calling its `AppendChild` method.

The `ClearAll` function is used to remove all the key-value pairs that have been saved in local storage:

```
func ClearAll() {
    localStorage.Call("clear")
    DisplayStorageContents()
}
```

We call the `clear` method of the `localStorage` object, and then make a call to the `DisplayStorageContents` function. If everything is working properly, all the items should be cleared, and we should see no values appear in the `itemList` element once the **Clear All** button has been clicked.

## Running the local storage demo

You can access the local storage demo at <http://localhost:8080/localstorage-demo>.

Let's add a new key-value pair to local storage. In the **Key** input text field, let's add the `"foo"` key, and in the **Value** input text field, let's add the `"bar"` value. Click on the **Save** button to add the new key-value pair to local storage.

*Figure 3.21* shows the newly created key-value pair appear, after clicking on the **Save** button:

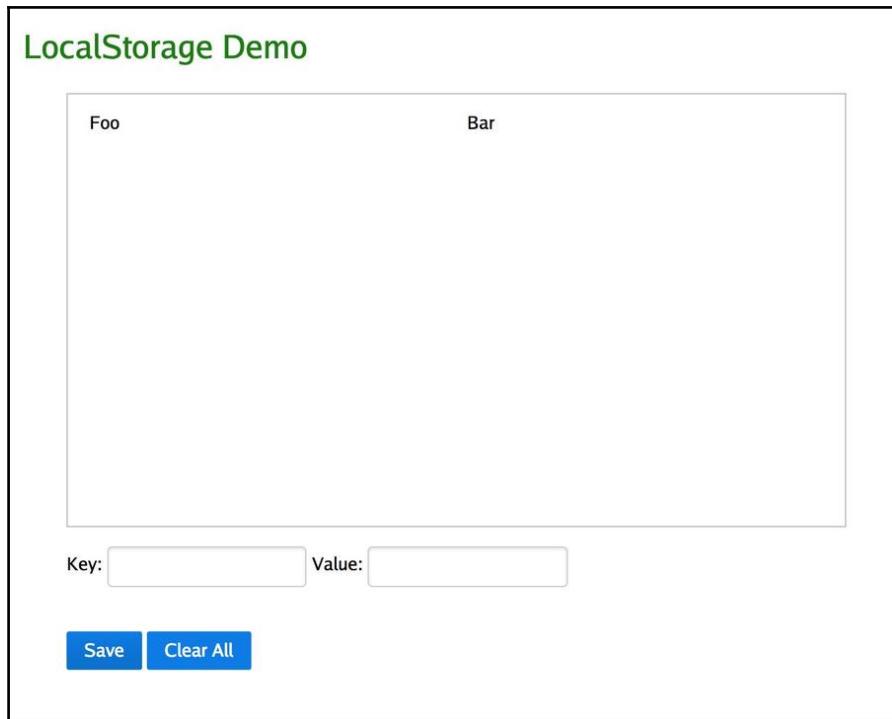


Figure 3.21: The Local Storage Inspector showing a newly added key-value pair

Try refreshing the web page, and after that, try restarting the web browser and returning to the web page. Notice that in these scenarios, local storage still retains the key-value pairs that were saved. Upon clicking on the **Clear All** button, you will notice that the `itemList` has been cleared, as shown in *Figure 3.20*, since local storage has been emptied of all key-value pairs.

The local storage inspector that we just created is especially handy to inspect key-value pairs that have been populated by third party JavaScript solutions, used by our client-side web application. If you land on the local storage demo page, after viewing the image carousel on the IGWEB home page, you will notice that the itemList is populated with the key-value pairs shown in *Figure 3.22*:

The screenshot shows a web application titled "LocalStorage Demo". The main content area displays a table of key-value pairs. The keys listed are: tADe, tADu, tAE, tC, tMQ, tSP, tTDe, tTDu, tTE, tTf, and tnsApp. The corresponding values are: animationDelay, animationDuration, animationend, calc, true, true, transitionDelay, transitionDuration, transitionend, transform, and Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_11\_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36. Below the table, there are input fields for "Key:" and "Value:", and two buttons: "Save" and "Clear All".

Key	Value
tADe	animationDelay
tADu	animationDuration
tAE	animationend
tC	calc
tMQ	true
tSP	true
tTDe	transitionDelay
tTDu	transitionDuration
tTE	transitionend
tTf	transform
tnsApp	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.94 Safari/537.36

Key:  Value:

**Save** **Clear All**

Figure 3.22: Local Storage Demo displaying key-value pairs that were populated by the image carousel

These key-value pairs were populated by the image carousel, which we will implement as a reusable component in Chapter 9, *Cogs – Reusable Components*.

## Summary

In this chapter, we introduced you to programming with Go on the front-end using GopherJS. We introduced you to the DOM and showed how you can access and manipulate it using GopherJS. We walked you through several microexamples to get you acquainted with what coding with GopherJS looks like. We then proceeded to show you fully fleshed out examples.

We showed you how to display the alert dialog and display a custom message. We also showed you how to change the CSS style property of an element. We proceeded to show you how to call JavaScript's `typeof` operator from within the confines of Go using the `jsbuiltin` package. We showed you how to create a simple lowercase text transformer and demonstrated how to send an `XHR Post` using the `xhr` package. We also showed you how to render an inline Go template, and finally, we showed you how to build a local storage inspector.

In Chapter 4, *Isomorphic Templates*, we will introduce isomorphic templates, which are templates that can be rendered either on the server side or the client side.

# 4

## Isomorphic Templates

In the previous chapter, we provided an introduction to GopherJS, and we covered code examples to perform various front-end operations. One of the interesting tasks that we performed on the client side, was template rendering, using an inline Go template.

However, rendering inline Go templates in the web browser is not a maintainable solution. For one thing, mixing HTML code from an inline Go template, along with Go source code, can become an unmaintainable arrangement as the project codebase grows. In addition to this, real-world web applications often require having multiple template files that are often nested together with a layout hierarchy in mind. In addition to that, the template package from Go's standard library was designed particularly for templates rendered on the server side since it depends on accessing template files from the filesystem.

To fully unleash the power of templates across environments, we need a solution that provides more flexibility to render any template within a set of templates for a given project. This flexibility can be found by implementing isomorphic template rendering using the `isokit` package from the Isomorphic Go toolkit. Using the functionality from the `isokit` package, we can render a template belonging to a template set, either on the server side or on the client side, and we'll show you exactly how that's done in this chapter.

Specifically, we will cover the following topics in this chapter:

- The web template system
- IGWEB page structure
- Template categories
- Custom template functions
- Feeding data to the content template
- Isomorphic template rendering

# The web template system

In web programming, a **web template** is a text document that describes the format in which a web page should appear to the user. In this book, we will focus on web templates from Go's `html/template` package—the package that implements data-driven templates suitable for use in web applications.

Web templates (we'll refer to them simply as *templates* moving forward) are text documents, that are typically implemented in HTML, and may contain special commands that are embedded inside of them. In Go, we refer to these commands as *actions*. We denote actions in templates by placing them inside a pair of opening and closing double curly braces—`{ {` and `} }`.

Templates form the means to present data to the user in a manner that is intuitive and acceptable. In fact, you can think of a template as the means by which we dress up data.



In this book, we will use the file extension of `.tmpl` to designate Go template source files. You may notice that some other Go projects use the file extension of `.html` instead. There is no hard set rule to prefer one extension over the other, just remember that once you choose which file extension to use, it's best to stick with it to promote uniformity in the project codebase.

Templates are used in conjunction with a **web template system**. In Go, we have the robust `html/template` package from the standard library to render templates. When we use the term *render template*, we refer to the process by which one or more templates, along with a **data object**, is processed through a **template engine** that generates HTML web page output, as depicted in *Figure 4.1*:

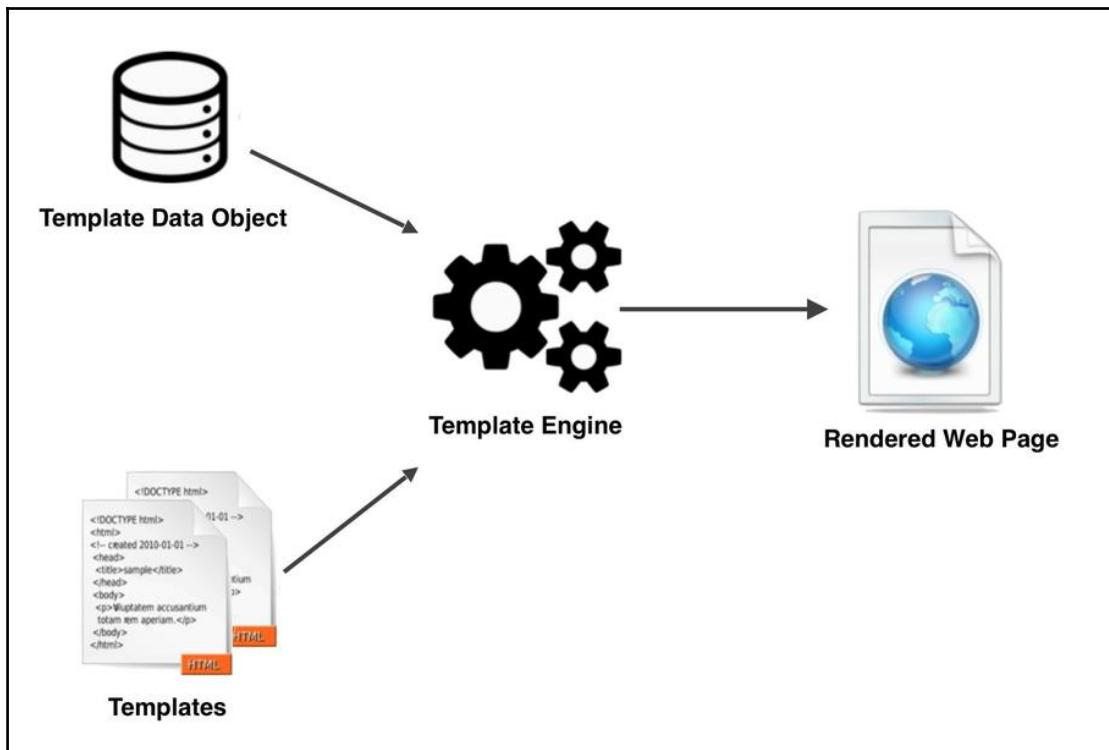


Figure 4.1: How a web page is rendered by a web template system

The key components in *Figure 4.1*, the **template engine**, the **template data object**, and the **templates**, can be classified as comprising a **web template system**. Each component plays an important role toward rendering the web page output, and in the following sections, we'll consider the role that each component plays in this process of producing the HTML output to display in the web browser. In this chapter, we will build IGWEB's **About** page.

## The template engine

The major responsibility of a template engine is to take one or more template files, along with a data object, and generate text output. In our specific field of study, isomorphic web development, this text output is in the HTML format and can be consumed by web clients. In Go, the `html/template` package can be considered as our template engine.

The template engine is activated by the route handler, when it's time to serve the HTML output. From the perspective of an isomorphic web application, the template engine can be activated by both the server-side route handler and the client-side route handler.

When the template engine is activated from the server-side route handler, the produced HTML web page output is written out to the web client in the server response, by the web server instance, using `http.ResponseWriter`. This activity typically occurs when a page on the website is accessed for the first time, and the initial page request is serviced on the server side. In this scenario, the HTML that is returned from the template engine describes a full HTML web page document and includes the opening and closing `<html>` and `<body>` tags.

When the template engine is activated from the client-side route handler, the produced HTML content is rendered in a designated area of a fully rendered web page. We will be rendering HTML content on the client side for a given web page on IGWEB in a designated area, known as the *primary content area*. We will cover the primary content area, a `<div>` container, later in this chapter. Client-side template rendering typically occurs on a subsequent user interaction with the website, such as when the user clicks on a link in the navigation bar to access a particular page on the website. In this scenario, the HTML that is returned from the template engine represents only a portion of a HTML web page.

Worth noting is that Go comes with two template packages. The `text/template` package is used to generate text, and the `html/template` package is meant to be used to generate HTML output. The `html/template` package provides the same interface as the `text/template` package. In this book, we are specifically interested in generating the HTML web page output, and this is why we will be focusing on the `html/template` package. The `html/template` package provides us with extra security by generating the HTML output that is safe against code injection, which the regular `text/template` package doesn't do. This is why it's best to use the `html/template` package for the purpose of web development.

## The template data object

The major responsibility of the template data object (or simply the *data object*) is to supply a given template with data that is to be presented to the user. In the **About** page that we will be constructing, there are two pieces of data that need to be presented. The first need is subtle, it's the title of the web page that will be displayed in the web browser's title bar window, or as the title of the web browser tab containing the web page. The second data need is more profound, it is the data object, the list of gophers that should be displayed on the **About** page.

We will use the following `About` struct from the `templatedata` package, defined in the `shared/templatedata/about.go` source file, to fulfill the data needs of the **About** page:

```
type About struct {
    PageTitle string
    Gophers []*models.Gopher
}
```

The `PageTitle` field represents the web page title that should be displayed in the web browser's title bar (or as the title of the web browser tab). The `Gophers` field is a slice of pointers to a `Gopher` struct. The `Gopher` struct represents a gopher, a member of the IGWEB team, that should be displayed on the **About** page.

The definition for the `Gopher` struct can be found in the `gopher.go` source file found in the `shared/models` folder:

```
type Gopher struct {
    Name string
    Title string
    Biodata string
    ImageURI string
    StartTime time.Time
}
```

The `Name` field represents the name of the Gopher. The `Title` field represents the title that the IGWEB organization has bestowed to a particular gopher. The `Biodata` field represents a brief bio about a particular gopher. We used a loren ipsum generator, to generate some random gibberish in Latin, to populate this field. The `ImageURI` field is the path to the image of the gopher that should be displayed, relative to the server root. The gopher's image will be displayed on the left-hand side of the page, and the the gopher's profile information will be displayed on the right-hand side of the page.

Finally, the `StartTime` field represents the date and time that the gopher joined the IGWEB organization. We will be displaying the gopher's start time in the standard time format, and later in this chapter we will learn how to display the start time using Ruby-style formatting by implementing a custom template function. In *Chapter 9, Cogs – Reusable Components*, we will learn how to display the start time in the human readable time format.

## The templates

Templates are responsible for presenting information to the user in an intuitive and understandable manner. Templates form the view layer of the isomorphic web application. Go templates are a combination of standard HTML markup mixed with a lightweight template language that provides us with the means to perform token substitution, looping, conditional flow of control, template nesting, and the ability to call custom template functions within the template using pipelining constructs. All of the aforementioned activities can be performed using template actions, and we will be using them throughout this book.

The templates for the IGWEB project can be found in the `shared/templates` folder. They are considered as isomorphic templates, since they can be utilized on the server side and on the client side. We will now explore the web page layout organization of IGWEB, and directly after that, we will examine the templates needed to implement the structure of a web page on IGWEB.

## IGWEB page structure

*Figure 4.2* depicts a wireframe design illustrating the structure of a web page on IGWEB. The figure provides us with a good idea of the fundamental layout and navigational needs of the website:

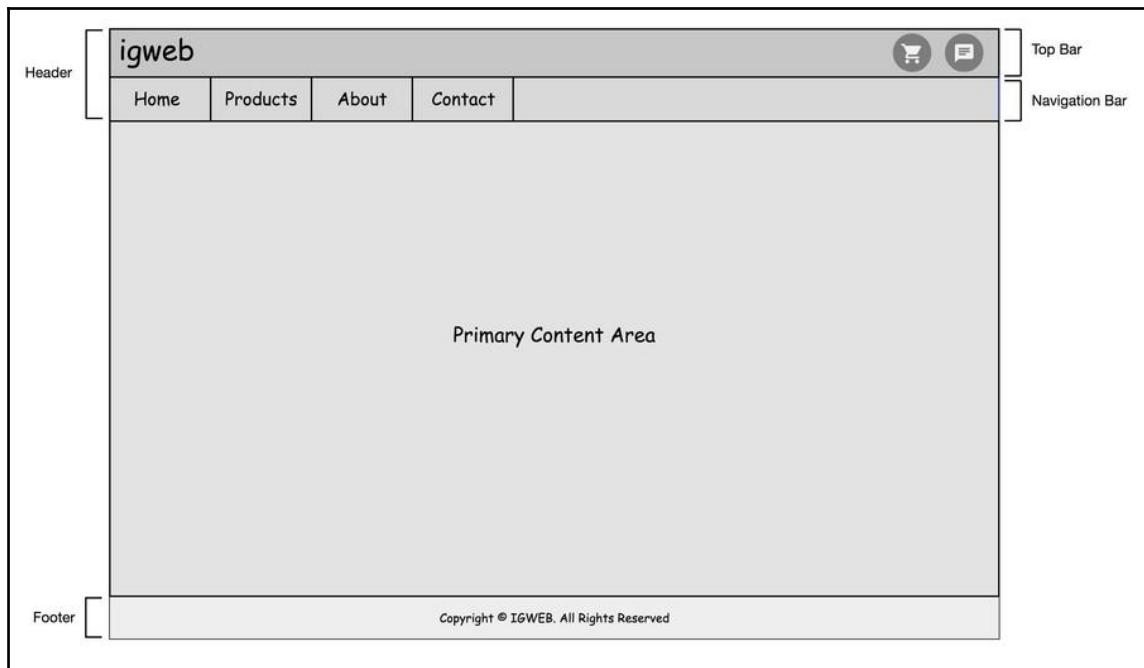


Figure 4.2: IGWEB wireframe design

By organizing the web page structure into these individual areas, we can demarcate the unique function(s) that each area plays in the web page structure as a whole. Let's go ahead and examine each individual area that comprises the page structure:

1. The header
2. The primary content area
3. The footer

## The header

The header area, as depicted in *Figure 4.2*, appears at the top of the web page. It marks the beginning of the web page, and it is useful for the purposes of branding, navigation, and user interactivity. It is comprised of the top bar and the navigation bar.

## The top bar

As depicted in *Figure 4.2*, the top bar is a subarea that exists within the header. At the left-most side of the top bar is the logo for IGWEB. Besides having an obvious function for branding purposes, the logo also serves as a navigational component, since when the user clicks on it, they will be returned to the home page. At the right-most side of the top bar are ancillary user controls to activate specific functionality—the shopping cart and the live chat feature.

## The navigation bar

The navigation bar, as depicted in *Figure 4.2*, is a subarea that exists within the header. The navigation area consists of links to various pages on the website.

## The primary content area

The primary content area, as depicted in *Figure 4.2*, is sandwiched between the header area and the footer area. The contents of an individual web page will be displayed here. For example, the **About** page will display the pictures and profile information for the IGWEB team gophers in the primary content area.

## The footer

The footer area, as depicted in *Figure 4.2*, appears at the bottom of the web page. It contains the copyright notice of the website. The footer marks the end of the web page.

Now that we have established the web page structure for IGWEB, we will learn how we can implement the structure using a preplanned hierarchy of Go templates. To improve our understanding, we will organize templates into categories based on their functional purpose.

# Template categories

Organizing templates into categories based on their functional purpose allows us to be productive when implementing the structure of a web page. Templates can be organized into the following three categories, based on the role they serve in implementing the web page structure:

- Layout templates
- Partial templates
- Regular templates

**Layout templates** describe the general layout of the entire web page. They provide us with a bird's-eye view of the page structure and give us a sense of how all the other templates fit in.

**Partial templates** contain only a part of the web page, and hence they are named **partials**. They are partial in nature, since they are meant to fulfill a particular need within a region of the web page, such as displaying the footer of the web page.

**Regular templates** contain content that is meant for a specific section of the website, and this content is meant to be displayed in the primary content area. In the following sections, we will examine each template category, and consider the respective template implementation performed for each category.

## Layout templates

A page layout template, also known as a **layout template**, holds the structure of the entire web page. Since they define the overall structure for the web page, they require other templates (partial and regular) to complete them. For isomorphic web applications, these types of templates are used to render the web page on the server side for the initial web page response, that is sent to the client. In the IGWEB project, we place layout templates in the `shared/templates/layouts` folder.

## The web page layout template

Here's the web page layout template found in the `webpage_layout.tpl` source file in the `shared/templates/layouts` directory:

```
<!doctype html>
<html>
  {{ template "partials/header_partial" . }}

  <div id="primaryContent" class="pageContent">
    {{ template "pagecontent" . }}
  </div>

  <div id="chatboxContainer">
  </div>

  {{ template "partials/footer_partial" . }}
</html>
```

Notice that the layout template covers the entire web page, from the opening `<html>` tag to the closing `</html>` tag. The layout template issues the `template` actions (shown in bold) to render the `header` partial template, the `pagecontent` regular template, and the `footer` partial template.

The dot `.` between the `partials/header_partial` template name and the closing pair of curly braces, `}}`, is known as an action. The template engine considers this to be a command that is to be replaced with the value of the `data` object that is fed into the template upon the execution of the template. By placing the dot here, we ensure that the `header` partial template, which is responsible for displaying the content in the header area of the website, has access to the `data` object that was fed into the template. Notice that we have done the same thing, for the `pagecontent` template, and the `partials/footer_partial` template.

## Partial templates

Partial templates, also known as **partials**, typically hold portions of content for a specific area of the web page. Examples of a partial template include the header and footer of the web page. Partial templates for the header and footer come in very handy when including them in the page layout template, since the header and footer will be preset on all web pages in the website. Let's examine how the header and footer partial templates are implemented. In the IGWEB project, we place partial templates in the `shared/templates/partials` folder.

## The header partial template

Here's an example of the header partial found in the `header_partial.tpl` source file in the `shared/templates/partials` folder:

```
<head>
  <title>{{.PageTitle}}</title>
  <link rel="icon" type="image/png"
    href="/static/images/isomorphic_go_icon.png">
  <link rel="stylesheet" href="/static/css/pure.css">
  <link rel="stylesheet" type="text/css" href="/static/css/cogimports.css">
  <link rel="stylesheet" type="text/css"
    href="/static/css/alertify.core.css" />
  <link rel="stylesheet" type="text/css"
    href="/static/css/alertify.default.css" />
  <link rel="stylesheet" type="text/css" href="/static/css/igweb.css">
  <script type="text/javascript" src="/static/js/alertify.js"
  type="text/javascript"></script>
  <script src="/static/js/cogimports.js" type="text/javascript"></script>
  <script type="text/javascript" src="/js/client.js"></script>
</head>
<body>

<div id="topbar">{{template "partials/topbar_partial"}}</div>
<div id="navbar">{{template "partials/navbar_partial"}}</div>
```

Between the opening `<head>` and closing `</head>` tags, we include the icon for the website along with external CSS style sheets and external JavaScript source files. The `igweb.css` stylesheet defines the styles for the IGWEB website (shown in bold). The `client.js` JavaScript source file, is the JavaScript source file for the client-side web application, which is transpiled into JavaScript by GopherJS (shown in bold).

Notice that we render the top bar, and the navigation partial templates, inside the header partial template using the `template` actions (shown in bold). We don't include the dot `.` here, since these partials don't need access to the `data` object. Both the content for the top bar and the navigation bar are within their own respective `<div>` containers.

## The top bar partial template

Here's the top bar partial template found in the `topbar_partial tmpl` source file in the `shared/templates/partials` folder:

```
<div id="topbar" >
  <div id="logoContainer" class="neon-text"><span><a href="/index">igweb</a></span></div>
  <div id="siteControlsContainer">
    <div id="shoppingCartContainer" class="topcontrol" title="Shopping Cart"><a href="/shopping-cart"></a></div>
    <div id="livechatContainer" class="topcontrol" title="Live Chat"></div>
  </div>
</div>
```

The top bar partial template is a good example of a static template where there's no dynamic action going on. There are no `template` actions defined within it, and its main purpose is to include the HTML markup to render the website logo, the shopping cart icon, and the live chat icon.

## The navigation bar partial template

Here's an example of the navigation bar partial template found in the `navbar_partial tmpl` source file in the `shared/templates/partials` folder:

```
<div id="navigationBar">
  <ul>
    <li><a href="/index">Home</a></li>
    <li><a href="/products">Products</a></li>
    <li><a href="/about">About</a></li>
    <li><a href="/contact">Contact</a></li>
  </ul>
</div>
```

The navigation bar partial template is also a static template. It contains the `div` container that contains the list of navigation links that make up IGWEB's navigation bar. These links allow users to access the **Home**, **Products**, **About**, and **Contact** pages.

## The footer partial template

Here's an example of the footer partial template found in the `footer_partial.tpl` source file in the `shared/templates/partials` folder:

```
<footer>
  <div id="copyrightNotice">
    <p>Copyright &copy; IGWEB. All Rights Reserved</p>
  </div>
</footer>
</body>
```

The footer partial template is also a static template, whose current, sole purpose is to have the HTML markup that contains the copyright notice for the IGWEB website.

Now that we've covered all the partial templates that comprise the web page structure, it's time to examine what a regular template looks like, both from the perspective of the server side and that of the client side.

## Regular templates

A **regular template** is used to hold the primary content that is to be displayed on the web page. For example, in the **About** page, the primary content would be the information about the gophers on the IGWEB team along with their individual pictures.

In this chapter, we will build the **About** page. We can see exactly what will go into the primary content area in the **About** page, by examining its wireframe design depicted in *Figure 4.3*:

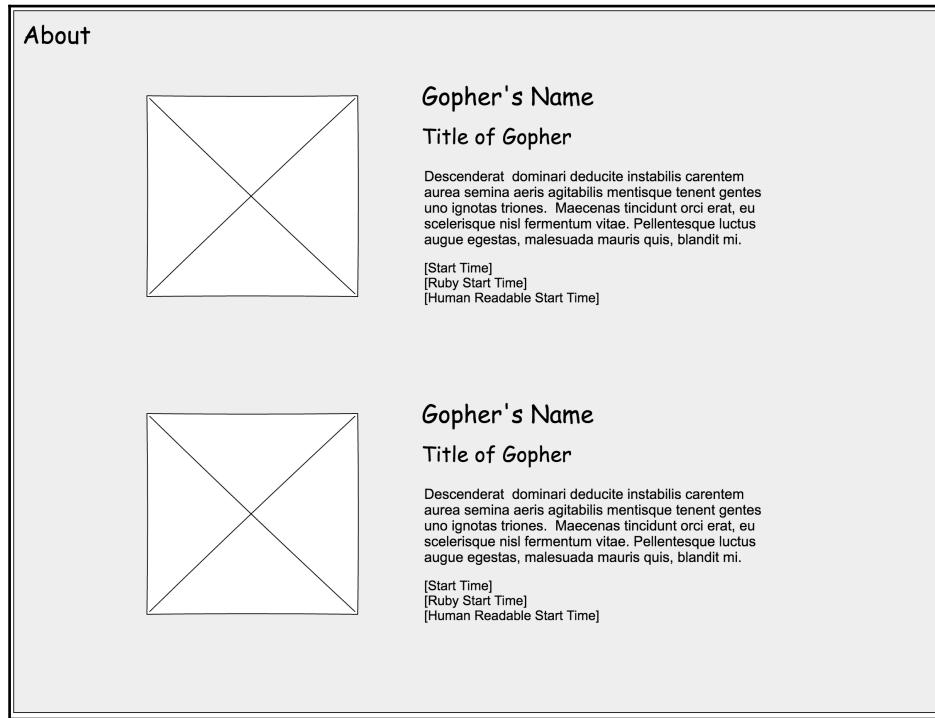


Figure 4.3: Wireframe design for the about page

For each gopher on the IGWEB team, we will display the gopher's picture, its name, its title, and a brief description about its role (randomly generated Latin gibberish). We will also display the date/time that the gopher joined the IGWEB team in several different time formats.

We will render the **About** page in two markedly different ways, depending on whether the rendering takes place on the server side or the client side. On the server side, when we render the **About** page, we are in need of a page template, that is a regular template, that contains the layout for the whole web page, in addition to containing the content of the **About** page. On the client side, we only need to render the content contained in the **About** page to populate the primary content area, since the web page has already been rendered for us from the initial page load.

At this point, we can define two subcategories of regular templates: a *page template* that will serve our server-side rendering needs, and a *content template* that will serve our client-side rendering needs. In the IGWEB project, we will place regular templates in the shared/templates folder.

## The page template for the about page

Here's an example of the page template for the **About** page, from the `about_page.tpl` source file in the shared/templates folder:

```
 {{ define "pagecontent" }}  
 {{ template "about_content" . }}  
 {{ end }}  
 {{ template "layouts/webpage_layout" . }}
```

We use the `define` action in the page template, to define the region of the template that contains the template section that we have declared as the `pagecontent` section. We have a corresponding `end` action to mark the end of the `pagecontent` section. Notice that we have a `template` action in between the `define` and `end` actions to include the template named `about_content`. Also note that we provide the dot (.) action to pass the `data` object to the `about_content` template.

This page template is a good example that shows how we can render layout templates inside of a regular template. In the last line of the template, we declare a `template` action to load the layout template for the web page that is named `layouts/webpage_layout`. Again, notice that we provide the dot (.) action to pass the `data` object to the web page layout template.

Now that we've examined the `about_page` template, it's time to examine the `about_content` template.

## The content template for the about page

Here's an example of the content template, which is rendered into the primary content area found in the **About** page, from the `about_content.tpl` source file in the shared/templates folder:

```
<h1>About</h1>  
  
<div id="gopherTeamContainer">  
 {{range .Gophers}}
```

```
<div class="gopherContainer">

  <div class="gopherImageContainer">
    
  </div>

  <div class="gopherDetailsContainer">
    <div class="gopherName"><h3><b>{{ .Name }}</b></h3></div>
    <div class="gopherTitle"><span>{{ .Title }}</span></div>
    <div class="gopherBiodata"><p>{{ .Biodata }}</p></div>
    <div class="gopherStartTime">
      <p class="standardStartTime">{{ .Name }} joined the IGWEB team on
      <span class="starttime">{{ .StartTime }}</span>.
      <p class="rubyStartTime">That's <span
      class="starttime">{{ .StartTime | rubyformat }}</span> in Ruby date
      format.</p>
      <div class="humanReadableGopherTime">That's <div id="Gopher-
      {{ .Name }}" data-starttimeunix="{{ .StartTime | unixformat }}" data-
      component="cog" class="humanReadableDate starttime"></div> in Human
      readable format.</div>
    </div>
  </div>
</div>

{{ end }}
```

We use the `range` action to iterate through the `Gophers` property of the data object supplied to the template (shown in bold). Notice that we use the dot (.) action to access the `Gophers` property of the data object. Remember, the `Gophers` property is a slice of pointers to a `Gopher` struct. We print out the fields of each `Gopher` struct, in their designated area in the template, using the dot (.) action (shown in bold). We use the `end` action to denote the end of the `range` looping action (shown in bold).

Important to note is that the content template is required on both the server side and the client side. Remember, that on the server side, the full web page layout needs to be rendered, in addition to the content template. On the client side, we only need to render the content template.

Notice that in the last two places where we print the `StartTime` field using the dot (.) action, we make use of the pipe (|) operator to format the `StartTime` field using a custom function. First, we use the `rubyformat` function to show the `StartTime` value in Ruby date/time format, and then we use the `unixformat` function to populate the "data-starttimeunix" attribute with the Unix time representation of the `StartTime` value. Let's take a look at where these custom functions are defined in the IGWEB project codebase.

## Custom template functions

We defined our custom template functions in the `funcs.go` source file found in the `shared/templatefuncs` folder:

```
package templatefuncs

import (
    "strconv"
    "time"
)

func RubyDate(t time.Time) string {
    layout := time.RubyDate
    return t.Format(layout)
}

func UnixTime(t time.Time) string {
    return strconv.FormatInt(t.Unix(), 10)
}
```

The `RubyDate` function displays a given time using the time layout specified by the `time.RubyDate` constant. We call the function in a template using the `rubyformat` function name.

As noted before, inside the `about` content template (`shared/templates/about_content tmpl`), we use the pipe (|) operator to apply the `rubyformat` function to the `StartTime` as shown here:

```
<p class="rubyStartTime">That's <span class="starttime">{{.StartTime |  
rubyformat}}</span> in Ruby date format.</p>
```

In this way, custom template functions provide us with the flexibility to format values in our templates to meet the unique needs that our project may require. You might be wondering, how we map the `rubyformat` name to the `RubyDate` function. We create a template function map that contains this mapping; we will cover how we can use the template function map across environments, later in this chapter.



The fact that the three subfolders, `templates`, `templatedata`, and `templatefuncs`, reside in the `shared` folder imply that the code found within these folders can be used across environments. In fact, any code contained within the `shared` folder and its subfolders, is code that's meant for sharing across environments.

We will be covering the `UnixTime` function, referred to as the `unixformat` function in the template, in [Chapter 9, Cogs – Reusable Components](#).

## Feeding data to the content template

The data object that we will feed to the `About` content template is a slice of pointers to `Gopher` structs that represent each gopher on the IGWEB team. The plan for feeding data to our template is to obtain a slice of gophers from the Redis datastore and populate the `Gophers` property of the template data object for the `About` page along with the data object's `PageTitle` property.

We call the `GetGopherTeam` method on our datastore object to obtain a slice of gophers that belongs to the IGWEB team. Here's the declaration of the `GetGopherTeam` function from the `redis.go` source file found in the `common/datastore` folder:

```
func (r *RedisDatastore) GetGopherTeam() []*models.Gopher {  
  
    exists, err := r.Cmd("EXISTS", "gopher-team").Int()  
  
    if err != nil {  
        log.Println("Encountered error: ", err)  
        return nil  
    } else if exists == 0 {  
        return nil  
    }  
  
    var t []*models.Gopher  
    jsonData, err := r.Cmd("GET", "gopher-team").Str()  
  
    if err != nil {
```

```
    log.Println("Encountered error when attempting to fetch gopher team data
from Redis instance: ", err)
    return nil
}

if err := json.Unmarshal([]byte(jsonData), &t); err != nil {
    log.Println("Encountered error when attempting to unmarshal JSON gopher
team data: ", err)
    return nil
}

return t
}
```

The `GetGopherTeam` function checks if the `gopher-team` key exists in the `Redis` database. The slice of gophers are stored as JSON-encoded data within the `Redis` database. If the `gopher-team` key exists, we attempt to `unmarshal` the JSON-encoded data into the `t` variable, which is a slice of pointers to the `Gopher` structs. If we were able to successfully `unmarshal` the JSON data we will return the `t` variable.

At this point, we've created the means to fetch the data for our team of gophers that will be displayed on the **About** page. You might be wondering, why can't we just feed the `about` content template with the slice of gophers, as the `data` object, and call it a day? Why do we need to pass a `data` object with a type of `templatedata>About` to the `about` content template?

The one-word answer to both of these questions is *extensibility*. Currently, the `About` section doesn't just need the slice of `Gophers`—it also needs a page title that will be displayed in the web browser's title window and/or in the web browser tab. So for all the sections of `IGWEB`, we have created accompanying structs to model the individual data needs for each page of the website in the `shared/templatedata` folder. Since the `templatedata` package is found in the `shared` folder, the `templatedata` package is isomorphic and can be accessed across environments.

We defined the `About` struct in the `about.go` source file found in the `shared/templatedata` folder:

```
type About struct {
    PageTitle string
    Gophers  []*models.Gopher
}
```

The `PageTitle` field of the `string` type is the title of the **About** page. The `Gophers` field is a slice of pointers pointing to the `Gopher` structs. This slice represents the gopher team that will be displayed on the `about` page. As we saw earlier in this chapter, we'll use the `range` action in the content template to iterate through the slice and display the profile information for each gopher.

Coming back to the topic of extensibility, the fields defined for the structs in the `templatedata` package are not meant to stay stationary and unchanging. They are meant to change over time to accommodate the future needs of the particular web page they are meant to serve.

For example, if an IGWEB product manager decides that they should have pictures of the gopher team members working, studying, and playing in the office, for public relations purposes, they can easily accommodate the request by adding a new field to the `About` struct called `OfficeActivityImages`. This new field could be a slice of strings that denotes the server relative path to the images of gophers, engaging in the various activities, that should be displayed on the **About** page. We would then add a new section in our template, where we would `range` through the `OfficeActivityImages` slice, and display each image.

At this point, we have the data needs for the **About** page satisfied, and we have all our templates lined up. It's now time to focus on how to perform the rendering of the templates, on both the server side and the client side. This is where isomorphic template rendering comes into play.

## Isomorphic template rendering

Isomorphic template rendering allows us to render and reuse templates across environments. The traditional procedure to render templates in Go, which relies on accessing templates through the file system, comes with certain limitations that prevent us from rendering these same templates on the client side. It's important for us to acknowledge these limitations to fully appreciate the benefits that isomorphic template rendering presents for us.

## Limitations of filesystem-based template rendering

When it comes to sharing template rendering responsibilities with the client, there are certain limitations in the template rendering workflow that we need to acknowledge. First, and foremost, template files are defined on the web server.

Let's consider an example that follows the classic web application architecture to fully understand the limitations we face. Here's an example of server-side template rendering using a template file, `edit.html`, taken from the *Writing Web Applications* article (<https://golang.org/doc/articles/wiki/>) from the Go website:

```
func editHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[len("/edit/"):]
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    t, _ := template.ParseFiles("edit.html")
    t.Execute(w, p)
}
```

The `editHandler` function is responsible for handling the `/edit` route. The last two lines (shown in bold) are of particular interest for our consideration. The `ParseFiles` function in the `html/template` package is called to parse the `edit.html` template file. After the template is parsed, the `Execute` function in the `html/template` package is called to execute the template along with the `p` data object, which is a `Page` struct. The produced web page output is then written out to the client as a web page response using `http.ResponseWriter`, `w`.



The *Writing Web Applications* article from the Go website is an excellent article to learn and understand classic, sever-side web application programming with Go. I highly recommend that you read this article: <https://golang.org/doc/articles/wiki/>.

The drawback of rendering templates in this manner is that we are anchored to the server-side file system, where the `edit.html` template file resides. The dilemma we face is that the client needs access to the contents of template files in order to render a template on the client-side. It is not possible to make the `ParseFiles` function call on the client side, because we don't have access to any template file that can be read on the local file system.

The robust security sandbox implemented in modern web browsers, prevents clients from accessing template files from the local file system, as it rightly should. In contrast, calling the `ParseFiles` function makes sense, from the server side, since the server-side application can actually access the server-side filesystem, where the templates reside.

So how do we get past this roadblock? The `isokit` package comes to our rescue by providing us the capability to gather a group of templates from the server-side file system, and create an in-memory template collection, called a template set.

## The in-memory template set

The `isokit` package comes with the functionality to render templates in an isomorphic manner. In order to think isomorphically, when it comes to template rendering, we have to step away from the thought process of rendering a template from the file system, as we have been much accustomed to in the past. Instead, we have to think in terms of maintaining a set of templates in-memory, where we could access a particular template by the name that we have given to it.

When we use the term, in-memory, we are not referring to an in-memory database, but rather having the template set persisted in the running application itself, whether it's on the server side or the client side. The template set stays resident in memory for the application to utilize while it is running.

The `Template` type from the `isokit` package represents an isomorphic template, one that can be rendered either on the server side or the client side. In the type definition of `Template`, notice that the `*template.Template` type is embedded:

```
type Template struct {
    *template.Template
    templateType int8
}
```

Embedding the `*template.Template` type allows us to tap into all the functionality from the `Template` type defined in the `html/template` package. The `templateType` field indicates the type of template we are dealing with. Here's the constant grouping declaration with all the possible values for this field:

```
const (
    TemplateRegular = iota
    TemplatePartial
    TemplateLayout
)
```

As you can see, the constant grouping declaration has accounted for all the template categories that we will be dealing with: regular templates, partial templates, and layout templates.

Let's take a look at what the `TemplateSet` struct from the `isokit` package looks like:

```
type TemplateSet struct {
    members map[string]*Template
    Funcs template.FuncMap
    bundle *TemplateBundle
    TemplateFilePath string
}
```

The `members` field is `map` with a key of type `string` and a value that is a pointer to an `isokit.Template` struct. The `Funcs` field is an optional function map (`template.FuncMap`) that can be supplied to the template set, to call custom functions inside of a template. The `bundle` field is the template bundle. The `TemplateBundle` is a map, where the key represents the name of the template (a `string` type) and the value is the contents of the template file (also of the `string` type). The `TemplateFilePath` field represents the path where all of the web application's isomorphic templates reside.

Here's what the `TemplateBundle` struct looks like:

```
type TemplateBundle struct {
    items map[string]string
}
```

The `items` field of the `TemplateBundle` struct is simply a map having a key of the `string` type and a value of the `string` type. The `items` map serves an important purpose, it is the data structure that will be `gob` encoded on the server side, and we'll expose it to the client side using the `/template-bundle` server-side route, where it can be retrieved through an XHR call and decoded, as depicted in *Figure 4.4*:

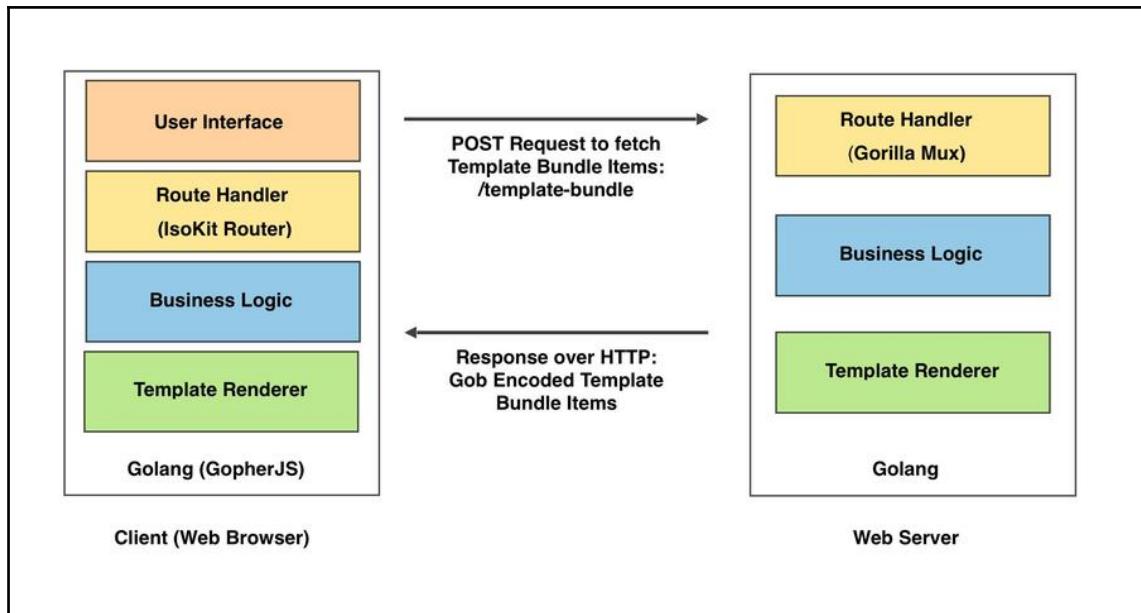


Figure 4.4 How the items in a template bundle get transported to the client-side

The template bundle type plays a critical role since we use it as the basis to recreate the in-memory template set on the client-side. This allows us to provide the full set of templates for use on the client side. Now that we're armed with the knowledge that we can utilize the concept of a template set to isomorphically render templates, let's see how it's done in practice.

## Setting up the template set on the server side

Let's examine the variable declarations at the beginning of the `igweb.go` source file found in the `igweb` folder:

```
var WebAppRoot string
var WebAppMode string
var WebServerPort string
var DBConnectionString string
var StaticAssetsPath string
```

The variables declared here are critical for the proper operation of the web server instance. The `WebAppRoot` variable is responsible for specifying where the `igweb` project folder resides. The `WebServerPort` variable is responsible for specifying on what port the web server instance should run on. The `DBConnectionString` variable is used to specify the connection string to the database. The `StaticAssetsPath` variable is used to specify the directory that contains all of the static (nondynamic) assets for the project. These assets may consist of CSS style sheets, JavaScript source files, images, fonts, and anything else that isn't meant to be dynamic.

We initialize the variables in the `init` function:

```
func init() {
    WebAppRoot = os.Getenv("IGWEB_APP_ROOT")
    WebAppMode = os.Getenv("IGWEB_MODE")
    WebServerPort = os.Getenv("IGWEB_SERVER_PORT")
    DBConnectionString = os.Getenv("IGWEB_DB_CONNECTION_STRING")

    // Set the default web server port if it hasn't been set already
    if WebServerPort == "" {
        WebServerPort = "8080"
    }

    // Set the default database connection string
    if DBConnectionString == "" {
        DBConnectionString = "localhost:6379"
    }

    StaticAssetsPath = WebAppRoot + "/static"
}
```

The `WebAppRoot` and `WebServerPort` variables are obtained from the `IGWEB_APP_ROOT` and `$IGWEB_SERVER_PORT` environment variables, respectively.

We will cover the `WebAppMode` variable and the `$IGWEB_MODE` environment variable in Chapter 11, *Deploying an Isomorphic Go Web Application*.

If the `$IGWEB_SERVER_PORT` environment variable has not been set, the default port is set to 8080.

The `DBConnectionString` variable is assigned the value of "localhost:6379", which is the hostname and port on which the Redis database instance is running.

The `StaticAssetsPath` variable is assigned to the `static` folder, which resides inside the `WebAppRoot` folder.

Let's examine the beginning of the `main` function:

```
func main() {  
  
    env := common.Env{}  
  
    if WebAppRoot == "" {  
        fmt.Println("The $IGWEB_APP_ROOT environment variable must be set before  
        the web server instance can be started.")  
        os.Exit(1)  
    }  
  
    initializeTemplateSet(&env, false)  
    initializeDatastore(&env)  
}
```

Right at the beginning of the `main` function, we check whether the `WebAppRoot` variable has been set, and if it hasn't been set, we exit from the application. One of the biggest advantages of setting the `$IGWEB_APP_ROOT` environment variable, which is used to populate the `WebAppRoot` variable, is that we can issue the `igweb` command from any folder on the system.

Inside the `main` function, we initialize the `env` object. Right after calling the `initializeDatastore` function to initialize the datastore, we make a call to the `initializeTemplateSet` function (shown in bold), passing a reference to the `env` object to the function. This function, as you may have guessed from its name, is responsible for initializing the template set. We will make use of the second argument, of the `bool` type, passed to the function in Chapter 11, *Deploying an Isomorphic Go Web Application*.

Let's examine the `initializeTemplateSet` function:

```
func initializeTemplateSet(env *common.Env, oneTimeStaticAssetsGeneration
bool) {
    isokit.WebAppRoot = WebAppRoot
    isokit.TemplateFilesPath = WebAppRoot + "/shared/templates"
    isokit.StaticAssetsPath = StaticAssetsPath
    isokit.StaticTemplateBundleFilePath = StaticAssetsPath +
"/templates/igweb tmplbundle"

    ts := isokit.NewTemplateSet()
    funcMap := template.FuncMap{"rubyformat": templatefuncs.RubyDate,
"unixformat": templatefuncs.UnixTime}
    ts.Funcs = funcMap
    ts.GatherTemplates()
    env.TemplateSet = ts
}
```

We start by initializing the `isokit` package's exported variables for the `WebAppRoot`, `TemplateFilesPath`, and `StaticAssetsPath` variables. We create a new template set, `ts`, by calling the `NewTemplateSet` function found in the `isokit` package.

Right after we create our template set object, `ts`, we declare a function map, `funcMap`. We have populated our map with two custom functions that will be exposed to our templates. The key for the first function is `rubyformat`, and the value is the `RubyDate` function found in the `templatefuncs` package. This function will return the Ruby format for a given time value. The key for the second function is `unixformat`, and this function will return the Unix timestamp for a given time value. We populate the `Funcs` field of the template set object with the `funcMap` object that we just created. Now, all the templates in our template set have access to these two custom functions.

Up to this point, we've prepped the template set, but we haven't populated the template set's `bundle` field. In order to do this, we must call the `GatherTemplate` method of the `TemplateSet` object, which will gather all the templates found in the directory specified by `isokit.TemplateFilesPath` and all of its subdirectories. The names of the template filenames without the `.tmpl` file extension will be used as the key in the bundle map. The string contents of the template file will be used as the value in the bundle map. If the template is a layout or partial, their respective directory name will be included in the name to refer to them. For example, the `partials/footer tmpl` template will have a name of `partials/footer`.

Now that our template set is fully prepped, we can populate the `TemplateSet` field of the `env` object, so that our server-side application has access to the template set. This comes in handy later on, since it allows us to access the template set from any request handler function defined in our server-side web application, providing us the capability to render any template that exists within the template set.

## Registering the server-side handlers

After we have initialized the template set inside the `main` function of the `igweb.go` source file, we create a new Gorilla Mux router, and we call the `registerRoutes` function to register all the routes of the server-side web application. Let's examine the lines of the `registerRoutes` function that are essential for the proper functioning of the client-side web application:

```
// Register Handlers for Client-Side JavaScript Application
r.Handle("/js/client.js",
  isokit.GopherjsScriptHandler(WebAppRoot)).Methods("GET")
r.Handle("/js/client.js.map",
  isokit.GopherjsScriptMapHandler(WebAppRoot)).Methods("GET")

// Register handler for the delivery of the template bundle
r.Handle("/template-bundle",
  handlers.TemplateBundleHandler(env)).Methods("POST")
```

We register a handler for the `/js/client.js` route and specify that it will be handled by the `GopherjsScriptHandler` function from the `isokit` package. This will associate the route to serving the `client.js` JavaScript source file that was built by running the `gopherjs build` command in the `client` directory.

We handle the `map` file of the `client.js.map` in a similar manner. We register a `/js/client.js.map` route and specify that it will be handled by the `GopherjsScriptMapHandler` function from the `isokit` package.

Now that we've registered the routes for the JavaScript source file and the JavaScript source `map` file that are critical for our client-side application to function, we need to register a route to access the template bundle. We will call the `Handle` method on the `r` router object and specify that the `/template-bundle` route will be handled by the `TemplateBundleHandler` function, found in the `handlers` package. This route will be retrieved by the client through an XHR call, and the server will send the template bundle as `gob` encoded data.

The very last route that we register, which is of particular interest for us right now, is the /about route. Here's the line of code where we register the /about route and associate it with the `AboutHandler` function found in the `handlers` package:

```
r.Handle("/about", handlers.AboutHandler(env)).Methods("GET")
```

Now that we've seen how to set up the template set in our server-side web application, and how we registered the routes that are of importance to us in this chapter, let's go ahead and take a look at the server-side handlers, starting with the `TemplateBundleHandler` function in the `handlers` package.

## Serving the template bundle items

Here's the `TemplateBundleHandler` function found in the `templatebundle.go` source file in the `handlers` folder:

```
func TemplateBundleHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        var templateContentItemsBuffer bytes.Buffer
        enc := gob.NewEncoder(&templateContentItemsBuffer)
        m := env.TemplateSet.Bundle().Items()
        err := enc.Encode(&m)
        if err != nil {
            log.Println("encoding err: ", err)
        }
        w.Header().Set("Content-Type", "application/octet-stream")
        w.Write(templateContentItemsBuffer.Bytes())
    })
}
```

The code to encode data to gob format should look familiar, it's just like how we encoded the cars slice to gob format in the *Transmitting gob encoded data section* from the Chapter 3, *Go on the Front-End with GopherJS*. Inside the `TemplateBundleHandler` function, we first declare `templateContentItemsBuffer`, of the `bytes.Buffer` type, which will hold the gob encoded data. We then create a new gob encoder, `enc`. Right after this, we'll create an `m` variable and assign it the value of the template bundle map. We call the `Encode` method of the `enc` object, and pass in the reference to the `m` map. At this point, `templateContentItemsBuffer` should contain the gob encoded data that represents the `m` map. We will write out a content-type header to specify that the server will be sending out binary data (`application/octet-stream`). We will then write out the binary contents of `templateContentItemsBuffer` by calling its `Bytes` method. In the *Setting up the template set on the client side* section of this chapter, we'll see how the client-side web application picks up the template bundle items, and utilizes it to create a template set on the client side.

## Rendering the about page from the server side

Now that we've seen how the server-side application transmits the template bundle to the client-side application, let's take a look at the `AboutHandler` function found in the `about.go` source file in the `handlers` folder. This is the server-side handler function that is responsible for rendering the **About** page:

```
func AboutHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        gophers := env.DB.GetGopherTeam()
        templateData := templatedata.About{PageTitle: "About", Gophers: gophers}
        env.TemplateSet.Render("about_page", &isokit.RenderParams{Writer: w, Data: templateData})
    })
}
```

The `AboutHandler` function has three responsibilities:

- Fetching the gophers from the datastore
- Creating the template data object
- Rendering the **About** page template

The first line of code, defined in the function, fetches the gopher objects from the datastore, where a gopher object represents an individual gopher team member. In our sample data set, there are three gophers: Molly, Case, and Wintermute.

The second line of code is used to set up the template data object of the `templatedata.About` type. This is the data object that will be fed into the template. The `PageTitle` property of the data object is used to display the page title, and we will populate the `Gophers` property of the object with the slice of gopher objects that are retrieved from the datastore.

In the third, and final line of the handler function, we call the `Render` method of the template set to render the template. The first parameter passed to the method is the name of the template to be rendered. In this case, we have specified that we want to render the `about_page` template. Take note that this is a page template that will not only render the **About** page content but also the entire web page layout, which in addition to the primary content area section will also include the header, the top bar, the navigation bar, and the footer areas of the web page.

The second parameter to the function is the template render parameters (`isokit.RenderParams`). We have populated the `Writer` field, with `http.ResponseWriter`, `w`. Also, we have populated the `Data` field, which represents the data object that should be supplied to the template with the `templateData` object that we had just created.

That's all there is to it. Now we can render this template on the server side. We have now implemented the classic web application architecture flow where the entire web page is rendered from the server side. We can access the **About** page at <http://localhost:8080/about>. Here's how the **About** page looks rendered from the server side:

The screenshot shows the 'About' page of the 'igweb' application. The top navigation bar includes 'Home', 'Products', 'About', and 'Contact' links, along with a shopping cart and message icon. The main content area is titled 'About'. It features two sections: 'Molly' and 'Case'. Each section contains a cartoon gopher character, a title, a role, and a bio. Below each bio is a timestamp indicating when the character joined the team, followed by the date in Ruby format and a human-readable timestamp.

Character	Title	Role	Joining Date (UTC)	Ruby Date Format	Human Readable Format
Molly	Founder	CEO	2017-05-24 17:09:00 +0000 UTC	Wed May 24 17:09:00 +0000 2017	That's 4 months ago in Human readable format.
Case	Resident	Isomorphic Gopher Agent	2017-07-14 18:36:00 +0000 UTC	Fri Jul 14 18:36:00 +0000 2017	That's 3 months ago in Human readable format.

Figure 4.5 The About page rendered from the server side

## Setting up the template set on the client side

Now that we've seen how a web template gets rendered on the server side, it's time to focus on how a web template gets rendered on the client side. The primary entry point of our client-side web application is the `main` function defined in the `client.go` source file in the `client` folder:

```
func main() {  
  
    var D = dom.GetWindow().Document().(dom.HTMLDocument)  
    switch readyState := D.ReadyState(); readyState {  
    case "loading":  
        D.AddEventListener("DOMContentLoaded", false, func(dom.Event) {  
            go run()  
        })  
    case "interactive", "complete":  
        run()  
    default:  
        println("Encountered unexpected document ready state value!")  
    }  
  
}
```

First, we assign the document object to the `D` variable, we're performing the usual aliasing operation here to save us some typing. We then declare a `switch` block on the document object's `readyState` property. We obtain the document object's `readyState` by calling the `ReadyState` method on the `Document` object.



The `readyState` property of a document describes the loading state of the document. You can read more about this property at the Mozilla Developer Network: <https://developer.mozilla.org/en-US/docs/Web/API/Document/readyState>.

In the first `case` statement, we will check whether the `readyState` value is "loading", and if it is, it indicates that the document is still loading. We set up an event listener to listen for the `DOMContentLoaded` event. The `DOMContentLoaded` event will be the cue that tells us that the web page has fully loaded, at which point, we can call the `run` function as a goroutine. We will call the `run` function as a goroutine since we don't want any operations inside the `run` function to block, because we are calling it from within the event handler function.

In the second `case` statement, we will check whether the `readyState` value is either `"interactive"` or `"complete"`. An `interactive` state indicates that the document has finished loading, but there may be some resources, such as images or style sheets, that haven't fully loaded. The `complete` state indicates that the document and all subresources have finished loading. If `readyState` is either `interactive` or `complete`, we will call the `run` function.

Finally, the `default` statement handles unexpected behavior. Ideally, we should never reach the `default` scenario, and if we ever do, we will print a message in the web console indicating that we have encountered an unexpected document `readyState` value.

The functionality that we have created in our `main` function provides us with the valuable benefit of being able to import our GopherJS produced JavaScript source file, `client.js`, from the `<head>` section of our HTML document as an external JavaScript source file, as shown here (shown in bold):

```
<head>
  <title>{{ .PageTitle }}</title>
  <link rel="icon" type="image/png"
    href="/static/images/isomorphic_go_icon.png">
  <link rel="stylesheet" href="/static/css/pure.min.css">
  <link rel="stylesheet" type="text/css" href="/static/css/cogimports.css">
  <link rel="stylesheet" type="text/css" href="/static/css/igweb.css">
  <script src="/static/js/cogimports.js" type="text/javascript"
    async></script>
  <script type="text/javascript" src="/js/client.js"></script>
</head>
```

This means that we don't have to import the external JavaScript source file right before the closing `</body>` tag to ensure that the web page has fully loaded. The procedure to include the external JavaScript source file, in the head declaration itself, is more robust, since our code specifically accounts for `readyState` in a responsible manner. The other, more brittle approach, is apathetic to `readyState`, and is dependent on the location of the included `<script>` tag within the HTML document to function properly.

Inside the `run` function, we will first print a message in the web console, indicating that we have successfully entered the client-side application:

```
println("IGWEB Client Application")
```

We will then fetch the template set from the server-side `/template-bundle` route that we set up earlier in this chapter:

```
templateSetChannel := make(chan *isokit.TemplateSet)
funcMap := template.FuncMap{"rubyformat": templatefuncs.RubyDate,
"unixformat": templatefuncs.UnixTime, "productionmode":
templatefuncs.IsProduction}
go isokit.FetchTemplateBundleWithSuppliedFunctionMap(templateSetChannel,
funcMap)
ts := <-templateSetChannel
```

We will create a channel called `templateSetChannel`, of the `*isokit.TemplateSet` type, in which we will receive the `TemplateSet` object. We will create a function map, containing the `rubyformat` and `unixformat` custom functions. We will then call the `FetchTemplateBundleWithSuppliedFunctionMap` function from the `isokit` package, supplying the `templateSetChannel` that we had just created, along with the `funcMap` variable.

The `FetchTemplateBundleWithSuppliedFunctionMap` function is responsible for fetching the template bundle items map from the server side, and assembling the template set using this map. In addition to that, the received `TemplateSet` object's `Funcs` property will be populated with the `funcMap` variable, ensuring that the custom functions will be accessible to all templates in our template set. Upon successfully calling this method, the template set will be sent over `templateSetChannel`. Finally, we will assign the `ts` variable with the `*isokit.TemplateSet` value that we receive from `templateSetChannel`.

We will create a new instance of the `Env` object, which we will use throughout the client-side application:

```
env := common.Env{}
```

We will then populate the `TemplateSet` property with the `Env` instance that we just created:

```
env.TemplateSet = ts
```

To save ourselves from having to type out `dom.GetWindow()` to access the `Window` object, and `dom.GetWindow().Document()` to access the `Document` object, we can populate the `Window` and `Document` properties of the `env` object with their respective values:

```
env.Window = dom.GetWindow()
env.Document = dom.GetWindow().Document()
```

We will be dynamically replacing the contents of the primary content `div` container as the user clicks on different sections of the website using the navigation bar. We will populate the `PrimaryContent` property of the `env` object to hold the primary content `div` container:

```
env.PrimaryContent = env.Document.GetElementByID("primaryContent")
```

This comes in handy, when we need to access this `div` container from within the route handler functions. It saves us from having to perform a DOM operation to retrieve this element each time we need it in the route handler.

We will call the `registerRoutes` function and supply it with the reference to the `env` object as the sole input argument to the function:

```
registerRoutes(&env)
```

This function is responsible for registering all the client-side routes and their associated handler functions.

We will call the `initializePage` function and supply it with the reference to the `env` object:

```
initializePage(&env)
```

This function is responsible for initializing interactive elements and components on the web page for a given client-side route.

There are two tasks that are of particular interest for us in the `registerRoutes` function:

1. Creating the client-side router
2. Registering the client-side route

## Creating the client-side router

First, we will create a new instance of the `isokit` router object and assign it to the `r` variable:

```
r := isokit.NewRouter()
```

## Registering the client-side route

The second line of code, registers the client-side `/about` route, along with its associated client-side handler function, `AboutHandler`, from the `handlers` package.

```
r.Handle("/about", handlers.AboutHandler(env))
```

We will be covering the rest of the `registerRoutes` function in more detail in [Chapter 5, End-to-End Routing](#).

## Initializing interactive elements on the web page

The `initializePage` function will get called only once when the web page is first loaded. Its role is to initialize the functionality that enables user interactivity with the client-side web application. This would be the respective `initialize` function for a given web page, which would be responsible for initializing event handlers and reusable components (cogs).

Inside the `initializePage` function, we will extract the `routeName` from the `PathName` property of the window's location object; the route name for the

`http://localhost:8080/about` URL will be "about":

```
l := strings.Split(env.Window.Location().Pathname, "/")
routeName := l[1]

if routeName == "" {
    routeName = "index"
}
```

If no `routeName` is available, we will assign the value of "index", the route name for the home page, to `routeName`.

We will declare a `switch` block on `routeName`, and here's the corresponding `case` statement that handles the scenario where `routeName` is equal to "about":

```
case "about":
    handlers.InitializeAboutPage(env)
```

The designated `initialize` function for the **About** page, is the `InitializeAboutPage` function, which is defined in the `handlers` package. This function is responsible for enabling user interactivity on the **About** page.

Now that we've set up the template set on the client-side, and registered the `/about` route, let's go ahead and take a look at the client-side **About** page handler function.

## Rendering the about page from the client side

Here's the definition of the client-side `AboutHandler` function in the `about.go` source file found in the `client/handlers` folder:

```
func AboutHandler(env *common.Env) isokit.Handler {
    return isokit.HandlerFunc(func(ctx context.Context) {
        gopherTeamChannel := make(chan []*models.Gopher)
        go FetchGopherTeam(gopherTeamChannel)
        gophers := <-gopherTeamChannel
        templateData := templatedata.About{PageTitle: "About", Gophers:
        gophers}
        env.TemplateSet.Render("about_content", &isokit.RenderParams{Data:
        templateData, Disposition: isokit.PlacementReplaceInnerContents, Element:
        env.PrimaryContent, PageTitle: templateData.PageTitle})
        InitializeAboutPage(env)
    })
}
```

We start out by creating a channel, `gopherTeamChannel`, which we will use to retrieve a slice of `Gopher` instances. We will call the `FetchGopherTeam` function as a goroutine, and supply it with `gopherTeamChannel` as the sole input argument to the function.

We will then receive the value returned from `gopherTeamChannel` and assign it to the `gophers` variable.

We will declare and initialize the `templateData` variable, the `about_content` template's data object, which has a type of `templatedata.About`. We will set the `PageTitle` property of the template data object and we populate the `Gophers` property with the `gophers` variable that we had just created.

We will call the `Render` method on the template set object to render the `about` template. The first argument we pass to the function is the name of the template, `about_content` that corresponds to the `about content` template. On the server side, we used the `about_page` template, since we also needed to generate the entire web page layout. Since we are operating from the client side, this is not necessary since we only need to populate the primary content area with the rendered content from the `about_content` template.

The second and final argument to the `Render` method is the render parameters of the `isokit.RenderParams` type. Let's examine each property that we set in the `RenderParams` object.

The `Data` property specifies the template data object that the template will be using.

The `Disposition` property specifies the disposition of the template content that will be rendered relative to an associated target element. The `isokit.PlacementReplaceInnerContents` disposition instructs the renderer to replace the inner contents of the associated target element.

The `Element` property specifies the associated target element that the renderer should take into consideration upon performing a template rendering. We will be placing the rendered content from the template in the primary content `div` container, and so we'll assign `env.PrimaryContent` to the `Element` property.

The `PageTitle` property specifies the web page title that should be used. The template data object's `PageTitle` property is equally important on the client side as it was on the server side, since the client-side renderer has the capability to change the web page's title.

Finally, we make a call to the `InitializeAboutPage` function to enable functionality that requires user interactivity. If the **About** page was the first page that was rendered on the website (from the server side), the `InitializeAboutPage` function would be called from the `initializePage` function found in the `client.go` source file. If we landed on the **About** page subsequently, by clicking on the **About** link on the navigation bar, then the request is serviced by the client-side `AboutHandler` function and we enable the functionality that requires user interactivity by calling the `InitializeAboutPage` function.

When it comes to user interactivity on the **About** page, we only have one reusable component that displays the time in a human readable format. We don't set up any event handlers, since we don't have any buttons or user input fields on this particular page. This being the case, we will skip the `InitializeAboutPage` function for now, and return to it in [Chapter 9, Cogs – Reusable Components](#). We'll show you an example of setting up event handlers in the designated `initialize` function for a given web page, in [Chapter 5, End-to-End Routing](#).

The `FetchGopherTeam` function is responsible for making an XHR call to the `/restapi/get-gopher-team` Rest API endpoint and retrieving the list of gophers that appear on the **About** page. Let's examine the `FetchGopherTeam` function:

```
func FetchGopherTeam(gopherTeamChannel chan []*models.Gopher) {
    data, err := xhr.Send("GET", "/restapi/get-gopher-team", nil)
    if err != nil {
        println("Encountered error: ", err)
    }
}
```

```
var gophers []*models.Gopher
json.NewDecoder(strings.NewReader(string(data))).Decode(&gophers)
gopherTeamChannel <- gophers
}
```

We make an XHR call by calling the `Send` function from the `xhr` package, and specifying that we will be using the `GET` HTTP method to make the call. We also specify that the call will be made to the `/restapi/get-gopher-team` endpoint. The last argument to the `Send` function is `nil`, since we will not be sending any data to the server from the client side.

If the XHR call is successfully made, the server will respond with JSON encoded data, representing a slice of gophers. We will create a new JSON decoder to decode the server's response into the `gophers` variable. Finally, we will send the `gophers` slice over `gopherTeamChannel`.

Now it's time to examine the Rest API endpoint that is responsible for servicing our XHR call to get the IGWEB team's gophers.

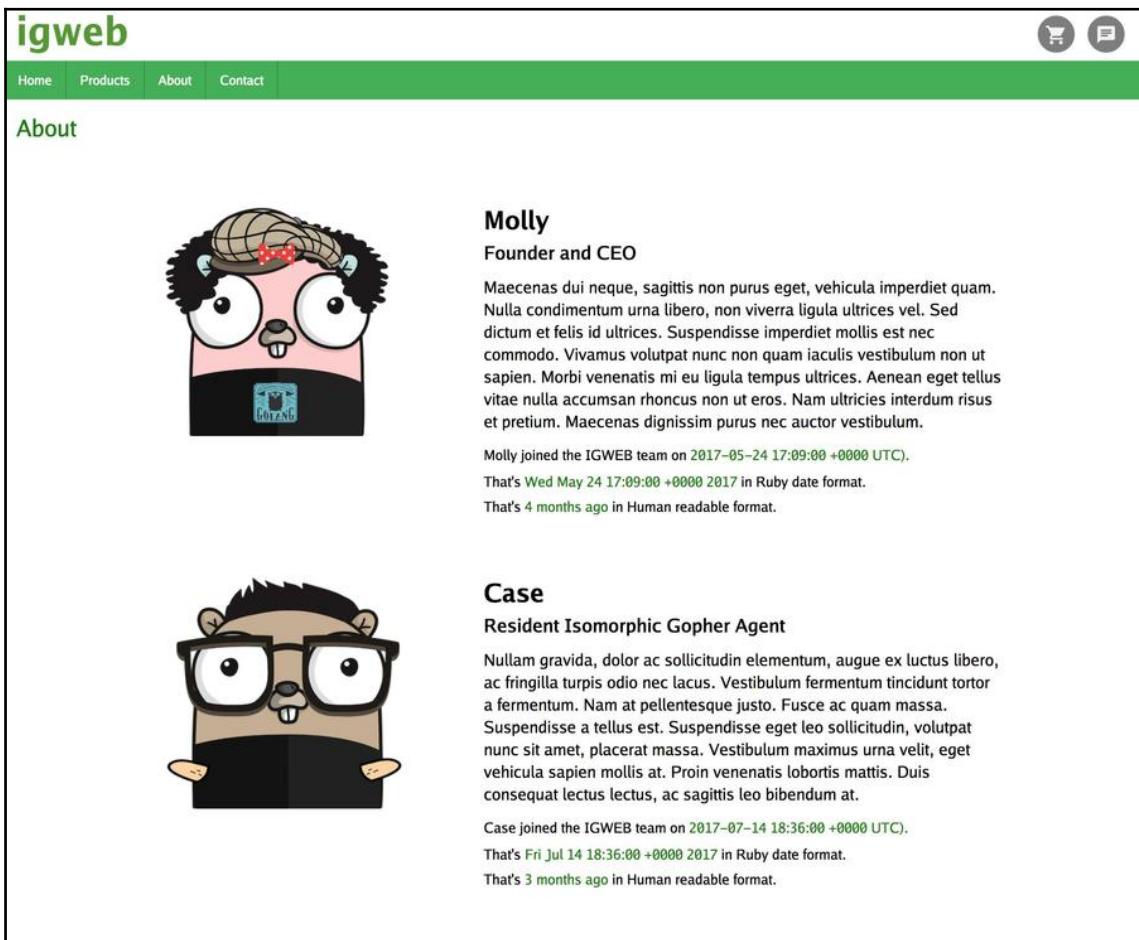
## The gopher team Rest API endpoint

The `/restapi/get-gopher-team` route is handled by the `GetGopherTeamEndpoint` function defined in the `gopherteam.go` source file found in the `endpoints` folder:

```
func GetGopherTeamEndpoint(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        gophers := env.DB.GetGopherTeam()
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(gophers)
    })
}
```

We will declare and initialize the `gophers` variable to the value returned from calling the `GetGopherTeam` method of the Redis datastore object, `env.DB`. We will then set a header to indicate that the server will be sending a JSON response. Finally, we will use a JSON encoder to encode the slice of gophers as JSON data. The data is sent to the client using `http.ResponseWriter, w`.

We have now set up everything we need to render the **About** page from the client side. We can view our client-side rendered **About** page by clicking on the **About** link on the navigation bar. Here's what the **About** page looks like rendered from the client side:



The screenshot shows the 'About' page of the 'igweb' application. The page has a green header with the 'igweb' logo, a navigation bar with 'Home', 'Products', 'About', and 'Contact' links, and a shopping cart and message icon. The main content area has a white background. It features two sections: 'Molly' and 'Case'. Each section contains a cartoon gopher agent illustration, the gopher's name in bold, their title, a short bio, and three lines of text at the bottom. The 'Molly' section includes a small 'Gophers' logo at the bottom of her illustration. The 'Case' section includes a small 'Gophers' logo at the bottom of his illustration.

**Molly**  
Founder and CEO

Maecenas dui neque, sagittis non purus eget, vehicula imperdiet quam. Nulla condimentum urna libero, non viverra ligula ultrices vel. Sed dictum et felis id ultrices. Suspendisse imperdiet mollis est nec commodo. Vivamus volutpat nunc non quam iaculis vestibulum non ut sapien. Morbi venenatis mi eu ligula tempus ultrices. Aenean eget tellus vitae nulla accumsan rhoncus non ut eros. Nam ultrices interdum risus et pretium. Maecenas dignissim purus nec auctor vestibulum.

Molly joined the IGWEB team on 2017-05-24 17:09:00 +0000 UTC.  
That's Wed May 24 17:09:00 +0000 2017 in Ruby date format.  
That's 4 months ago in Human readable format.

**Case**  
Resident Isomorphic Gopher Agent

Nullam gravida, dolor ac sollicitudin elementum, augue ex luctus libero, ac fringilla turpis odio nec lacus. Vestibulum fermentum tincidunt tortor a fermentum. Nam at pellentesque justo. Fusce ac quam massa. Suspendisse a tellus est. Suspendisse eget leo sollicitudin, volutpat nunc sit amet, placerat massa. Vestibulum maximus urna velit, eget vehicula sapien mollis at. Proin venenatis lobortis mattis. Duis consequat lectus lectus, ac sagittis leo bibendum at.

Case joined the IGWEB team on 2017-07-14 18:36:00 +0000 UTC.  
That's Fri Jul 14 18:36:00 +0000 2017 in Ruby date format.  
That's 3 months ago in Human readable format.

Figure 4.6 The About page rendered from the client-side

Can you make out any difference between the **About** page rendered on the server side and the one that was rendered on the client side? You shouldn't be able to see any differences since they are practically identical! We saved the user from having to witness a full page reload by simply rendering the **About** page content in the primary content area `div` container.

Take a look at the start times that are displayed for each gopher. The first time presented here, follows Go's default time formatting. The second time is the time using the Ruby date format. Recall that we use a custom function to present the time in this format. The third start time is displayed in human readable format. It uses a reusable component to format the time, which we will cover in Chapter 9, *Cogs – Reusable Components*.

We now know how to render templates isomorphically, and we will be following this same procedure for the other pages on IGWEB.

## Summary

In this chapter, we introduced you to the web template system and the individual components that comprise it—the template engine, the template data object, and the templates. We explored the purpose of each component of the web template system, and we designed the web page structure for IGWEB. We covered the three template categories: the layout template, the partial template, and the regular template. We then implemented each section of the IGWEB page structure as a template. We showed you how to define custom template functions that we could reuse across environments.

We then introduced you to the concept of isomorphic template rendering. We identified the limitations of standard template rendering, based on loading the template file from the file system, and introduced the in-memory template set, provided by the `isokit` package to render templates isomorphically. We then demonstrated how to set up the template set and render the **About** page on both the server side and on the client side.

In this chapter, we briefly touched upon routing, only so much to understand how to register the `/about` route, with its associated handler function on both the server side and the client side. In Chapter 5, *End-to-End Routing*, we will explore end-to-end application routing in further detail.

# 5

## End-to-End Routing

**End-to-end application routing** is the magic that allows us to leverage the benefits of the classic web application architecture along with the benefits of the single page application architecture. When implementing modern web applications, we have to strike a balance between satisfying the needs of two distinct audiences—humans and machines.

Let's first consider the experience from the human user's perspective. When a human user directly accesses the **About** page that we demonstrated in the previous chapter, template rendering is initially performed on the server side. This provides an initial page load that is perceived to be fast by the human user, since the web page content is readily available. This is the hallmark of the classic web application architecture. A different approach was taken for subsequent user interactions with the website. When the user clicked on the link to the **About** page from the navigation menu, template rendering was performed on the client side without the need of a full page reload, allowing for a more smooth and fluid user experience. This is the hallmark of the single page application architecture.

Machine users consist of the the various search engine bot crawlers that periodically visit the website. As you learned in the [Chapter 1, Isomorphic Web Applications with Go](#), single page applications, are primarily not search-engine friendly since the vast majority of search engine bots don't have the intelligence to traverse them. Traditional search engine bots are accustomed to parsing well-formed HTML markup that has already been rendered. It's much more difficult for the bots to be trained to parse through the JavaScript that is used to implement the single page application architecture. If we want greater search engine discoverability, we must satisfy the needs of our machine audience.

The goal of striking the balance between fulfilling the needs of these two distinct audiences is the essence of isomorphic web applications. We will learn how to accomplish this goal, in this chapter, while we implement the product-related pages of IGWEB.

In this chapter, we will cover the following topics:

- Routing perspectives
- The design of the product-related pages
- Implementing product-related templates
- Modeling product data
- Accessing product data
- Registering server-side routes with Gorilla Mux
- Server-side handler functions
- Registering client-side routes with the isokit router
- Client-side handler functions
- Rest API endpoints

## Routing perspectives

Let's consider how routing works in an Isomorphic Go web application from the perspective of the server side and from that of the client side. Remember, our goal is to utilize end to end routing to provide web page content access to machine users, and to deliver an enhanced user experience to human users.

## Routing on the server side

*Figure 5.1* depicts an initial page load in an Isomorphic Go application, implementing the classic web application architecture. The client can either be a web browser or a bot (machine) that accesses the website by providing a URL. The URL contains the route that the client is accessing. For example, the `/products` route, will serve the products listing page. The `/product-detail/swiss-army-knife` route, will serve the product detail page for the swiss army knife product that is being sold on the website. The request router is responsible for mapping the route to its designated route handler function. The request router we will be using on the server side is the Gorilla Mux router, and it is available in the `mux` package:

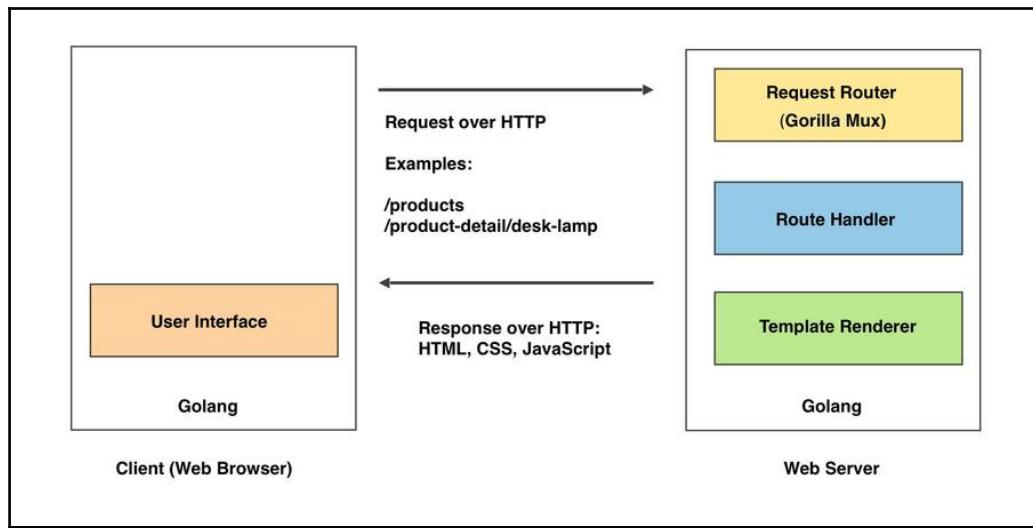


Figure 5.1: Initial page load in an Isomorphic Go application

The route handler is responsible for servicing a particular route. It contains a set of logic that is executed to perform a given task for a route. For example, the route handler for the `/products` route is responsible for fetching the products to display, rendering the products listing web page from the associated template, and sending the response back to the client. The response from the web server is an HTML document, which contains links to associated CSS and JavaScript source files. It may also be possible that the returned web page contains inline CSS or JavaScript sections as well.

Take note that although the diagram depicts Golang running inside the web browser, in reality, it is the JavaScript representation (which was transpiled using GopherJS) of the Go program that is running inside the web browser. When the client receives the server response, the web page is rendered in the user interface inside the web browser.

## Routing on the client side

*Figure 5.2* depicts routing from the perspective of the client side in an Isomorphic Go application, implementing the single page application architecture.

In *Figure 5.1*, the client side played a trivial role in simply rendering the web page server response. Now, in addition to displaying the rendered web page, the client contains a request router, route handlers, and the application business logic inside of it.

We will use the `isokit` router from the `isokit` package to perform client-side routing. The client-side router works much in the same manner as the server-side router, except instead of evaluating an HTTP request, the router intercepts the click of a hyperlink defined on the web page and routes it to a particular route handler defined on the client side itself. The client-side route handler that services a particular route, interacts with the server through a Rest API endpoint that is accessed by making XHR requests. The response from the web server is data that can take on a variety of formats, such as JSON, XML, Plain Text, and HTML Fragment, or even Gob-encoded data. We will use JSON as our means of data exchange in this chapter. The business logic of the application will determine how the data is handled, and it may be displayed to the user in the user interface. At this point, all rendering operations can take place on the client side, allowing us to prevent a full page reload:

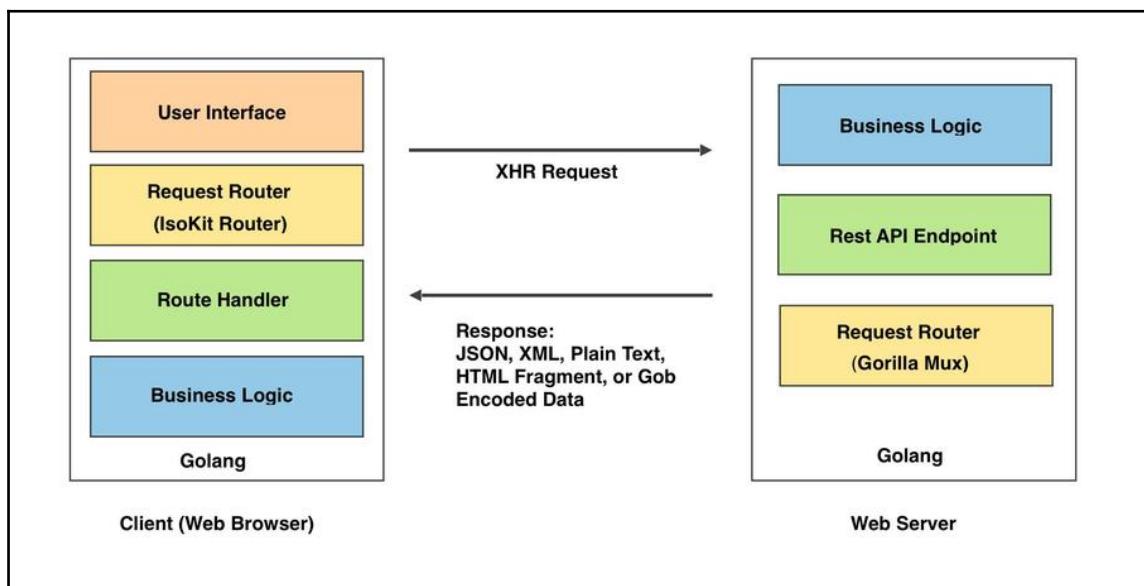


Figure 5.2: End to End Routing consists of a router on both ends

## Design of the product-related pages

The product-related pages of IGWEB consist of the products listing page and the product detail page. The **Products** page, which may also be referred to as the products listing page, will display a list of items that a user may purchase from the website. As depicted by the wireframe in *Figure 5.3*, each product will contain a thumbnail-sized image of the product, the product price, the product name, a brief description of the product, and a button to add the product to the shopping cart. Clicking on a product image will take the user to the product detail page for the given product. The route to access the products listing page is `/products`:

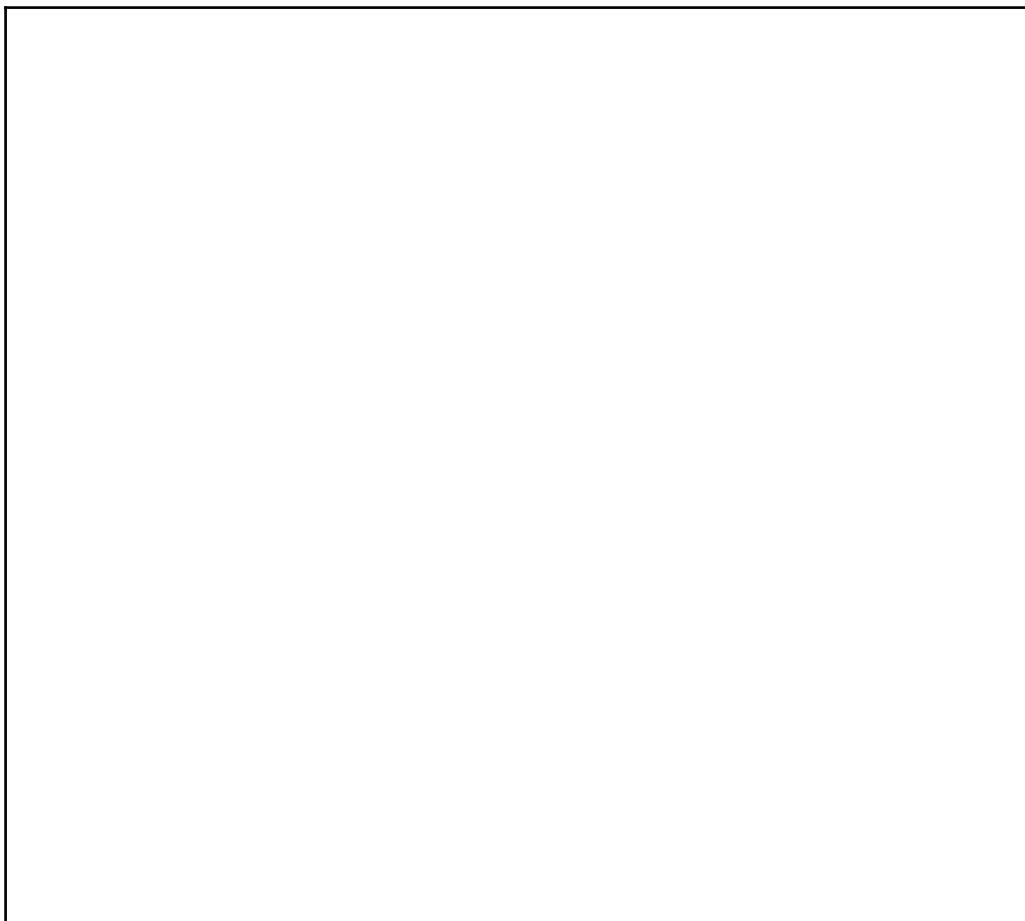


Figure 5.3: Wireframe design of the Products page

The product detail page contains information on an individual product. As depicted by the wireframe in *Figure 5.4*, the product detail page consists of a full size image of the product, the product name, the price of the product, a long description of the product, and a button to add the product to the shopping cart. The route to access the product detail page is `/product-detail/{productTitle}`. The `{productTitle}` is the **SEO** (short for, **search engine optimization**) friendly name of the product, for example, the swiss army knife product will have a `{productTitle}` value of "swiss-army-knife". By defining SEO friendly product names inside our `/product-detail` route, we make it easier for search engine bots to index a website and derive semantic meaning from the collection of product detail URLs. In fact, a URL that is search engine friendly is known as a **semantic URL**:

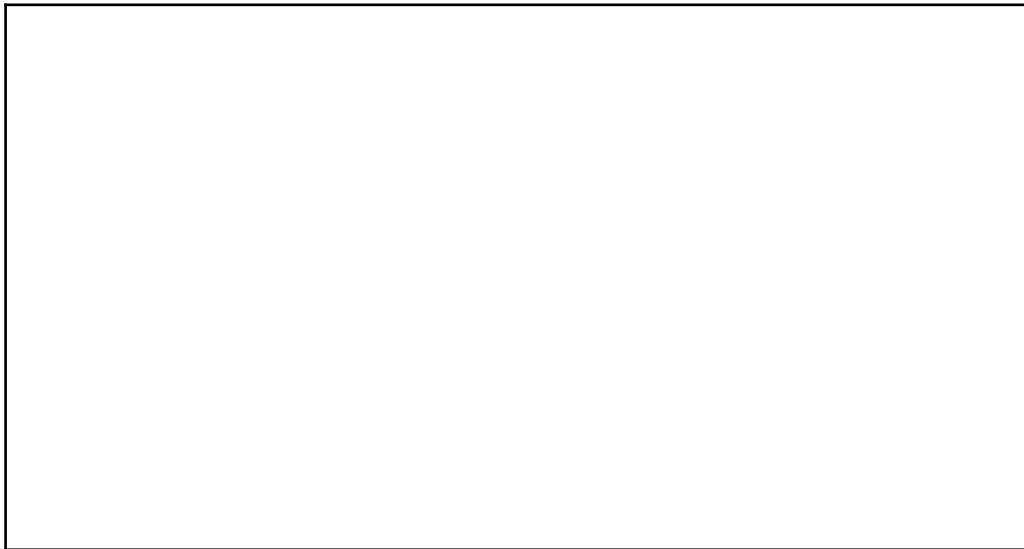


Figure 5.4: Wireframe design of the Product detail page

## Implementing product-related templates

Implementing the product-related templates consists of implementing the templates for the products listing page and the templates for the product detail page. The products listing page is depicted in *Figure 5.3*, and the product detail page is depicted in *Figure 5.4*. We will implement templates to realize these wireframe designs.

# Implementing the templates for the products listing page

Let's examine the `products_page.tmpl` source file found in the `shared/templates` directory:

```
 {{ define "pagecontent" }}
{{template "products_content" . }}
{{end}}
{{template "layouts/webpage_layout" . }}
```

This is the page template for the products listing page. The primary purpose of this template is to render the contents of the `products_content` template and place it inside of the web page layout.

Let's examine the `products_content.tmpl` source file found in the `shared/templates` directory:

```
<h1>Products</h1>

<div id="productsContainer">
{{if .Products}}
{{range .Products}}
<div class="productCard">
<a href="{{.Route}}">
<div class="pricebox"><span>${{.Price}}</span></div>
<div class="productCardImageContainer">

</div>
</a>
<div class="productContainer">

<h3><b>{{.Name}}</b></h3>
<p>{{.Description}}</p>

<div class="pure-controls">
<button class="addToCartButton pure-button pure-button-primary" data-
sku="{{.SKU}}>Add To Cart</button>
</div>

</div>
</div>
{{end}}
{{else}}
<span>If you're not seeing any products listed here, you probably need
```

```
to load the sample data set into your Redis instance. You can do so by <a  
target="_blank" href="/config/load-sample-data">clicking this  
link</a>.</span>  
{end}  
</div>
```

This is the content template for the products listing page. The purpose of this template is to display all the available products for sale. Inside the `productsContainer` `div` element, we have specified an `{{if}}` conditional that checks whether there are products available to be displayed. If there are products available, we use the `{{range}}` template action to iterate over all the available `Product` objects and generate the HTML markup required for each product card. We have defined an anchor (`<a>`) tag to make the image clickable, so that the user can navigate directly to the product detail page upon clicking on the product image. We have also defined a button to add the product to the shopping cart.

If there are no products to be displayed, we reach the `{{else}}` conditional and we place a helpful message to indicate that the products need to be loaded into the Redis database instance from the sample data set. For the reader's convenience, we have provided a hyperlink that can be clicked, which will populate the sample data into the Redis instance.

## Implementing the templates for the product detail page

Let's examine the `product_detail_page.tpl` source file found in the `shared/templates` directory:

```
{{ define "pagecontent" }}  
{{template "product_detail_content" . }}  
{end}  
{{template "layouts/webpage_layout" . }}
```

This is the page template for the product detail page. Its primary purpose is to render the contents of the `product_detail_content` template and place it inside of the web page layout.

Let's examine the `product_detail_content.tpl` source file found in the `shared/templates` directory:

```
<div class="productDetailContainer">  
  <div class="productDetailImageContainer">  
    
```

```
</div>

<div class="productDetailHeading">
  <h1>{{ .Product.Name }}</h1>
</div>

<div class="productDetailPrice">
  <span>${{ .Product.Price }}</span>
</div>

<div class="productSummaryDetail">
  {{ .Product.SummaryDetail }}
</div>

<div class="pure-controls">
  <button class="addToCartButton pure-button pure-button-primary" data-
sku="{{ .Product.SKU }}">Add To Cart</button>
</div>

</div>
```

Inside this template, we defined the HTML markup required to render the product detail container for the product detail page. We render the product image along with the product name, product price, and a detailed summary of the product. Finally, we declared a button to add the product to the shopping cart.

## Modeling product data

We have defined the `Product` struct in the `shared/models/product.go` source file to model product data:

```
package models

type Product struct {
  SKU string
  Name string
  Description string
  ThumbnailPreviewURI string
  ImagePreviewURI string
  Price float64
  Route string
  SummaryDetail string
  Quantity int
}
```

The `SKU` field represents the product's stock keeping unit (SKU), a unique id that represents the product. In the sample data set provided, we use incrementing integer values, however, this field is of the `string` type to accommodate alphanumeric SKUs in the future for extensibility. The `Name` field represents the name of the product. The `Description` field represents the short description that will be included in the products listing page. The `ThumbnailPreviewURI` field provides the path to the thumbnail image of the product. The `Price` field represents the price of the product and is of the `float64` type. The `Route` field is the server relative path to the product detail page for the given product. The `SummaryDetail` field represents the long description for the product that will be displayed in the product detail page. Finally, the `Quantity` field, which is of the `int` type, is the quantity of the particular product item that is presently in the shopping cart. We will be making use of this field in the next chapter, when we implement the shopping cart functionality.

## Accessing product data

For our product data access needs, we have defined two methods in our Redis datastore. The `GetProducts` method will return a slice of products and will drive the data needs for the products listing page. The `GetProductDetail` method will return the profile information for a given product that will drive the data needs of the product detail page.

## Retrieving products from the datastore

Let's examine the `GetProducts` method defined in the `common/datastore/redis.go` source file:

```
func (r *RedisDatastore) GetProducts() []*models.Product {

    registryKey := "product-registry"
    exists, err := r.Cmd("EXISTS", registryKey).Int()

    if err != nil {
        log.Println("Encountered error: ", err)
        return nil
    } else if exists == 0 {
        return nil
    }

    var productKeys []string
    jsonData, err := r.Cmd("GET", registryKey).Str()
```

```
if err != nil {
    log.Println("Encountered error when attempting to fetch product registry
data from Redis instance: ", err)
    return nil
}

if err := json.Unmarshal([]byte(jsonData), &productKeys); err != nil {
    log.Println("Encountered error when attempting to unmarshal JSON product
registry data: ", err)
    return nil
}

products := make([]*models.Product, 0)

for i := 0; i < len(productKeys); i++ {
    productTitle := strings.Replace(productKeys[i], "/product-detail/", "", -1)
    product := r.GetProductDetail(productTitle)
    products = append(products, product)
}

return products
}
```

Here, we start by checking whether the product registry key, "product-registry", exists in the Redis datastore. If it does exist, we declare a string slice called `productKeys`, which is a slice containing all the keys of the available products to display on the products listing page. We use the `Cmd` method on the Redis datastore object, `r`, to issue a Redis "GET" command, which is used to retrieve a record, for a given key. We supply the `registryKey` as the second argument to the method. Finally, we chain the method call to the `.Str()` method, which will convert the output to a string type.

## Retrieving the product detail from the datastore

The product registry data store in the Redis datastore is JSON data representing a slice of strings. We use the `Unmarshal` function found in the `json` package to unmarshal the JSON encoded data into the `productKeys` variable. Now, that we have all the product keys that should be displayed on the products listing page, it's time to create a product instance for each key. We do so, by first declaring the `products` variable that will be a slice of products. We iterate through the product keys and derive the `productTitle` value, which is the SEO friendly name of the product. We supply the `productTitle` variable to the `GetProductDetail` method of the Redis datastore to fetch a product for the given product title. We assign the fetched product to the `product` variable and append it to the `products` slice. Once the `for` loop ends, we will have collected all the products that should be displayed on the product listing page. Finally, we return the `products` slice.

Let's examine the `GetProductDetail` method defined in the `common/datastore/redis.go` source file:

```
func (r *RedisDatastore) GetProductDetail(productTitle string) *models.Product {
    productKey := "/product-detail/" + productTitle
    exists, err := r.Cmd("EXISTS", productKey).Int()

    if err != nil {
        log.Println("Encountered error: ", err)
        return nil
    } else if exists == 0 {
        return nil
    }

    var p models.Product
    jsonData, err := r.Cmd("GET", productKey).Str()

    if err != nil {
        log.Print("Encountered error when attempting to fetch product data from
Redis instance: ", err)
        return nil
    }

    if err := json.Unmarshal([]byte(jsonData), &p); err != nil {
        log.Print("Encountered error when attempting to unmarshal JSON product
data: ", err)
        return nil
    }
}
```

```
    return &p  
}
```

We assign the `productKey` variable of the `string` type with the value of the route to the product detail page. This involves concatenating the `"/product-detail"` string with the `productTitle` variable for the given product. We check to see whether the product key exists in the Redis datastore. If it doesn't exist, we return from the method, and if it does exist, we continue on and declare the `p` variable of the `Product` type. This will be the variable that the function will return. The product data stored in the Redis datastore is the JSON representation of a `Product` object. We `unmarshal` the JSON encoded data into the `p` variable. If we didn't encounter any errors, we return `p`, which represents the `Product` object for the requested `productTitle` variable that was specified as the input argument to the `GetProductDetail` method.

At this point, we have satisfied the data needs to display a list of products for the `/products` route and to display a product's profile page for the `/product-detail/{productTitle}` route. Now it's time to register the server-side routes for the product-related pages.

## Registering server-side routes with Gorilla Mux

We will use the Gorilla Mux router to handle the server-side application routing needs. This router is very flexible since it can not only handle simple routes such as `/products` but it can also handle routes with embedded variables. Recall that the `/product-detail` route contains the embedded `{productTitle}` variable.

We will start by creating a new instance of the Gorilla Mux router and assigning it to the `r` variable as follows:

```
r := mux.NewRouter()
```

Here's the section of code from the `registerRoutes` function, defined in the `igweb.go` source file, where we register routes along with their associated handler functions:

```
r.Handle("/", handlers.IndexHandler(env)).Methods("GET")
r.Handle("/index", handlers.IndexHandler(env)).Methods("GET")
r.Handle("/products", handlers.ProductsHandler(env)).Methods("GET")
r.Handle("/product-detail/{productTitle}",
    handlers.ProductDetailHandler(env)).Methods("GET")
r.Handle("/about", handlers.AboutHandler(env)).Methods("GET")
r.Handle("/contact", handlers.ContactHandler(env)).Methods("GET", "POST")
```

We use the `Handle` method to associate a route to a given handler function that is responsible for servicing the given route. For example, when the `/products` route is encountered, it will be handled by the `ProductsHandler` function defined in the `handlers` package. The `ProductsHandler` function will be responsible for fetching the products from the datastore, using the product records to render the products listing page from a template and sending the web page response back to the web client. Similarly, the `/product-detail/{productTitle}` route will be handled by the `ProductDetailHandler` function. This handler function will be responsible for fetching the product record for an individual product, using the product record to render the product detail page from a template and sending the web page response back to the web client.

## Server-side handler functions

Now that we have registered the server-side routes for the product-related pages, it's time to examine the server-side handler functions that are responsible for servicing these routes.

## The handler function for the products listing page

Let's examine the `products.go` source file found in the `handlers` directory:

```
package handlers

import (
    "net/http"

    "github.com/EngineerKamesh/igb/igweb/common"
    "github.com/EngineerKamesh/igb/igweb/shared/templatedata"
    "github.com/isomorphicgo/isokit"
)
```

```
func ProductsHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        products := env.DB.GetProducts()
        templateData := &templatedata.Products{PageTitle: "Products", Products: products}
        env.TemplateSet.Render("products_page", &isokit.RenderParams{Writer: w, Data: templateData})
    })
}
```

Here, we fetch the slice of products, featured on the **Products** page, by calling the `GetProducts` method on the Redis datastore object, `env.DB`. We declared the `templateData` variable of the `templatedata.Products` type, and it represents the data object that will be passed to the template engine, alongside the `products_page` template, to render the **Products** page. The `PageTitle` field represents the web page title, and the `Products` field is the slice of the products that are to be displayed on the **Products** page.

Inside the `ProductsHandler` function, we call the `GetProducts` method on the datastore object to fetch the available products for displaying from the datastore. We then create a template data instance having a `PageTitle` field value of "Products", and we assign the products that were fetched from the datastore to the `Products` field. Finally, we render the `products_page` template from the template set. With regard to the `RenderParams` object that we pass to the `env.TemplateSet` object's `Render` method, we set the `Writer` property to the `w` variable, which is `http.ResponseWriter`, and we set the `Data` property to the `templateData` variable, which is the data object that will be supplied to the template. At this point, the rendered web page will be sent back to the web client in the server response.

Figure 5.5 shows the rendered **Products** page that has been generated after accessing the `/products` route by visiting the following link: <http://localhost:8080/products>:

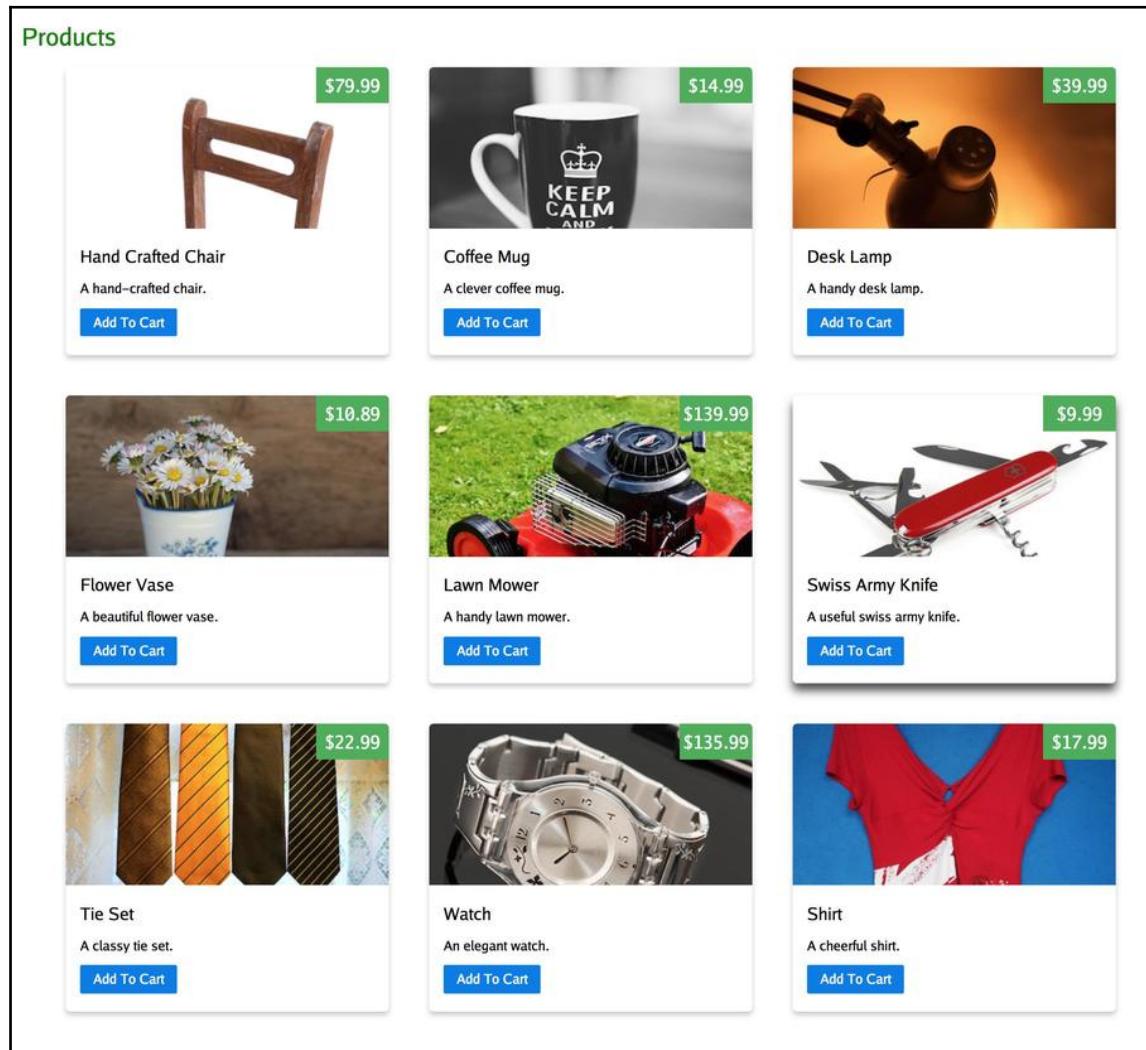


Figure 5.5: The Products Page

Now that we are able to display the **Products** page, let's take a look at the handler function for the product detail page.

## The handler function for the product detail page

Let's examine the `productdetail.go` source file found in the `handlers` directory:

```
package handlers

import (
    "net/http"

    "github.com/EngineerKamesh/igb/igweb/common"
    "github.com/EngineerKamesh/igb/igweb/shared/templatedata"
    "github.com/gorilla/mux"
    "github.com/isomorphicgo/isokit"
)

func ProductDetailHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        productTitle := vars["productTitle"]
        product := env.DB.GetProductDetail(productTitle)
        templateData := &templatedata.ProductDetail{PageTitle: product.Name,
Product: product}
        env.TemplateSet.Render("product_detail_page",
&isokit.RenderParams{Writer: w, Data: templateData})
    })
}
```

This is the handler function that handles the `/product/{productTitle}` route.

Remember, this is the route with the embedded variable inside of it. Inside the `ProductDetailHandler` function, we first gather the variables defined in the route, by calling the `Vars` function of the `mux` package. We supply `r`, the pointer to `http.Request`, as the input parameter to the `Vars` function. The result of this function is a map of the `map[string]string` type, where the key is the name of the variable in the route and the value is the value for that particular variable. For example, if we were accessing the `/product-detail/swiss-army-knife` route, the key would be `"productTitle"` and the value would be `"swiss-army-knife"`.

We get the value supplied for the `productTitle` variable in the route, and assign it to the `productTitle` variable. We then get the product object from the datastore by supplying the `GetProductDetail` method of the datastore object with the `productTitle` variable. We then set up our template data object, setting the fields for the page title and the product record. Finally, we call the `render` method on the template set indicating that we want to render the `product_detail_page` template. We assign the `http` response writer object and the template data object to the respective fields of the `render` `params` object, which is passed in as the second argument to the template set's `Render` method.

At this point, we have everything that we need in place to render the product detail page. Let's visit the product detail page for the swiss army knife from `http://localhost:8080/products/swiss-army-knife`. Here is the rendered product detail page in the web browser:

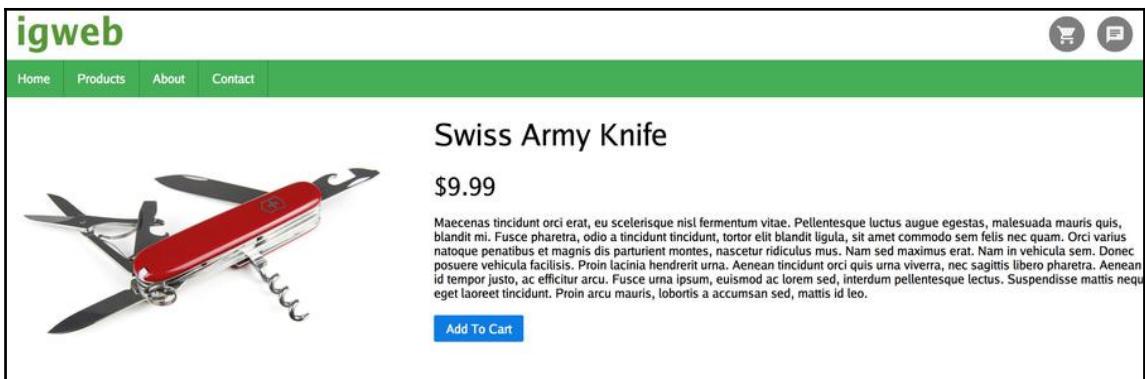


Figure 5.6: The Product detail page for the Swiss Army Knife

Now we have made the `/products` and `/product-title/{productTitle}` routes available to humans and machines alike, and we have implemented the classic web application architecture. Our machine users (the search engine bots) will be satisfied, since they can easily index the links of all the products available in the products listing page, and they can readily parse the HTML markup on each product detail page.

However, we haven't completely satisfied our human audience yet. You may have noticed that clicking on an individual product from the product listing page results in a full page reload. In a brief instance, the screen may go white in the transition of leaving one page and rendering the next page within the web browser. The same full page reload happens when we click on the **Products** link in the navigation menu to return to the products listing page from the product detail page. We can enhance the user experience of transitioning through web pages by implementing the single page architecture for subsequent interactions with the website, after the initial page load. In order to do so, we need to define client-side routes along with their associated client-side route handler functions.

## Registering client-side routes with the isokit router

On the client side, we use the isokit router to handle routes. The isokit router works by intercepting the click events to hyperlinks and checking if the hyperlink contains a route defined in its routing table.

We can register a route in the router's routing table using the `Handle` method of the isokit router object. The `Handle` method takes two parameters—the first parameter is the route and the second parameter is the handler function that should service the route. Notice that the code to declare and register routes is very similar to the Gorilla Mux router on the server side. Because of this similarity, registering routes on the client side, with the isokit router, is straightforward and feels like second nature.

Here's the section of code from the `registerRoutes` function defined in the `client.go` source file found in the `client` folder, which is responsible for registering routes:

```
r := isokit.NewRouter()
r.Handle("/index", handlers.IndexHandler(env))
r.Handle("/products", handlers.ProductsHandler(env))
r.Handle("/product-detail/{productTitle}",
  handlers.ProductDetailHandler(env))
r.Handle("/about", handlers.AboutHandler(env))
r.Handle("/contact", handlers.ContactHandler(env))
r.Listen()
env.Router = r
```

Here, we start by creating a new isokit router by calling the `NewRouter` function from the `isokit` package and assigning it to the `r` variable. We have defined the `/products` route for the product listing page, and the `/product-data/{productTitle}` route for the product detail page. After defining all the routes, we make a call to the `Listen` method of the router object, `r`. The `Listen` method is responsible for adding an event listener to all hyperlinks, to listen for click events. Links that are defined in the router's routing table will be intercepted upon a click event and their associated client-side route handler function will service them. Finally, we assign the `r` router to the `Router` field of the `env` object, so that we can access the router throughout the client-side web application.

## Client-side handler functions

Now that we've registered the routes for the product-related pages on the client side, let's take a look at the client-side route handlers functions that are responsible for servicing these routes.

## The handler function for the products listing page

Let's examine the `ProductsHandler` function in the `products.go` source file found in the `client/handlers` directory:

```
func ProductsHandler(env *common.Env) isokit.Handler {
    return isokit.HandlerFunc(func(ctx context.Context) {

        productsChannel := make(chan []*models.Product)
        go FetchProducts(productsChannel)
        products := <-productsChannel
        templateData := &templatedata.Products{PageTitle: "Products", Products:
            products}
        env.TemplateSet.Render("products_content", &isokit.RenderParams{Data:
            templateData, Disposition: isokit.PlacementReplaceInnerContents, Element:
            env.PrimaryContent, PageTitle: templateData.PageTitle})
        InitializeProductsPage(env)
        env.Router.RegisterLinks("#primaryContent a")
    })
}
```

Recall that from the diagram depicted in *Figure 5.2*, that the client-side web application accesses the server-side functionality through the use of XHR calls to Rest API endpoints. Here, we create the `productsChannel` channel to retrieve a slice of the `Product` objects. We call the `FetchProducts` function, which will make the XHR call to the Rest API endpoint on the server that is responsible for retrieving the list of available products to display on the **Products** page. Notice that we call the `FetchProducts` function as a goroutine. We must do this to ensure that the XHR call does not block. We supply the `productsChannel` channel as the sole input argument to the `FetchProducts` function. We then retrieve the list of products over the `productsChannel` channel and assign it to the `products` variable.

We create a new template data object instance, `templateData`, and set the respective fields for the `PageTitle` and the `Products` field. After this, we call the `Render` method on the `env.TemplateSet` object, specifying that we want to render the `products_content` template. In the `RenderParams` object that we supply to the `Render` function, we set the `Data` field with the template data object, `templateData`. We set the `Disposition` field to `isokit.PlacementReplaceInnerContents` to specify that the disposition of the render should replace the inner HTML contents of the associated element. We set the `Element` field to be the primary content `div` container, where the main page content is rendered. We call the `InitializeProductsEventHandlers` function to set up the event handlers found in the `products` page. For the **Products** page, the only DOM element needing an event handler is the **Add To Cart** button, which we will cover in *Chapter 6, Isomorphic Handoff*.

As far as client-side routing is concerned, the last line of code in the `ProductsHandler` function is the most important line. When each product card is rendered by the template renderer, we need to intercept the links of each product item. We can tell the `isokit` router to intercept these links by providing a query selector that will target the links found in the primary content `div` container. We do this by calling the `RegisterLinks` method of the `isokit` router object and specifying that the query selector should be `"#primaryContent a"`. This will ensure that all the links for the product items are intercepted, and that when we click on a product item, instead of performing a full page reload to get to the `/product-detail/{productTitle}` route, the client-side route handler for the `product-detail` route will kick in and service the request.

## Fetching the list of products

Now that we've seen how the client-side route handler function works, let's take a look at the `FetchProducts` function that is used to make the XHR call to the server and gather the list of products to display on the page:

```
func FetchProducts(productsChannel chan []*models.Product) {  
  
    data, err := xhr.Send("GET", "/restapi/get-products", nil)  
    if err != nil {  
        println("Encountered error: ", err)  
        return  
    }  
    var products []*models.Product  
    json.NewDecoder(strings.NewReader(string(data))).Decode(&products)  
  
    productsChannel <- products  
}
```

Here, we use the `xhr` package to make XHR calls to the server. We call the `Send` function from the `xhr` package and specify that our request will be using the `GET` method, and we will be making the request to the `/restapi/get-products` endpoint. For the third argument to the function, we pass a value of `nil` to indicate that we are not sending data in our XHR call. If the XHR call is successful, we will receive JSON data from the server, which will represent a slice of the `Product` objects. We create a new JSON Decoder to decode the data and store it in the `products` variable, which we send over the `productsChannel`. We will be examining the Rest API endpoint that services this XHR call, in the section, *The endpoint to get the list of products*.

At this point, our web application has attained the goal of being able to render the **Products** page without causing a full page reload on subsequent interactions with the website. For example, if we were to access the **About** page at `http://localhost:8080/about`, the initial page load will be serviced on the server side. If we initiated a subsequent interaction by clicking on the **Products** link in the navigation menu, the client-side routing will kick in, and the **Products** page will load, without a full page reload taking place.

In the *Verifying client-side routing functionality* section, we will show you how you can verify that client-side routing is functioning properly, using the web browser's inspector. Now it's time to implement the client-side route handler for the product detail page.

## The handler function for the product detail page

Let's examine the `ProductDetailHandler` function defined in the `productdetail.go` source file found in the `client/handlers` directory:

```
func ProductDetailHandler(env *common.Env) isokit.Handler {
    return isokit.HandlerFunc(func(ctx context.Context) {
        routeVars := ctx.Value(isokit.RouteVarsKey("Vars")).(map[string]string)
        productTitle := routeVars[`product-detail/{productTitle}`]
        productChannel := make(chan *models.Product)
        go FetchProductDetail(productChannel, productTitle)
        product := <-productChannel
        templateData := &templatedata.ProductDetail{PageTitle: product.Name,
        Product: product}
        env.TemplateSet.Render("product_detail_content",
        &isokit.RenderParams{Data: templateData, Disposition:
        isokit.PlacementReplaceInnerContents, Element: env.PrimaryContent,
        PageTitle: templateData.PageTitle})
        InitializeProductDetailPage(env)
    })
}
```

The `ProductDetailHandler` function returns an `isokit.Handler` value. Notice that we specify `isokit.HandlerFunc` as a closure, so that we can perform a dependency injection of the `env` object to our client-side handler function. Notice that the input argument to `isokit.HandlerFunc` is of the `context.Context` type. This `context` object is important because it contains the variable information embedded within a route. By calling the `Value` method on the `ctx` `context` object, we can obtain the route variables by specifying the key of "Vars" to the `context` object. Notice that we perform a type assertion to specify that the value obtained from the `context` object is of the `map[string]string` type. We can extract the value for the `productTitle` from the route by providing the ``product-detail/{productTitle}`` key. The value for the `productTitle` is important since we will be sending it as a route variable in the XHR call we make to the server to obtain a `product` object.

We create a `product` channel, `productChannel`, that will be used to send and receive a `Product` object. We call the `FetchProductDetail` function, providing `productChannel` and `productTitle` as input arguments to the function. Notice that we call the function as a goroutine, and upon successfully running the function, we will have a `product` object sent over `productChannel`.

We set up our template data object, specifying values for the `PageTitle` and `Product` fields. We then set the title of the page to the product name. Once this is done, we call the `Render` method of the template set object, and we specify that we want to render the `product_detail_content` template. We set the fields of the `render` parameters object, populating the fields for the template data object, the disposition, and the associated element where the template will be rendered to, which is the primary content `<div>` container. Finally, we make a call to the `InitializeProductDetailEventHandlers` function, which is responsible for setting up the event handlers for the product detail page. The only element in need of handlers for this page is the **Add To Cart** button, which we will cover in the next chapter.

## Fetching the product detail

Let's examine the `FetchProductDetail` function defined in the `productdetail.go` source file in the `client/handlers` folder:

```
func FetchProductDetail(productChannel chan *models.Product, pageTitle string) {  
  
    data, err := xhr.Send("GET", "/restapi/get-product-  
    detail"+"/"+pageTitle, nil)  
    if err != nil {  
        println("Encountered error: ", err)  
        println(err)  
    }  
    var product *models.Product  
    json.NewDecoder(strings.NewReader(string(data))).Decode(&product)  
  
    productChannel <- product  
}
```

This function is responsible for making the XHR call to the Rest API endpoint on the server that provides the product data. The function takes in a product channel and a product title as input arguments. We make the XHR call by calling the `Send` function of the `xhr` package. Take note that in the second input argument to the function (the destination that we are making the request to), we concatenate the `pageTitle` variable to the `/restapi/get-product-detail` route. So, for example, if we wanted to request the product object for the swiss army knife, we would specify a route of `/restapi/get-product-detail/swiss-army-knife`, and in this scenario, the `pageTitle` variable would be equal to `"swiss-army-knife"`.

If the XHR call was successful, the server will return the JSON encoded product object. We use a JSON decoder to decode the JSON data returned from the server, and we set the `product` variable to the decoded `Product` object. Finally, we pass `product` over `productChannel`.

## Rest API endpoints

The Rest API endpoints on the server side are pretty handy. They are the means to provide the client-side web application with data, behind the scenes, and we apply this data to the respective template to display the page content without having the need to do a full page reload.

Now, we'll consider what goes in to creating these Rest API endpoints. We will first have to register routes for them on the server side. We will follow the same procedure that we did at the beginning of this chapter for the product listing page and the product detail page. The only difference is that our handler functions will be found in the `endpoints` package as opposed to being in the `handlers` package. The fundamental difference here is that the `handlers` package contains handler functions that return a full web page response back to the web client. The `endpoints` package, on the other hand, contains handler functions that will return data, most likely in the JSON format, back to the web client.

Here's the section of code from the `igweb.go` source file, where we register our Rest API endpoints:

```
r.Handle("/restapi/get-products",
  endpoints.GetProductsEndpoint(env)).Methods("GET")
r.Handle("/restapi/get-product-detail/{productTitle}",
  endpoints.GetProductDetailEndpoint(env)).Methods("GET")
```

Notice that the `/restapi/get-products` route that drives the data needs for the client-side products page is serviced by the `GetProductsEndpoint` function found in the `endpoints` package.

Similarly, the `/restapi/get-product-detail/{productTitle}` route that drives the data needs for the client-side product detail page, is serviced by the `GetProductDetailEndpoint` function found in the `endpoints` package.

## The endpoint to get the list of products

Let's examine the `products.go` source file in the `endpoints` folder:

```
package endpoints

import (
    "encoding/json"
    "net/http"

    "github.com/EngineerKamesh/igb/igweb/common"
)

func GetProductsEndpoint(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        products := env.DB.GetProducts()
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(products)
    })
}
```

Inside the `GetProductsEndpoint` function, we first fetch the slice of products that will be displayed on the client-side products page by calling the `GetProducts` method of the `datastore` object, `env.DB`. We then set a header to indicate that the server response will be in JSON format. Finally, we use a JSON encoder to encode the slice of products as JSON data and write it out using the the `http.ResponseWriter`, `w`.

## The endpoint to get the product detail

Let's examine the `productdetail.go` source file in the `endpoints` folder:

```
package endpoints

import (
    "encoding/json"
    "net/http"

    "github.com/EngineerKamesh/igb/igweb/common"
    "github.com/gorilla/mux"
)

func GetProductDetailEndpoint(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
```

```
    productTitle := vars["productTitle"]
    products := env.DB.GetProductDetail(productTitle)
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(products)
  })
}
```

Inside the `GetProductDetailEndpoint` function, we fetch embedded route variables, by calling the `Vars` function from the `mux` package and supplying the router object, `r`, as the sole input argument to the function. We then obtain the value for the `{productTitle}` embedded route variable and assign it to the variable `productTitle`. We supply the `productTitle` to the `GetProductDetail` method of the datastore object, `env.DB`, to retrieve the corresponding `Product` object from the datastore. We set a header to indicate that the server response will be in the JSON format, and we use the JSON encoder to encode the `Product` object as JSON data, which will be sent out to the web client using `http.ResponseWriter`, `w`.

We have now reached a major milestone. We have implemented the product-related pages in a manner that is friendly to both humans and machines alike. When a user initially access the products listing page, by entering the URL (`http://localhost:8080/products`) in the web browser, the page is rendered on the server side, and the web page response is sent back to the client. The user is able to see the web page instantly since the web page response is pre-rendered. This behavior exhibits the desired trait from the classic web application architecture.

When the human user initiates subsequent interactions, by clicking on a product item, the product detail page is rendered from the client side, and the user is saved from having to experience a full page reload. This behavior exhibits the desired trait from the SPA architecture.

The machine users (the search engine bot crawlers) are also satisfied since they can traverse each link to a product item on the products page and readily index the website, since we have semantic URLs in place along with well-formed HTML markup that the search engine bots can understand.

## Verifying the client-side routing functionality

To ensure that the client-side routing is functioning properly, you can exercise the following procedure:

1. Access the **Products** page in your web browser and open up the web browser's inspector.
2. Click on the network tab to watch network traffic, and make sure to filter on XHR calls. Now, click on a product item to get to the product's detail page.
3. Return to the **Products** page by clicking the **Products** link on the navigation menu.

Repeat this process several times, and you should be able to see all the XHR calls being made in the background. *Figure 5.7* includes a screenshot of this procedure to verify that the client-side routing is functioning properly:

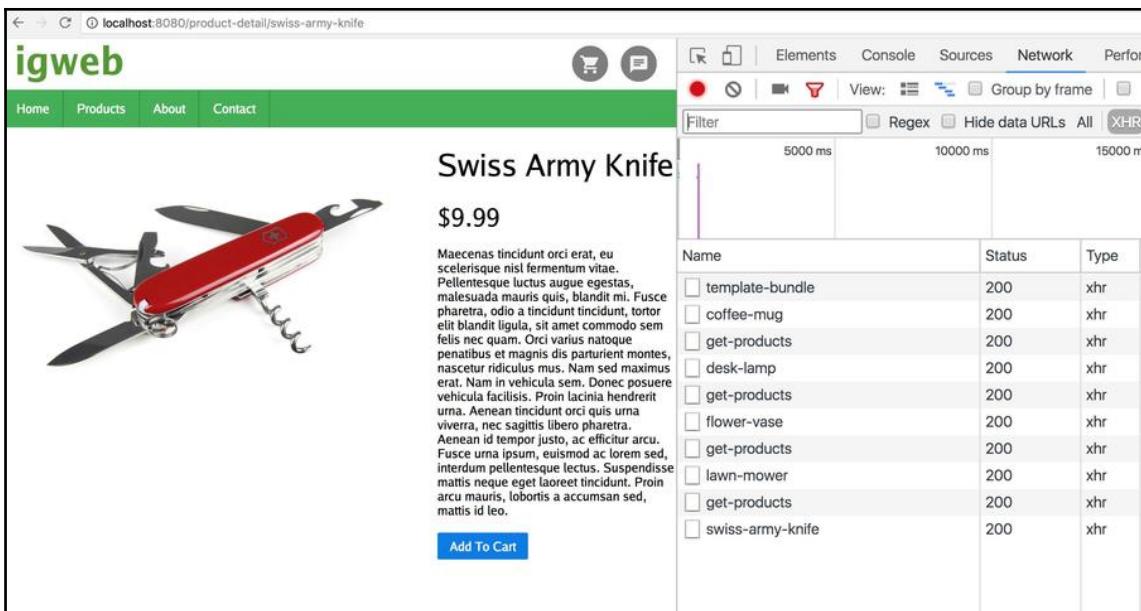


Figure 5.7: The XHR calls in the web console confirms that the client-side routing is functioning properly

## Summary

In this chapter, we implemented end to end application routing for IGWEB as we built the products-related pages. We started by registering server-side routes using the Gorilla Mux router. We associated each route with a corresponding, server-side route handler function that would service the server-side route. We then examined the implementation of the server-side route handler functions for the product-related pages.

Having satisfied the needs of implementing the classic web application architecture for the initial page load, we moved on to the client side by first registering routes for the product-related pages, using the isokit router. Just as we had done on the server side, we associated each client-side route with a corresponding client side route handler function that would service the client-side route. You learned how to implement client-side route handlers and how to make XHR calls from them to server-side Rest API endpoints. Finally, you learned how to create server-side Rest API endpoints that handled XHR requests and returned JSON data to the client.

The product-related pages had a persistent state since the list of available products were driven by the contents of the datastore. How do we maintain state in a situation where the user interaction with the website changes the given state? For example, if the user added items to the shopping cart, how can we go about maintaining the state of the shopping cart and syncing it up between the server side and the client side? You are going to learn about *isomorphic handoff*, the process of handing off state between the server and the client in Chapter 6, *Isomorphic Handoff*. We will implement the shopping cart functionality for the website in the process of doing this.

# 6

## Isomorphic Handoff

In the development of Isomorphic Go web applications, two critical techniques were introduced in the previous two chapters. First, you learned how to utilize an in-memory template set to render templates across environments. Second, you learned how to perform end-to-end routing on both the client and the server. Client-side routing is the magic that allows the client-side web application to operate in the single page mode.

The aforementioned techniques now provide us with the ability to navigate to different sections of the website, on the client itself, and render any given template across environments. As the implementors of an Isomorphic Go web application our responsibility is to ensure that state is maintained between the client and the server. For example, when rendering the **Products** page, it wouldn't make sense if the list of products was rendered differently on the client side than it was on the server side. The client needs to work in lockstep with the server to ensure that the state, in this case, the list of products is maintained—and that's where *isomorphic handoff* comes into the picture.

**Isomorphic handoff** is the process by which the server hands off state to the client and the client uses the passed state to render the web page on the client side. Keep in mind that the state the server passes off to the client must include the exact same state that was used to render the server-side web page response. Isomorphic handoff essentially allows the client to seamlessly pick up things where they were left off on the server. In this chapter, we'll revisit the product-related pages, to see exactly how the state is maintained from the server side to the client side. In addition to that, we will also complete the implementation of the product-related pages, by implementing the user interactivity portions, which involve adding event handlers to the **Add To Cart** buttons found in these pages.

The shopping cart feature for the IGWEB website will be implemented in this chapter, and it will allow us to consider the scenario where the user can change the state of the shopping cart by adding and removing items to and from the shopping cart. We will use isomorphic handoff to ensure that the current state of the shopping cart is seamlessly maintained across the server and the client. By properly maintaining the state of the shopping cart we can guarantee that the shopping cart page rendered from the server side always matches the shopping cart page rendered from the client side.

In this chapter, we will cover the following topics:

- The isomorphic handoff procedure
- Implementing isomorphic handoff for the product-related pages
- Implementing isomorphic handoff for the shopping cart

## The isomorphic handoff procedure

A recurring theme in the development of isomorphic web applications centers around the ability to share between the server and the client. In an isomorphic web application, the server and client must work in unison to seamlessly maintain the state of a particular workflow in the application. In order to do so, the server must share the current state, which was used to render the web page output on the server side with the client.

## The ERDA strategy

The isomorphic handoff procedure consists of the following four steps:

1. Encode
2. Register
3. Decode
4. Attach

We can use the acronym **ERDA** (**E**ncode-**R**egister-**D**ecode-**A**ttach) to easily recall each individual step of the procedure. In fact, we can collectively refer to the steps to implement the isomorphic handoff procedure as the **ERDA strategy**.

By implementing the four steps of the isomorphic handoff procedure, as depicted in *Figure 6.1*, we can guarantee that state is successfully persisted between the server and the client:

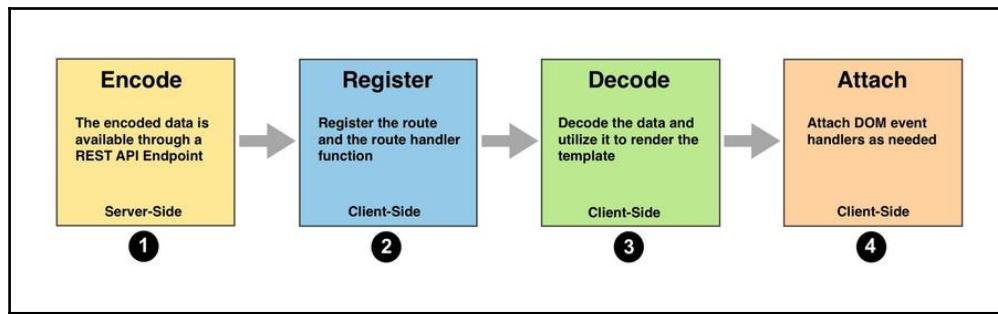


Figure 6.1: The ERDA strategy to implement isomorphic handoff

The first step, **Encode**, involves encoding a data object that represents the state we wish to retain to the client into a data exchange format (JSON, Gob, XML, and so on). The subsequent steps are all performed on the client side. The second step, **Register**, involves registering a client-side route and its respective handler function. The third step, **Decode**, involves decoding the encoded data retrieved from the server, through a Rest API endpoint, and utilizing it to render the template for the web page on the client side. The fourth and last step, **Attach**, involves attaching any needed event handlers to the rendered web page to enable user interactivity.

Figure 6.2 depicts the key modules involved, on both the server and the client, in implementing the isomorphic handoff procedure:

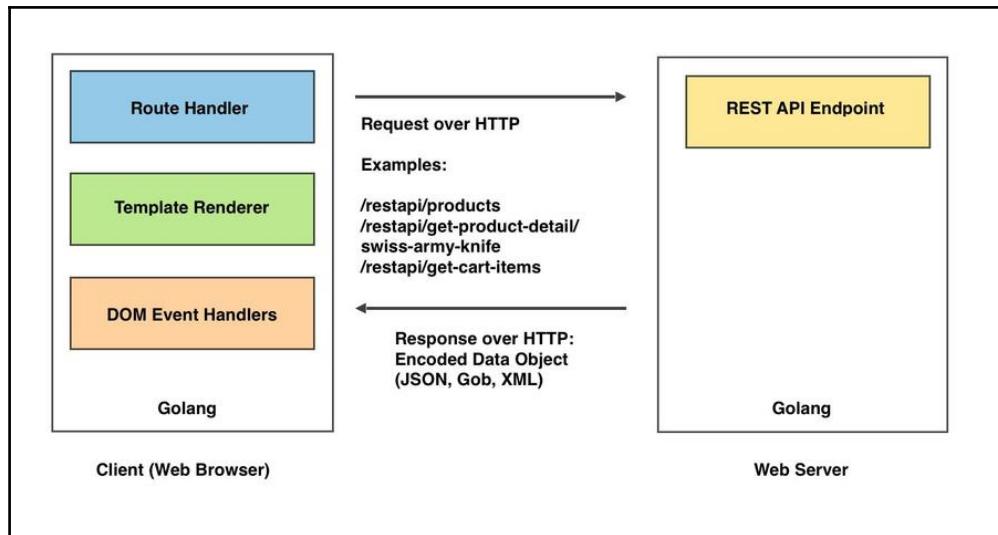


Figure 6.2: The key modules involved in isomorphic handoff

The **Encode** step, is performed inside the **Rest API Endpoint** that exists in the server-side web application. The **Register** step is performed inside the **Route Handler** that exists in the client-side web application. The **Decode** step is performed prior to calling the client-side **Template Renderer**. The **Attach** Step is performed by implementing the **DOM Event Handlers** on the client side.

Now that we have introduced each step in the ERDA strategy, let's explore each step in detail.

## The Encode step

Our goal to regenerate the state on the client side begins by identifying the data object that represents the state we wish to retain in order to maintain state in a particular web page. To identify the object, we simply need to take a look at the server-side handler function that produced the rendered web page output. For example, in the products listing page, the slice of `Product` objects would be the data object that we would want to retain to the client side, so that the web page rendered on the client-side would render the same list of products.

We can expose the slice of the `Product` objects to the client side by implementing a Rest API Endpoint (depicted in *Figure 6.2*). The **Encode** step (depicted in *Figure 6.1*), consists of encoding the slice of the `Product` objects to a common data exchange format. For this chapter, we will encode objects using the **JSON** (short for, **JavaScript Object Notation**) format. The client-side web application can access the encoded object by making an XHR call to the Rest API endpoint.

Now that the encoded state object is available, the rest of the steps to implement the isomorphic handoff procedure happen on the client side.

## The Register step

To fulfill the register step (depicted in *Figure 6.1*), we must first register a client-side route and its respective handler function (depicted in the Route Handler box in *Figure 6.2*). For example, for the **Products** page, we would register the `/products` route and its associated handler function, `ProductsHandler`. When a user clicks on the **Products** link from the navigation bar, the click event will be intercepted by the isokit router and the handler function, `ProductsHandler`, which is associated with handling the `/products` route, will be called. The route handler function plays the role of executing the last two steps of the isomorphic handoff process—decode and attach.

Keep in mind that if the user were to land first on the `/products` route by directly accessing the web page by entering the URL in the web browser, the server-side handler function will kick-in and the **Products** page will be rendered on the server side. This provides us the capability to render the web page instantly, providing a page load that is perceived to be fast to the user.

## The Decode step

Within the route handler function, we initiate an XHR call to the Rest API endpoint, that will return the encoded data that represents the state we wish to maintain on the client side. Once the encoded data is obtained, we will perform the third step, **Decode** (depicted in *Figure 6.1*), of the isomorphic handoff procedure. In this step, we decode the encoded data back into an object instance. The object instance is then utilized to populate the corresponding field of the template data object that is passed to the Template Renderer (depicted in *Figure 6.2*), so that the web page can be successfully rendered on the client side, in the same manner as it was rendered on the server side.

## The Attach step

The fourth and last step, **Attach** (depicted in *Figure 6.1*), is responsible for attaching event handlers (depicted in *Figure 6.2*) to DOM elements that exist in the rendered web page. For example, in the **Products** page, we would need to attach event handlers to all the **Add To Cart** buttons found on the web page. When an **Add To Cart** button is pressed, the respective product item will be added to the user's shopping cart.

Up to this point, we have laid out the groundwork needed to implement the isomorphic handoff procedure for a given web page. To solidify our understanding of isomorphic handoff, let's consider two specific examples where we implement all four steps of the procedure. First, we will implement the isomorphic handoff procedure in the product-related pages, which include the products listing page (`/products`) and the product detail page (`/product-detail/{productTitle}`). Second, we will implement the isomorphic handoff procedure for the shopping cart page. The second example will be more dynamic, since the user will have the capability to alter the state, because the user can add and remove items to the shopping cart as they wish. This capability allows the user to exert control on the current state of the shopping cart.

# Implementing isomorphic handoff for the product-related pages

As noted earlier, the product-related pages consist of the products listing page and the product detail page. We will follow the ERDA strategy to implement the isomorphic handoff procedure for these pages.

## Implementing the sort interface for the product model

Before we get started, we will define a new type called `Products` (in the `shared/models/product.go` source file), which will be a slice of `Product` objects:

```
type Products []*Product
```

We will have the `Products` type implement the `sort` interface by defining the following methods:

```
func (p Products) Len() int { return len(p) }
func (p Products) Less(i, j int) bool { return p[i].Price < p[j].Price }
func (p Products) Swap(i, j int) { p[i], p[j] = p[j], p[i] }
```

By examining the `Less` method, you will be able to see that we will sort the products displayed on the product listing page by the product's price in ascending order (lowest to highest).

At the first glance we may presume that the products obtained from the Redis database are already sorted in some predetermined order. However, if we want isomorphic handoff to succeed, we cannot operate in the realm of assumption; we must operate in the realm of fact. In order to do so, we need a predictable criteria for the sorting of products.

This is why we perform the additional work of implementing the `sort` interface for the `Products` type, so that we have a predictable criteria by which the products are listed on the products listing page. It provides us a benchmark when verifying the success of isomorphic handoff, since we simply need to confirm that the products listing page rendered on the client side is identical to the products listing page rendered on the server side. It is indeed helpful, that we have a common, predictable criteria that the products are sorted by price in the ascending order.

We add the following line (shown in bold) in the `GetProducts` method in the `redis.go` source file to sort the products:

```
func (r *RedisDatastore) GetProducts() []*models.Product {

    registryKey := "product-registry"
    exists, err := r.Cmd("EXISTS", registryKey).Int()

    if err != nil {
        log.Println("Encountered error: ", err)
        return nil
    } else if exists == 0 {
        return nil
    }

    var productKeys []string
    jsonData, err := r.Cmd("GET", registryKey).Str()
    if err != nil {
        log.Print("Encountered error when attempting to fetch product registry
data from Redis instance: ", err)
        return nil
    }

    if err := json.Unmarshal([]byte(jsonData), &productKeys); err != nil {
        log.Print("Encountered error when attempting to unmarshal JSON product
registry data: ", err)
        return nil
    }

    products := make(models.Products, 0)

    for i := 0; i < len(productKeys); i++ {

        productTitle := strings.Replace(productKeys[i], "/product-detail/", "", -1)
        product := r.GetProductDetail(productTitle)
        products = append(products, product)

    }
sort.Sort(products)
    return products
}
```

# Implementing isomorphic handoff for the products listing page

First, we must implement the **Encode** step. To do this, we need to decide the data that must be persisted to the client side. We can easily identify the data that must be persisted to the client, by examining the server side handler function, `ProductsHandler`, responsible for rendering the products listing web page:

```
func ProductsHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        products := env.DB.GetProducts()
        templateData := &templatedata.Products{PageTitle: "Products", Products: products}
        env.TemplateSet.Render("products_page", &isokit.RenderParams{Writer: w, Data: templateData})
    })
}
```

The products listing page is responsible for displaying the list of products, therefore, the `products` variable (shown in bold), a slice of `Product` objects, must be persisted to the client side.

Now that we have identified the data that needs to be persisted to the client, to maintain state, we can create a Rest API Endpoint, `GetProductsEndpoint`, that is responsible for delivering the slice of products to the client, in the JSON-encoded form:

```
func GetProductsEndpoint(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        products := env.DB.GetProducts()
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(products)
    })
}
```

Our work to implement isomorphic handoff is complete on the server side, and now it's time to turn our attention to the client side.

To implement the **Register** step, we add the following line to register the `/products` route and its associated handler function, `ProductsHandler`, in the `registerRoutes` function found in the `client.go` source file:

```
r.Handle("/products", handlers.ProductsHandler(env))
```

The **Decode** and **Attach** steps are performed within the `ProductsHandler` function:

```
func ProductsHandler(env *common.Env) isokit.Handler {
    return isokit.HandlerFunc(func(ctx context.Context) {

        productsChannel := make(chan []*models.Product)
        go FetchProducts(productsChannel)
        products := &productsChannel
        templateData := &templatedata.Products{PageTitle: "Products", Products:
            products}
        env.TemplateSet.Render("products_content", &isokit.RenderParams{Data:
            templateData, Disposition: isokit.PlacementReplaceInnerContents, Element:
            env.PrimaryContent, PageTitle: templateData.PageTitle})
        InitializeProductsPage(env)
        env.Router.RegisterLinks("#primaryContent a")
    })
}
```

First, we call the `FetchProducts` function a goroutine to fetch the list of products from the endpoint on the server side. The **Decode** step (shown in bold), is performed inside the `FetchProducts` function:

```
func FetchProducts(productsChannel chan []*models.Product) {

    data, err := xhr.Send("GET", "/restapi/get-products", nil)
    if err != nil {
        println("Encountered error: ", err)
        return
    }
    var products []*models.Product
    json.NewDecoder(strings.NewReader(string(data))).Decode(&products)

    productsChannel <- products
}
```

After fetching the encoded data from the Rest API endpoint, we use a JSON decoder to decode the encoded data back into a slice of `Product` objects. We then send the result over the `productsChannel`, where it is received inside the `ProductsHandler` function.

Now that we have the data object to populate the list of products on the products listing page, we can populate the `Products` field of the `templatedata.Products` struct. Recall that `templateData` is the data object that will be passed into the `Render` method of the `env.TemplateSet` object:

```
templateData := &templatedata.Products{PageTitle: "Products", Products:
    products}
```

```
env.TemplateSet.Render("products_content", &isokit.RenderParams{Data:  
templateData, Disposition: isokit.PlacementReplaceInnerContents, Element:  
env.PrimaryContent, PageTitle: templateData.PageTitle})
```

Up to this point, we have fulfilled the third step of the isomorphic handoff procedure, which means that we can effectively render the products listing page on the client side. However, we aren't done just yet, since we have to fulfill the last step of attaching DOM event handlers to the rendered web page.

Inside the `ProductsHandler` function, there are two calls that are instrumental to performing the **Attach** step:

```
InitializeProductsPage(env)  
env.Router.RegisterLinks("#primaryContent a")
```

First, we call the `InitializeProductsPage` function to add the event handlers necessary to enable user interactivity for the products listing page:

```
func InitializeProductsPage(env *common.Env) {  
  
    buttons := env.Document.GetElementsByClassName("addToCartButton")  
    for _, button := range buttons {  
        button.AddEventListener("click", false,  
        handleAddToCartButtonClickEvent)  
    }  
  
}
```

We retrieve all the **Add To Cart** buttons that exist on the product listing page, by calling the `GetElementsByClassName` method on the `env.Document` object, and specifying the "`addToCartButton`" class name.

When an **Add To Cart** button is clicked, the `handleAddToCartButtonClickEvent` function will be called. We will cover this function when we implement the shopping cart feature.

Let's return to the `ProductsHandler` function. We will call the `RegisterLinks` method on the Isokit router object and specify the CSS query selector of "`#primaryContent a`":

```
env.Router.RegisterLinks("#primaryContent a")
```

This ensures that when the web page is rendered on the client side, all click events for the product item links will be intercepted by the client-side router. This will allow us to render the product detail page on the client side itself, without having to perform a full page reload.

Up to this point, we have implemented the isomorphic handoff procedure for the products listing page. To render the products listing page on the client side, click on the **Products** link in the navigation bar. To render the products listing page on the server side, enter the following URL directly in the web browser: `http://localhost:8080/products`. *Figure 6.3* depicts the products listing page rendered on the client side:

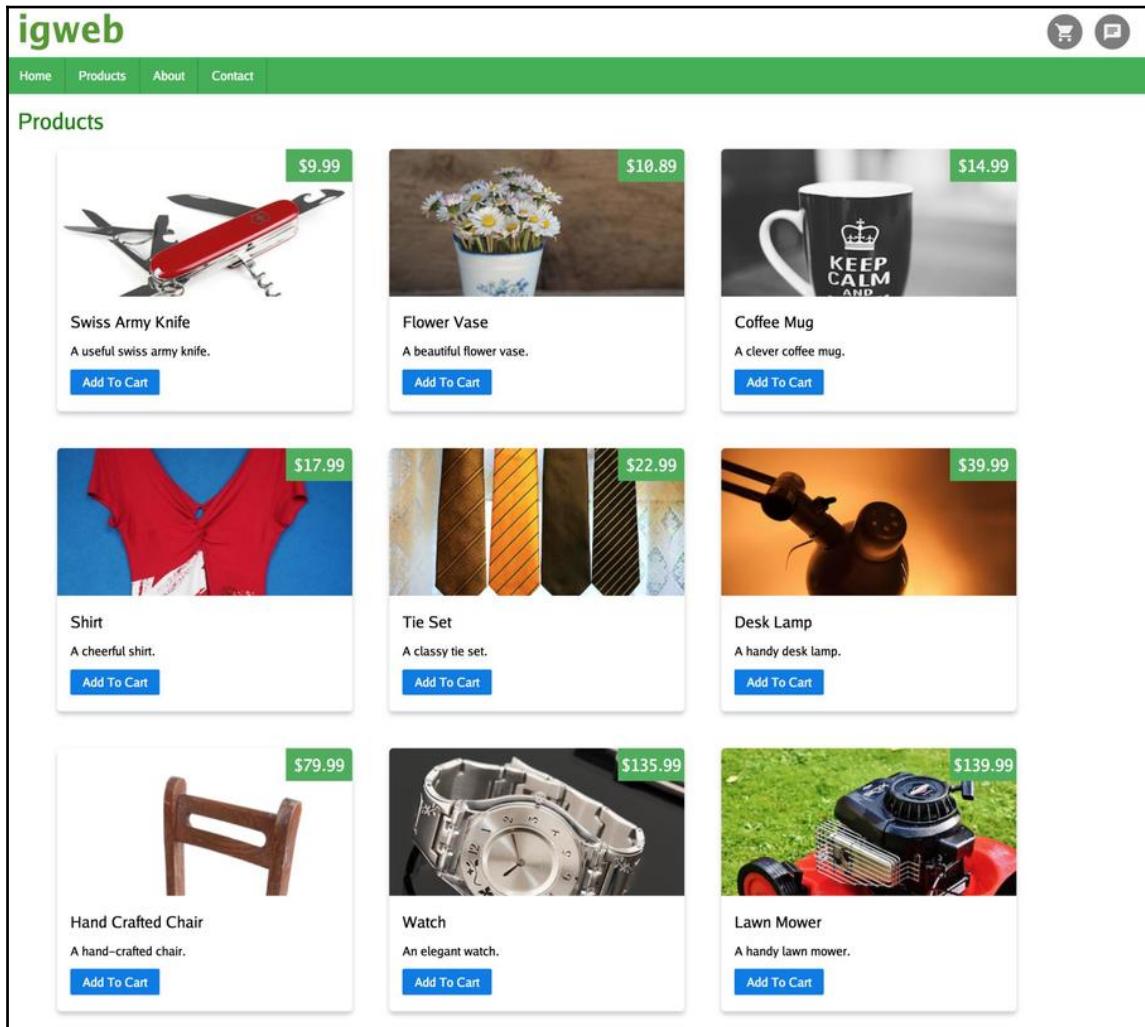


Figure 6.3: The products listing page rendered on the client side

You may also refresh the web page to force the page to be rendered on the server side. We can verify that the isomorphic handoff procedure was implemented properly by comparing the web page loaded on the client side with the web page that was loaded on the server side. Since both web pages are identical, we can determine that the isomorphic handoff procedure has been successfully implemented.

## Implementing isomorphic handoff for the product detail page

Having successfully implemented the isomorphic handoff procedure on the products listing page using the ERDA strategy, let's focus on implementing isomorphic handoff for the product detail page.

To implement the **Encode** step, we first need to identify the data object that will represent the state we wish to persist to the client. We identify the data object by examining the `ProductDetailHandler` function found in the `handlers/productdetail.go` source file. This is the server-side handler function responsible for servicing the `/product-detail` route:

```
func ProductDetailHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        vars := mux.Vars(r)
        productTitle := vars["productTitle"]
        product := env.DB.GetProductDetail(productTitle)
        templateData := &templatedata.ProductDetail{PageTitle: product.Name,
        Product: product}
        env.TemplateSet.Render("product_detail_page",
        &isokit.RenderParams{Writer: w, Data: templateData})
    })
}
```

The `product` object, a pointer to a `models.Product` struct, is obtained from the Redis datastore (shown in bold). This object contains the product data that will be displayed on the **Product** page; therefore, it is the object that we need to persist to the client side.

The `GetProductDetailEndpoint` function, found in the `endpoints/productdetail.go` source file, is the Rest API endpoint responsible for providing the JSON encoded Product data to the client:

```
func GetProductDetailEndpoint(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        vars := mux.Vars(r)
        productTitle := vars["productTitle"]
        product := env.DB.GetProductDetail(productTitle)
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(product)
    })
}
```

Inside the `GetProductDetailEndpoint` function, we obtain the product object from the Redis datastore and encode it as JSON formatted data.

Now that we have taken care of the **Encode** step, we can implement the next three steps on the client side.

To implement the **Register** step, we add the following line to register the `/product-detail` route and its associated handler function in the `client.go` source file:

```
r.Handle("/product-detail/{productTitle}",
    handlers.ProductDetailHandler(env))
```

The **Decode** and **Attach** steps are carried out by the `ProductDetailHandler` function:

```
func ProductDetailHandler(env *common.Env) isokit.Handler {
    return isokit.HandlerFunc(func(ctx context.Context) {
        routeVars := ctx.Value(isokit.RouteVarsKey("Vars")).(map[string]string)
        productTitle := routeVars[`product-detail/{productTitle}`]
        productChannel := make(chan *models.Product)
        go FetchProductDetail(productChannel, productTitle)
        product := &productChannel
        templateData := &templatedata.ProductDetail{PageTitle: product.Name,
            Product: product}
        env.TemplateSet.Render("product_detail_content",
            &isokit.RenderParams{Data: templateData, Disposition:
                isokit.PlacementReplaceInnerContents, Element: env.PrimaryContent,
                PageTitle: templateData.PageTitle})
        InitializeProductDetailPage(env)
    })
}
```

Inside the `ProductDetailHandler` function, we call the `FetchProductDetail` function as a goroutine to obtain the product object. The **Decode** step (shown in bold) is implemented inside the `FetchProductDetail` function:

```
func FetchProductDetail(productChannel chan *models.Product, productTitle string) {  
  
    data, err := xhr.Send("GET", "/restapi/get-product-detail"+"/"+productTitle, nil)  
    if err != nil {  
        println("Encountered error: ", err)  
        println(err)  
    }  
    var product *models.Product  
    json.NewDecoder(strings.NewReader(string(data))).Decode(&product)  
  
    productChannel <- product  
}
```

We make an XHR call to the Rest API endpoint to obtain the encoded `Product` data. We use a JSON decoder to decode the encoded data back into a `Product` object. We send the `Product` object over the `productChannel`, where it is received back in the `ProductDetailHandler` function.

Returning to the `ProductDetailHandler` function, we use the `product` data object to populate the product information on the product detail page. We do so by populating the `Product` field of the `templatedata.ProductDetail` object. Again, recall that the `templateData` variable is the data object that will be passed into the `Render` method of the `env.TemplateSet` object:

```
templateData := &templatedata.ProductDetail{PageTitle: product.Name,  
Product: product}  
env.TemplateSet.Render("product_detail_content",  
&isokit.RenderParams{Data: templateData, Disposition:  
isokit.PlacementReplaceInnerContents, Element: env.PrimaryContent,  
PageTitle: templateData.PageTitle})
```

Up to this point, we have fulfilled the third step of the isomorphic handoff procedure, which means that we can now render the product detail page on the client side. Now, it's time to complete the last step of the procedure, **Attach**, by attaching DOM event handlers to their respective UI elements on the rendered web page.

We call the `InitializeProductDetailPage` function to add the event handler necessary to enable user interactivity for the products listing page:

```
func InitializeProductDetailPage(env *common.Env) {  
  
    buttons := env.Document.GetElementsByTagName("addToCartButton")  
    for _, button := range buttons {  
        button.AddEventListener("click", false,  
        handleAddToCartButtonClickEvent)  
    }  
}
```

Similar to the `InitializeProductsPage` function, we retrieve all the **Add To Cart** buttons on the web page and specify the event handler function, `handleAddToCartButtonClickEvent`, that will be called when an **Add To Cart** button is clicked.

Up to this point, we have implemented the isomorphic handoff procedure for the product detail page. To render the product detail page on the client side, click on a product image found in the the products listing page. To render the product detail page on the server-side, enter the URL of a product in the web browser. For example, the URL to the product detail page for the swiss army knife is

`http://localhost:8080/product-detail/swiss-army-knife`. *Figure 6.4* depicts the product detail page for the swiss army knife, which was rendered on the client side:

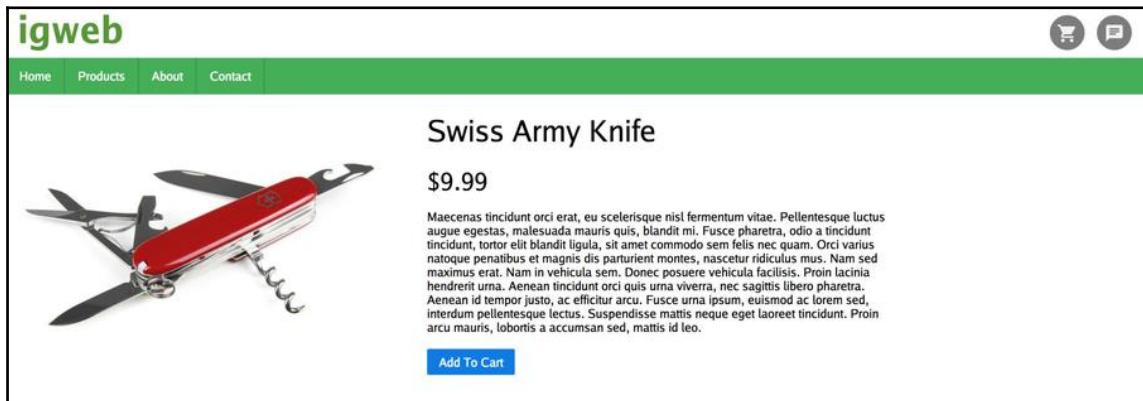


Figure 6.4: The product detail page rendered on the client side

Again, we can verify that the isomorphic handoff procedure is functioning properly by confirming that the web page rendered on the client side is identical to the web page rendered on the server side. Since both web pages are identical, we can conclude that we have successfully implemented the isomorphic handoff procedure for the product detail page.

## Implementing isomorphic handoff for the shopping cart

Now that we have implemented isomorphic handoff for the product-related web pages, it's time to start implementing IGWEB's shopping cart feature. We'll start by designing the **Shopping Cart** web page.

### Designing the shopping cart page

The design of the **Shopping Cart** page, as depicted in the wireframe design in *Figure 6.5*, is very similar to the product listings page. Each product item will contain a thumbnail sized image of the product, the product price, the product name, and a brief description of the product, just like the product listings page. In addition to these fields, the shopping cart page will have a field to display the quantity, that is, the amount of items of a particular product that are in the shopping cart, and a **Remove From Cart** button, that when clicked on, will remove the product from the shopping cart:

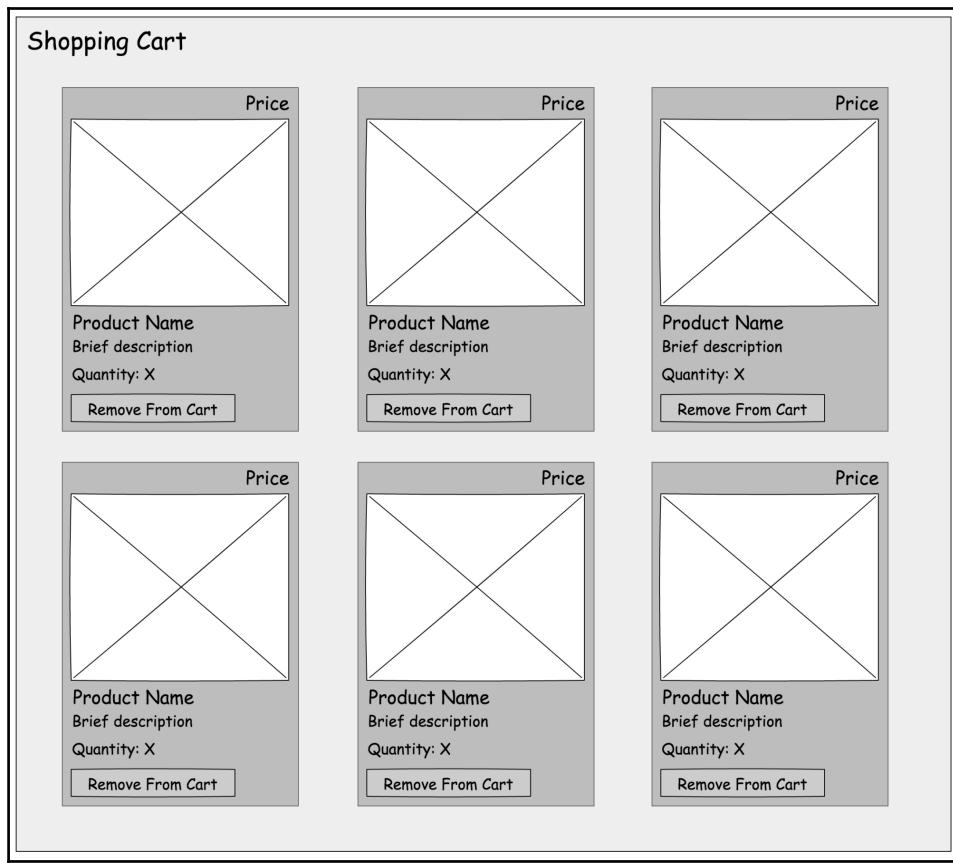


Figure 6.5: The wireframe design depicting the shopping cart page with products in the shopping cart

Keep in mind that the first wireframe design covers the scenario, when the shopping cart has been populated with items. We must also consider the design of the page when the shopping cart is empty. The shopping cart could either be empty on the user's initial visit to the IGWEB website or when the user empties the shopping cart entirely. *Figure 6.6* is a wireframe design of the shopping cart page, depicting the scenario where the shopping cart is empty:



Figure 6.6: The wireframe design depicting the shopping cart page when the shopping cart is empty

Now that we have the design of the shopping cart page locked down, it's time to implement the templates to realize the design.

## Implementing the shopping cart templates

We will use the shopping cart page template in order to render the shopping cart on the server side. Here are the contents of the shopping cart page template, as defined in the `shared/templates/shopping_cart_page.tpl` source file:

```
 {{ define "pagecontent" }}  
 {{template "shopping_cart_content" . }}  
 {{end}}  
 {{template "layouts/webpage_layout" . }}
```

As you may have noticed, the shopping cart page template calls a `shopping_cart_content` subtemplate, which is responsible for rendering the shopping cart itself.

Here are the contents of the shopping cart content template, as defined in the `shared/templates/shopping_cart_content tmpl` source file:

```
&lt;h1&gt;Shopping Cart&lt;/h1&gt;

{{if .Products }}
{{range .Products}}
  &lt;div class="productCard"&gt;
    &lt;a href="{{.Route}}&gt;
      &lt;div class="pricebox"&gt;&lt;span&gt;${{.Price}}&lt;/span&gt;&lt;/div&gt;
      &lt;div class="productCardImageContainer"&gt;
        &lt;img src="{{.ThumbnailPreviewURI}}&gt;
      &lt;/div&gt;
    &lt;/a&gt;
    &lt;div class="productContainer"&gt;

      &lt;h3&gt;&lt;b&gt;{{.Name}}&lt;/b&gt;&lt;/h3&gt;
      &lt;p&gt;{{.Description}}&lt;/p&gt;

      &lt;div class="productQuantity"&gt;&lt;span&gt;Quantity:
      {{.Quantity}}&lt;/span&gt;&lt;/div&gt;

      &lt;div class="pure-controls"&gt;
        &lt;button class="removeFromCartButton pure-button pure-button-primary" data-sku="{{.SKU}}&gt;Remove From Cart&lt;/button&gt;
      &lt;/div&gt;
    &lt;/div&gt;
  &lt;/div&gt;
{{end}}
{{else}}
  &lt;h2&gt;Your shopping cart is empty.&lt;/h2&gt;
{{end}}
```

Using the `if` action, we check to see whether there are any products to display in the shopping cart. If there are, we render each shopping cart item, using the `range` action. We render the name of the template, its thumbnail preview, and its description, along with its quantity. Finally, we render a button to remove the product from the shopping cart. Notice that we have embedded a data attribute called `data-sku` to include the product's unique SKU code along with the button element. This value come in handy later, when we make the XHR call to the Rest API endpoint responsible for removing a shopping cart item by clicking on this button.

If there are no items to display in the shopping cart, the flow of control reaches the `else` action. In this scenario, we will display the message that the shopping cart is empty.

Finally, we will use the `end` template action to signal the end of the `if-else` conditional blocks.

## The template data object

The template data object that will be passed to the template renderer will be a `templatedata.ShoppingCart` struct (defined in the `shared/templatedata/shoppingcart.go` source file):

```
type ShoppingCart struct {
    PageTitle string
    Products  []*models.Product
}
```

The `PageTitle` field will be used to display the web page title and the `Products` field, a slice of `Product` objects, will be used to display the products that are currently in the shopping cart.

Now that we have the templates in place, let's take a look at modeling the shopping cart.

## Modeling the shopping cart

The `ShoppingCartItem` struct, defined in the `shared/models/shoppingcart.go` source file, represents an item in the shopping cart:

```
type ShoppingCartItem struct {
    ProductSKU string `json:"productSKU"`
    Quantity   int    `json:"quantity"`
}
```

The `ProductSKU` field holds the SKU code of the product (the unique code used to distinguish a product) and the `Quantity` field holds the quantity of that particular product the user wishes to purchase. Each time the user hits the **Add To Cart** button on the products listing or product detail page, the quantity value for that particular product will be incremented in the shopping cart.

The `ShoppingCart` struct, also defined in the `shoppingcart.go` source file, represents the shopping cart:

```
type ShoppingCart struct {
    Items map[string]*ShoppingCartItem `json:"items"`
}
```

The `Items` field is a map of items having a key of the `string` type (which will be the product SKU code), and the value will be a pointer to a `ShoppingCartItem` struct.

The `NewShoppingCart` function is a constructor function to create a new instance of `ShoppingCart`:

```
func NewShoppingCart() *ShoppingCart {
    items := make(map[string]*ShoppingCartItem)
    return &ShoppingCart{Items: items}
}
```

The `ItemTotal` method of the `ShoppingCart` type is responsible for returning the number of items that are currently in the shopping cart:

```
func (s *ShoppingCart) ItemTotal() int {
    return len(s.Items)
}
```

The `IsEmpty` method of the `ShoppingCart` type is responsible for telling us if the shopping cart is empty or not:

```
func (s *ShoppingCart) IsEmpty() bool {  
  
    if len(s.Items) > 0 {  
        return false  
    } else {  
        return true  
    }  
  
}
```

The `AddItem` method of the `ShoppingCart` type is responsible for adding an item to the shopping cart:

```
func (s *ShoppingCart) AddItem(sku string) {  
  
    if s.Items == nil {  
        s.Items = make(map[string]*ShoppingCartItem)  
    }  
  
    _, ok := s.Items[sku]  
    if ok {  
        s.Items[sku].Quantity += 1  
  
    } else {  
        item := ShoppingCartItem{ProductSKU: sku, Quantity: 1}  
        s.Items[sku] = &item  
    }  
  
}
```

If a particular product item already exists in the shopping cart, the `Quantity` field will be incremented by one, upon each new request to add the product item.

Similarly, the `RemoveItem` method of the `ShoppingCart` type is responsible for removing all items of a specific product type from the shopping cart:

```
func (s *ShoppingCart) RemoveItem(sku string) bool {  
  
    _, ok := s.Items[sku]  
    if ok {  
        delete(s.Items, sku)  
        return true  
    } else {  
        return false  
    }
```

```
    }  
}
```

The `UpdateItemQuantity` method of the `ShoppingCart` type is responsible for updating the quantity of a specific product in the shopping cart:

```
func (s *ShoppingCart) UpdateItemQuantity(sku string, quantity int) bool {  
  
    _, ok := s.Items[sku]  
    if ok {  
        s.Items[sku].Quantity += 1  
        return true  
    } else {  
  
        return false  
    }  
}
```

## Shopping cart routes

By implementing the `ShoppingCart` type, we now have the business logic in place, to drive the shopping cart functionality. Now it's time to register the server-side routes that are needed to implement the shopping cart.

We register the `/shopping-cart` route along with its associated handler function, `ShoppingCartHandler`, inside the `registerRoutes` function, which is found in the `igweb.go` source file:

```
r.Handle("/shopping-cart", handlers.ShoppingCartHandler(env))
```

The route handler function, `ShoppingCartHandler`, is responsible for generating the web page for the shopping cart page on the server side.

We also register the following Rest API endpoints:

- Fetching a list of items (`/restapi/get-cart-items`)
- Adding an item (`/restapi/add-item-to-cart`)
- Removing an item (`/restapi/remove-item-from-cart`)

## Fetching a list of items

For fetching a list of items in the shopping cart, we will register the `/restapi/get-cart-items` endpoint:

```
r.Handle("/restapi/get-cart-items",
  endpoints.GetShoppingCartItemsEndpoint(env)).Methods("GET")
```

This endpoint will be handled by the `GetShoppingCartItemsEndpoint` handler function. This endpoint is responsible for encoding the shopping cart as JSON encoded data and providing it to the client-side application. Note that we use the HTTP `GET` method to call this endpoint.

## Adding an item

For adding an item to the shopping cart, we will register the `/restapi/add-item-to-cart` endpoint:

```
r.Handle("/restapi/add-item-to-cart",
  endpoints.AddItemToShoppingCartEndpoint(env)).Methods("PUT")
```

This route will be handled by the `AddItemToShoppingCartEndpoint` handler function. Note that since we are performing a mutating operation on the web server (adding a shopping cart item), we use the HTTP `PUT` method when calling this endpoint.

## Removing an item

For removing an item of a specific product type, and all quantities of it, from the shopping cart, we will register the `/restapi/remove-item-from-cart` endpoint:

```
r.Handle("/restapi/remove-item-from-cart",
  endpoints.RemoveItemFromShoppingCartEndpoint(env)).Methods("DELETE")
```

This endpoint will be handled by the `RemoveItemFromShoppingCartEndpoint` handler function. Again, note that we use the HTTP `DELETE` method when calling this endpoint since we are performing a mutating operation on the web server (removing a shopping cart item).

## The session store

Unlike the product records, which are stored in the Redis database, the items a user chooses to place in their shopping cart is transitory, and it's customized for individual use. This being the case, it makes much more sense to store the state of the shopping cart in a session, rather than in the database.

We will use the Gorilla sessions package to create sessions and store data to the sessions. We will utilize the `session.NewFileSystemStore` type to save the session data to the server's file system.

First, we will add a new field (shown in bold) to the `common.Env` struct (defined in the `common/common.go` source file), which will hold the `FileSystemStore` instance so that it is accessible throughout the server-side web application:

```
type Env struct {
    DB datastore.Datastore
    TemplateSet *isokit.TemplateSet
    Store *sessions.FileSystemStore
}
```

Inside the `main` function defined in the `igweb.go` source file, we will make a call to the `initializeSessionstore` function and pass in the `env` object:

```
initializeSessionstore(&env)
```

The `initializeSessionstore` function is responsible for creating the session store on the server side:

```
func initializeSessionstore(env *common.Env) {
    if _, err := os.Stat("/tmp/igweb-sessions"); os.IsNotExist(err) {
        os.Mkdir("/tmp/igweb-sessions", 711)
    }
    env.Store = sessions.NewFilesystemStore("/tmp/igweb-sessions",
        []byte(os.Getenv("IGWEB_HASH_KEY")))
}
```

In the `if` conditional, we first check to see whether the designated path where the session data will be stored, `/tmp/igweb-sessions`, exists. If the path does not exist, we create the folder by calling the `Mkdir` function from the `os` package.

We will initialize a new file system session store by calling the `NewFileSystemStore` function in the `sessions` package, passing in the path where sessions will be saved and the authentication key for the session. We will populate the `Store` property of the `env` object with the newly created `FileSystemStore` instance.

Now that we have the session store in place, let's implement the server-side `ShoppingCartHandler` function.

## The server-side shopping cart handler function

The `ShoppingCartHandler` function, defined in `handlers/shoppingcart.go`, is responsible for servicing the `/shopping-cart` route:

```
func ShoppingCartHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
```

The main purpose of the server-side shopping cart handler function is to produce the output for the shopping cart web page.

Recall that the handler functions for the product-related pages, tapped into the Redis datastore to retrieve the list of products. The shopping cart handler, on the other hand, taps into the server-side session to get the list of items that are in the shopping cart.

We will declare the variables for the `templateData` object and the shopping cart:

```
var templateData *templatedata.ShoppingCart
var cart *models.ShoppingCart
```

We've define and initialize the `igwSession` variable of the `gorilla.SessionStore` type, which will hold our server-side session:

```
igwSession, _ := env.Store.Get(r, "igweb-session")
```

Recall that we can access the `FileSystemStore` object by accessing the `Store` property of the `env` object. We call the `Get` method of the session store object passing in the pointer to `http.Request`, `r`, and the name of the session, `"igweb-session"`.

If the session does not exist, a new session named `"igweb-session"` will automatically be created for us.

To access values in the session, we use the `Values` property of the `igwSession` object, which is a map of keys to values. The keys are strings, and the values are of type `empty interface`, `interface{ }`, so that they can hold any type (since all types in Go implement the empty interface).

In the `if` conditional block, we check to see whether a value for the `"shoppingCart"` session key exists in the `Values` map:

```
if _, ok := igwSession.Values["shoppingCart"]; ok == true {
    // Shopping cart exists in session
    decoder := json.NewDecoder(strings.NewReader(string(igwSession.Values["shoppingCart"])[
        []byte()]))

    err := decoder.Decode(&cart)
    if err != nil {
        log.Println("Encountered error when attempting to decode json data
from session: ", err)
    }
}
```

The JSON-encoded value for the shopping cart object is accessed using the `"shoppingCart"` key. If the shopping cart exists in the session, we decode the JSON object using the `Decode` method of the JSON decoder object. If the JSON object is successfully decoded, the decoded object is stored in the `cart` variable.

Now that we have the shopping cart object from the session, we need to get the product information for each item in the shopping cart. We do so by calling the `GetProductsInShoppingCart` method of the `datastore` object and supplying the `cart` variable as an input argument to the method:

```
products := env.DB.GetProductsInShoppingCart(cart)
```

This function will return the slice of products that are to be displayed on the shopping cart page. Notice that we populate the `Products` field of the `templatedata.ShoppingCart` object with the slice of products obtained from the `datastore`:

```
templateData = &templatedata.ShoppingCart{PageTitle: "Shopping Cart",
Products: products}
```

Since we will utilize this slice of products to render the server-side shopping cart template page, the slice of products returned from the `GetProductsInShoppingCart` method is the state data that we need to persist to the client-side when implementing isomorphic handoff.

If the "shoppingCart" key did not exist in the session, the flow of control reaches the `else` block:

```
    } else {
        // Shopping cart doesn't exist in session
        templateData = &templatedata.ShoppingCart{PageTitle: "Shopping Cart",
Products: nil}
    }
```

In this situation, we set the `Products` field of the `templatedata.ShoppingCart` struct to `nil`, to indicate that there are no products in the shopping cart since the shopping cart does not exist in the session.

Finally, we render the shopping cart page by calling the `Render` method on the template set object, passing in the name of the template we wish to render (the `shopping_cart_page` template) along with the render parameters:

```
    env.TemplateSet.Render("shopping_cart_page", &isokit.RenderParams{Writer:
w, Data: templateData})
    })
}
```

Notice that we have set the `RenderParams` object's `Writer` property to `http.ResponseWriter`, `w`, and we have set the `Data` property to the `templateData` variable.

Let's take a look at the `GetProductsInShoppingCart` method defined in the Redis datastore (found in the `common/datastore/redis.go` source file):

```
func (r *RedisDatastore) GetProductsInShoppingCart(cart
*models.ShoppingCart) []*models.Product {

    products := r.GetProducts()
    productsMap := r.GenerateProductsMap(products)

    result := make(models.Products, 0)
    for _, v := range cart.Items {
        product := &models.Product{}
        product = productsMap[v.ProductSKU]
        product.Quantity = v.Quantity
        result = append(result, product)
    }
    sort.Sort(result)
    return result
}
```

The job of this method is to return a slice of `Product` objects for all the products that exist within a shopping cart. The `ShoppingCart` struct simply keeps track of a product's type (through its `SKU` code) and the `Quantity` of that product in the shopping cart.

We declare a `result` variable, which is a slice of the `Product` objects. We loop through each shopping cart item, and we retrieve the `Product` object from the `productsMap`, providing the product's `SKU` code as the key. We populate the `Quantity` field of the `Product` object and append the `Product` object to the `result` slice.

We call the `Sort` method from the `sort` package passing in the `result` slice. Since we have implemented the `sort` interface for the `Products` type, the `Product` objects in the `result` slice will be sorted by price in the ascending order. Finally, we return the `result` slice.

## Shopping cart endpoints

At this point, while we are finishing up the server-side functionality to implement the shopping cart feature, we are also ready to start implementing the isomorphic handoff procedure, following the ERDA strategy.

### The endpoint to get items in the shopping cart

Let's examine the shopping cart's Rest API endpoints, which help service actions that the client-side web application is dependent on. Let's start with the endpoint function, `GetShoppingCartItemsEndpoint`, which is responsible for getting the items in the shopping cart. The **Encode** step, of the isomorphic handoff procedure is performed in this endpoint function.

Here's the source listing of the `GetShoppingCartItemsEndpoint` function:

```
func GetShoppingCartItemsEndpoint(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        var cart *models.ShoppingCart
        igwSession, _ := env.Store.Get(r, "igweb-session")
        if _, ok := igwSession.Values["shoppingCart"]; ok == true {
            // Shopping cart exists in session
            decoder := json.NewDecoder(strings.NewReader(string(igwSession.Values["shoppingCart"]).
                ([]byte)))
            err := decoder.Decode(&cart)
            if err != nil {

```

```
        log.Println("Encountered error when attempting to decode json data
from session: ", err)
    }

    products := env.DB.GetProductsInShoppingCart(cart)
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(products)

} else {
    // Shopping cart doesn't exist in session
    cart = nil
    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(cart)
    return
}

})
}
```

In this function, we fetch the shopping cart from the session. If we are able to successfully fetch the shopping cart from the session, we use a JSON encoder to encode the ShoppingCart object and write it using http.ResponseWriter, w.

If the shopping cart does not exist in the session, then we simply JSON encode the value of nil (which is equivalent to a JavaScript null on the client side) and write it out in the response, using http.ResponseWriter, w.

With this code in place, we have fulfilled the Encode step in the isomorphic handoff procedure.

## The endpoint to add items to the shopping cart

We declared a **m** variable (shown in bold), of the map[string]string type, in AddItemToShoppingCartEndpoint, which is the endpoint function that is responsible for adding a new item to the shopping cart:

```
func AddItemToShoppingCartEndpoint(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        igwSession, _ := env.Store.Get(r, "igweb-session")
        decoder := json.NewDecoder(r.Body)
var m map[string]string
        err := decoder.Decode(&m)
        if err != nil {
            log.Println("Encountered error when attempting to decode json data from
```

```
    request body: ", err)
    }
    defer r.Body.Close()

    var cart *models.ShoppingCart
```

We use a JSON decoder to decode the request body, which will contain a JSON encoded map sent from the client. The map will contain the `SKU` value of the product to add to the shopping cart, given the `"productSKU"` key.

We will check to see whether the shopping cart exists in the session. If it does, we will decode the shopping cart JSON data back into a `ShoppingCart` object:

```
if _, ok := igwSession.Values["shoppingCart"]; ok == true {
    // Shopping Cart Exists in Session
    decoder := json.NewDecoder(strings.NewReader(string(igwSession.Values["shoppingCart"]).
        ([]byte)))
    err := decoder.Decode(&cart)
    if err != nil {
        log.Println("Encountered error when attempting to decode json data
from session: ", err)
    }
}
```

If the shopping cart doesn't exist, flow of control reaches the `else` block, and we will create a new shopping cart:

```
} else {
    // Shopping Cart Doesn't Exist in Session, Create a New One
    cart = models.NewShoppingCart()
}
```

We will then make a call to the `AddItem` method of the `ShoppingCart` object to add the product item:

```
cart.AddItem(m["productSKU"])
```

To add an item to our shopping cart, we simply have to provide the product's `SKU` value, which we can obtain from the `m` map variable by accessing the value that exists in the map for the `productSKU` key.

We will encode the cart object into its JSON representation and save it into the session, with the session key "shoppingCart":

```
b := new(bytes.Buffer)
w.Header().Set("Content-Type", "application/json")
err = json.NewEncoder(b).Encode(cart)
if err != nil {
    log.Println("Encountered error when attempting to encode cart struct as
json data: ", err)
}
igwSession.Values["shoppingCart"] = b.Bytes()
igwSession.Save(r, w)
w.Write([]byte("OK"))
})
```

We then write back the response, "OK", to the client, to indicate that the operation to add a new item to the cart was performed successfully.

## The endpoint to remove items from the shopping cart

Here's the source listing of RemoveItemFromShoppingCartEndpoint, the endpoint that is responsible for removing all items of a specific product from the shopping cart:

```
func RemoveItemFromShoppingCartEndpoint(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        igwSession, _ := env.Store.Get(r, "igweb-session")
        decoder := json.NewDecoder(r.Body)
        var m map[string]string
        err := decoder.Decode(&m)
        if err != nil {
            log.Println("Encountered error when attempting to decode json data from
request body: ", err)
        }
        defer r.Body.Close()

        var cart *models.ShoppingCart
        if _, ok := igwSession.Values["shoppingCart"]; ok == true {
            // Shopping Cart Exists in Session
            decoder :=
                json.NewDecoder(strings.NewReader(string(igwSession.Values["shoppingCart"].
                ([]byte))))
            err := decoder.Decode(&cart)
            if err != nil {
                log.Println("Encountered error when attempting to decode json data
from session: ", err)
            }
        }
    })
}
```

```
        }
    } else {
        // Shopping Cart Doesn't Exist in Session, Create a New One
        cart = models.NewShoppingCart()
    }

    for k, v := range cart.Items {
        if v.ProductSKU == m["productSKU"] {
            delete(cart.Items, k)
        }
    }

    b := new(bytes.Buffer)
    w.Header().Set("Content-Type", "application/json")
    err = json.NewEncoder(b).Encode(cart)
    if err != nil {
        log.Println("Encountered error when attempting to encode cart struct as
json data: ", err)
    }
    igwSession.Values["shoppingCart"] = b.Bytes()
    igwSession.Save(r, w)

    w.Write([]byte("OK"))

    })
}
```

Remember that we can have multiple quantities for a given product. In the current shopping cart implementation, if the user clicks on the **Remove From Cart** button, the given product (and all quantities of it) are removed from the shopping cart.

We start out by fetching the JSON encoded shopping cart data from the session. If it exists, we decode the JSON object into a new `ShoppingCart` object. If the shopping cart does not exist in the session, we simply create a new shopping cart.

We range through the items found in the shopping cart, and if we are able to find a product in the cart containing the same product SKU code supplied in the `m` map variable that was obtained from the client-side web application, we will remove the element from the shopping cart object's `Items` map by calling the built-in `delete` function (shown in bold). Finally, we will write out a JSON encoded response to the client that the operation was successfully completed.

Now that we have our server-side endpoints in place, it's time to take a look at the functionality needed on the client side to implement the final pieces of the shopping cart feature.

# Implementing the shopping cart functionality on the client side

To fulfill the **Register** step of the ERDA strategy, we will register the `/shopping-cart` route and its associated handler function, `ShoppingCartHandler`, inside the `registerRoutes` function found in the `client/client.go` source file:

```
r.Handle("/shopping-cart", handlers.ShoppingCartHandler(env))
```

Remember that this route will kick-in when a user access the shopping cart by clicking on the shopping cart icon in the navigation bar. Upon clicking the shopping cart icon, the `ShoppingCartHandler` function will be called.

Let's take a look at the `ShoppingCartHandler` function:

```
func ShoppingCartHandler(env *common.Env) isokit.Handler {
    return isokit.HandlerFunc(func(ctx context.Context) {
        renderShoppingCartItems(env)
    })
}
```

The main purpose of this function is to call the `renderShoppingCartItems` function to render the shopping cart on the client side. We have consolidated the logic to render the shopping cart and its contents in the `renderShoppingCartItems` function so that the shopping cart page can be re-rendered as the user removes items from the shopping cart.

## Rendering the shopping cart

The `renderShoppingCartItems` function is responsible for conducting the last two steps of the ERDA strategy, the **Decode** and **Attach** steps. Here's the source listing of the `renderShoppingCartItems` function:

```
func renderShoppingCartItems(env *common.Env) {
    productsChannel := make(chan []*models.Product)
    go fetchProductsInShoppingCart(productsChannel)
    products := &lt;-productsChannel
    templateData := &templatedata.ShoppingCart{PageTitle: "Shopping Cart",
        Products: products}
    env.TemplateSet.Render("shopping_cart_content",
        &isokit.RenderParams{Data: templateData, Disposition:
            isokit.PlacementReplaceInnerContents, Element: env.PrimaryContent,
            PageTitle: templateData.PageTitle})
```

```
    InitializeShoppingCartPage (env)
    env.Router.RegisterLinks ("#primaryContent a")
}
```

In this function, we create a new channel called `productsChannel`, which is a channel that we'll use to send and receive a slice of products over. We call the `fetchProductsInShoppingCart` function as a goroutine and provide `productsChannel` as an input argument to the function. This function is responsible for fetching the product items in the shopping cart from the server by performing an XHR call.

Here's the source listing for the `fetchProductsInShoppingCart` function:

```
func fetchProductsInShoppingCart (productsChannel chan []*models.Product) {

    data, err := xhr.Send("GET", "/restapi/get-cart-items", nil)
    if err != nil {
        println("Encountered error: ", err)
        println(err)
    }
    var products []*models.Product
    json.NewDecoder(strings.NewReader(string(data))).Decode(&products)

    productsChannel <- products
}
```

In this function, we simply make an XHR call to the Rest API endpoint, `/restapi/get-cart-items`, which is responsible for returning the JSON-encoded data that represents the slice of products. We use a JSON decoder to decode the encoded slice of products into the `products` variable. Finally, we send the `products` variable over `productsChannel`.

Let's return to the `renderShoppingCartItems` function and receive the slice of products from the `productsChannel`, and then, we will set the `Products` property of the `templateData` object with the received products:

```
templateData := &templatedata.ShoppingCart{PageTitle: "Shopping Cart",
Products: products}
```

We will then render the shopping cart template on the client side:

```
env.TemplateSet.Render("shopping_cart_content", &isokit.RenderParams{Data:
templateData, Disposition: isokit.PlacementReplaceInnerContents, Element:
env.PrimaryContent, PageTitle: templateData.PageTitle})
```

At this point, we have fulfilled the **Decode** step of the ERDA strategy.

To fulfill the **Attach** step of the ERDA strategy, we will make a call to the `InitializeShoppingCartEventHandlers` function to attach any needed event listeners to the shopping cart web page.

Here's the source listing of the `InitializeShoppingCartEventHandlers` function:

```
func InitializeShoppingCartPage(env *common.Env) {  
  
    buttons := env.Document.GetElementsByTagName("removeFromCartButton")  
    for _, button := range buttons {  
        button.AddEventListener("click", false,  
            func(event dom.Event) {  
                handleRemoveFromCartButtonClickEvent(env, event)  
  
            })  
    }  
  
}
```

This function is responsible for attaching a click event on all the **Remove From Cart** buttons that are found in each product container listed on the shopping cart web page. The event handler function that is called when the **Remove From Cart** button is clicked, is the `handleRemoveFromCartButtonClickEvent` function.

We have now fulfilled, the fourth and last step of the ERDA strategy, by attaching event listeners to the **Remove From Cart** buttons on the shopping cart web page. The implementation of isomorphic handoff for the shopping cart feature is complete.

## Removing an item from the shopping cart

Let's take a look at the `handleRemoveFromCartButtonClickEvent` function, which gets called when the **Remove From Cart** button is clicked:

```
func handleRemoveFromCartButtonClickEvent(env *common.Env, event dom.Event)  
{  
    productSKU := event.Target().GetAttribute("data-sku")  
    go removeFromCart(env, productSKU)  
}
```

In this function, we obtain the product SKU code from the event target element's `data-sku` attribute. We then call the `removeFromCart` function as a goroutine, passing in the `env` object and `productSKU`.

Here is the source listing of the `removeFromCart` function:

```
func removeFromCart(env *common.Env, productSKU string) {  
  
    m := make(map[string]string)  
    m["productSKU"] = productSKU  
    jsonData, _ := json.Marshal(m)  
  
    data, err := xhr.Send("DELETE", "/restapi/remove-item-from-cart",  
    jsonData)  
    if err != nil {  
        println("Encountered error: ", err)  
        notify.Error("Failed to remove item from cart!")  
        return  
    }  
    var products []*models.Product  
    json.NewDecoder(strings.NewReader(string(data))).Decode(&products)  
    renderShoppingCartItems(env)  
    notify.Success("Item removed from cart")  
}
```

We create a new map, `m`, in the `removeFromCart` function, which is used to house `productSKU`. We can access the product's SKU value from the `m` map by providing the `"productSKU"` key. We have intended to send this map to the web server through the request body. The reason that we have chosen the `map` type, as opposed to simply sending the product's SKU string value, is that we want to make our solution extensible. In the future, if there is any additional information that should be sent to the server, we can include that value as part of an additional key-value pair in the map.

We encode the map into its JSON representation and make an XHR call to the web server, sending the map JSON data. Finally, we make a call to the `renderShoppingCartItems` function to render the shopping cart items. Remember that by calling this function, we will be performing an XHR call to get the latest products in the shopping cart (which represents the current state of the shopping cart). This ensures that we will have the most up-to-date state of the shopping cart, since again, we are using the server-side session (where the shopping cart state is stored) as our single source of truth.

## Adding an item to the shopping cart

The functionality for the **Add To Cart** button is implemented in a similar manner. Recall that on the product-related pages, if any **Add To Cart** button is clicked, the `handleAddToCartButton` function is called. Here is the source listing of the function:

```
func handleAddToCartButtonClickEvent (event dom.Event) {
    productSKU := event.Target().GetAttribute("data-sku")
    go addToCart(productSKU)
}
```

In a manner similar to the `handleRemoveFromCartButtonClickEvent` function, inside the `handleAddToCart` function, we obtain the product's SKU code from the event target element by getting the data attribute with the "data-sku" key. We then call the `addToCart` function as a goroutine and supply `productSKU` as an input argument to the function.

Here's the source listing of the `addToCart` function:

```
func addToCart (productSKU string) {
    m := make(map[string]string)
    m["productSKU"] = productSKU
    jsonData, _ := json.Marshal(m)

    data, err := xhr.Send("PUT", "/restapi/add-item-to-cart", jsonData)
    if err != nil {
        println("Encountered error: ", err)
        notify.Error("Failed to add item to cart!")
        return
    }
    var products []*models.Product
    json.NewDecoder(strings.NewReader(string(data))).Decode(&products)
    notify.Success("Item added to cart")
}
```

Inside the `addToCart` function, we make an XHR call to the Rest API endpoint on the web server that is responsible for adding an item to the cart. Prior to making the XHR call, we create a map containing `productSKU`, and then we encode the map into its JSON representation. We send the JSON data using an XHR call to the server endpoint.

We can now display the shopping cart on the client side, and we can also accommodate the user interactions with the shopping cart, notably adding a product to the shopping cart and removing a product from the shopping cart.



The shopping cart implementation that was covered in this chapter is meant for illustration purposes only. It is up to the reader to implement further functionality.

## Verifying the shopping cart functionality

Now it's time to verify that the state of the shopping cart is maintained from the server to the client, as the user adds and removes items from the shopping cart.

Verifying that isomorphic handoff was successfully implemented is straightforward. We simply need to verify that the server-side generated shopping cart page is identical to the client-side generated shopping cart page. By clicking on the shopping cart icon, we can see the client-side generated web page. By clicking on the refresh button while on the shopping cart page, we can see the server-side generated web page.

Starting out, there are no items placed in the shopping cart. *Figure 6.7* is a screenshot depicting the shopping cart in its empty state:

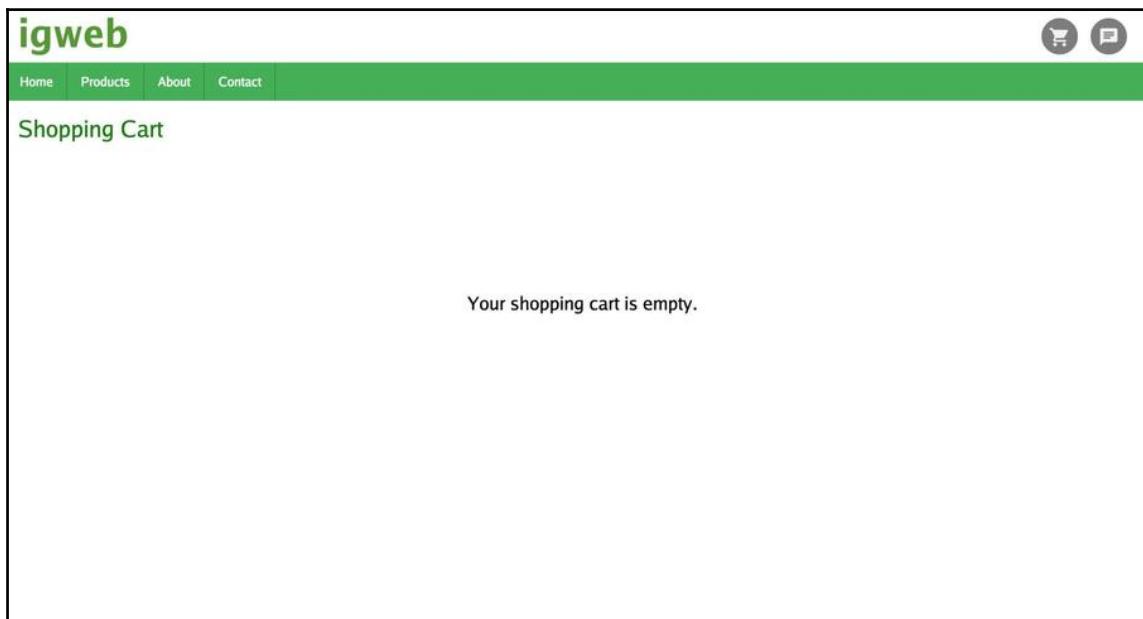


Figure 6.7: The shopping cart page when the shopping cart is empty

The shopping cart page rendered on the client side matches the page rendered on the server side, indicating that the empty state of the shopping cart is maintained properly.

Now, let's visit the product listings page by clicking on the **Products** link on the navigation bar. Let's add a few items to the shopping cart by clicking on the **Add To Cart** button. Let's click on the shopping cart icon in the top bar of the website to return to the shopping cart page. *Figure 6.8* is a screenshot depicting the shopping cart with some products added:

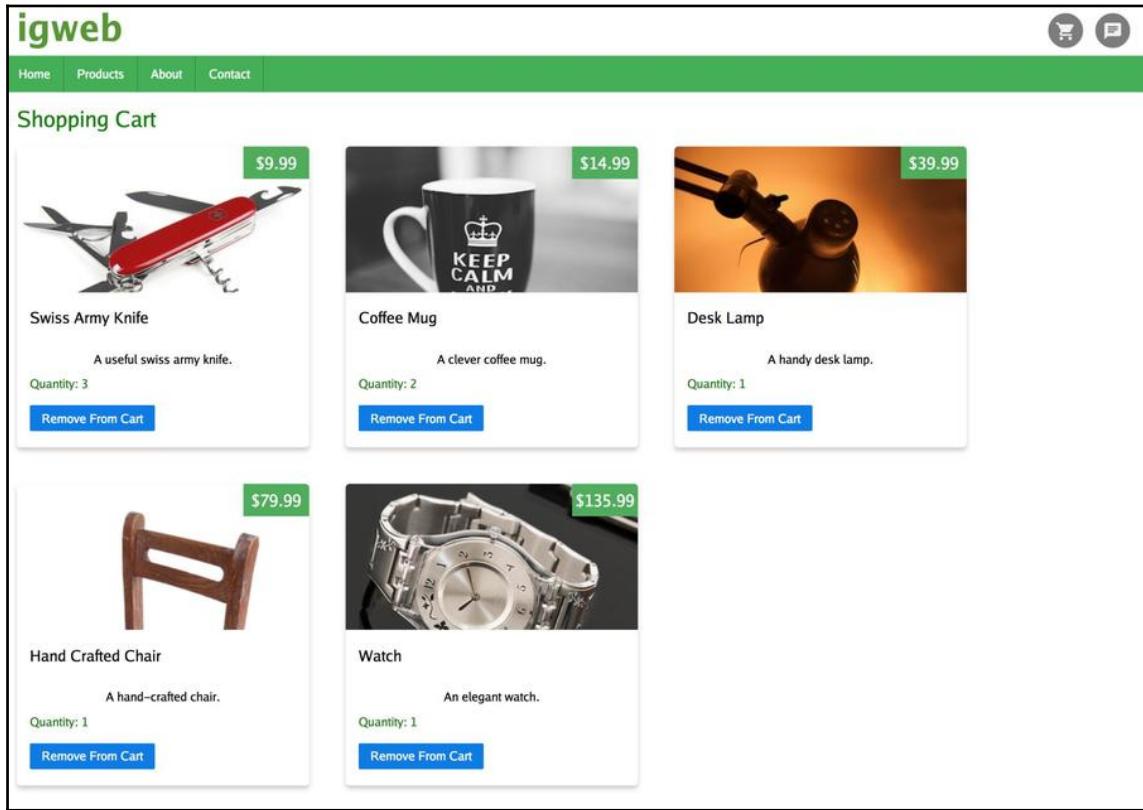


Figure 6.8: The shopping cart page with some products in the shopping cart

After checking that the shopping cart page rendered on the client side matches the page rendered on the server side, we can determine that the state of the shopping cart has been maintained successfully.

Now remove all the items from the shopping cart by clicking on the **Remove From Cart** button on each product. Once the shopping cart is empty, we can perform the same verification step of checking the page rendered on the client side with the page rendered on the server side to determine that the shopping cart state is successfully maintained.

At this point, we can acknowledge that the isomorphic handoff procedure has been implemented successfully for the shopping cart feature.

You may have noticed that as we add items to the shopping cart, a notification is displayed on the lower right-hand side of the screen, as displayed in *Figure 6.9*. Notice that the notification is displayed at the lower right-hand side of the web page, and it indicates that the product has been successfully added to the shopping cart:

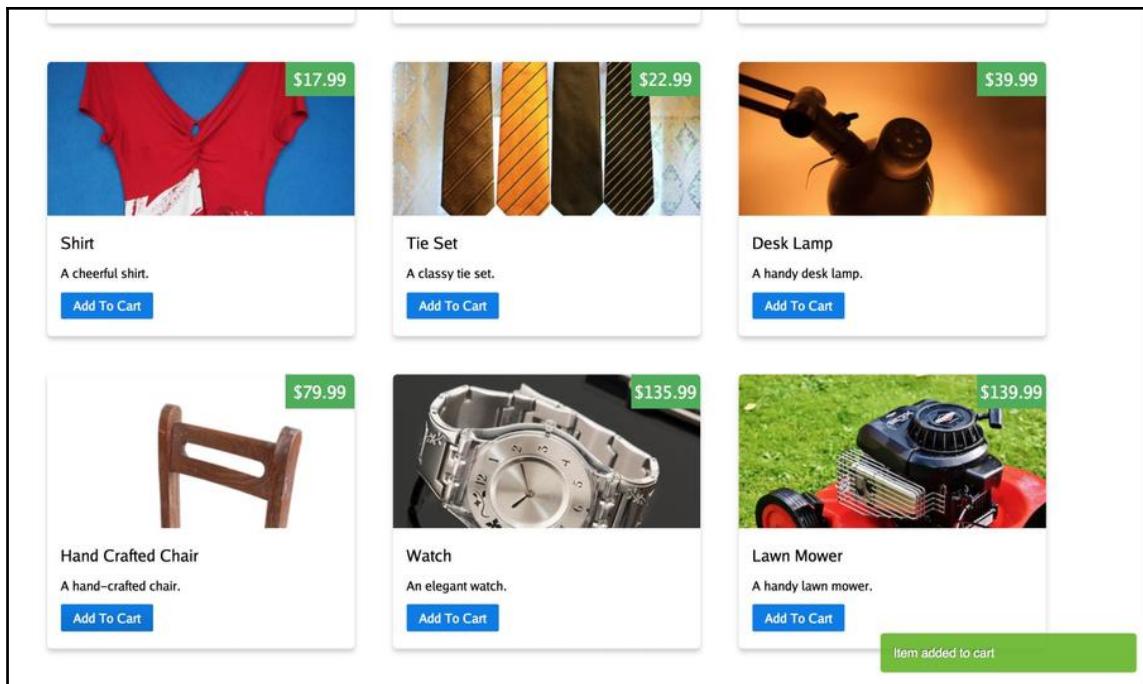


Figure 6.9: A notification appears on the page's lower right-hand side when an item has been added to the shopping cart

Note that similar notifications also appear when items are removed from the shopping cart. We utilized a cog, a reusable component, to generate this notification. We will cover the implementation of the cog responsible for generating these notifications in *Chapter 9, Cogs – Reusable Components*.

## Summary

In this chapter, we introduced you to *isomorphic handoff*, the means by which the server passes off state to the client. This is a significant procedure, that allows the client to pick up where the server left off in an isomorphic web application. We demonstrated the ERDA strategy to implement isomorphic handoff for the product-related web pages as well as the shopping cart web page. While implementing the shopping cart feature, we created a server-side session store, which acted as the source of truth for the current state of the user's shopping cart. We implemented server-side endpoints to implement the functionality to get items from the shopping cart, add items to the shopping cart, and remove items from the shopping cart. Finally, we verified that isomorphic handoff was successfully implemented by confirming that the web page rendered on the client side was identical to the the web page rendered on the server side.

We also relied on sources of truth on the server side to maintain state with the client. For the product-related pages, the source of truth was the Redis datastore, and for the shopping cart page, the single source of truth was the server-side session store. In [Chapter 7, \*The Isomorphic Web Form\*](#), we'll consider how to handle situations that go beyond basic user interactions. You'll learn how to accept user generated data from the client side, submitted through an isomorphic web form. You will learn how to validate and process the data submitted by the user, by implementing the contact form on IGWEB's **Contact** web page.

# 7

## The Isomorphic Web Form

In the previous chapter, we focused on how we could have the server-side application handoff data to the client-side application to seamlessly maintain state while implementing the shopping cart feature. In [Chapter 6, Isomorphic Handoff](#), we treated the server, as the single source of truth. The server dictated, to the client, what the current state of the shopping cart was. In this chapter, we are going to go beyond the simple user interactions that we have considered thus far and step into the realm of accepting user-generated data submitted through an isomorphic web form.

This signifies that now, the client has a voice, to dictate the user-generated data that should be stored on the server, within good reason of course (validation of user-submitted data). Using an isomorphic web form, validation logic can be shared across environments. The client-side application can chip in and inform the user that they've made a mistake prior to the form data being submitted to the server. The server-side application has the ultimate veto power because it will rerun the validation logic on the server side (where, ostensibly, the validation logic can't be tampered with) and process the user-generated data only upon a successful validation result.

Besides providing the ability to share validation logic and form structure, isomorphic web forms also provide a means to make forms more accessible. We must address accessibility concerns for web clients that may not have a JavaScript runtime or may have the JavaScript runtime disabled. To accomplish this goal, we will build an isomorphic web form for the contact section of IGWEB, with progressive enhancement in mind. This means that only after implementing form functionality to satisfy the bare-minimum, JavaScript-disabled web client scenario, we will proceed to implement the client-side form validation that runs directly within a JavaScript equipped web browser.

By the end of this chapter, we'll have a robust, isomorphic web form, implemented in a single language (Go), which will reuse common code across environments. Most importantly, the isomorphic web form will be accessible to the most stripped down web client running in a terminal window, and at the same time, accessible to the GUI-based web client with the latest JavaScript runtime.

In this chapter, we will cover the following topics:

- Understanding the form flow
- Designing the contact form
- Validating email address syntax
- The form interface
- Implementing the contact form
- The accessible contact form
- Client-side considerations
- Contact form Rest API Endpoint
- Checking the client-side validation

## Understanding the form flow

Figure 7.1 depicts an image showing the web form with only server-side validation in place. The form is submitted to the web server through an HTTP Post request. The server provides a fully rendered web page response. If the user did not fill out the form properly, errors will be populated and displayed in the web page response. If the user did fill out the form properly, a HTTP redirect will be made to the confirmation web page:

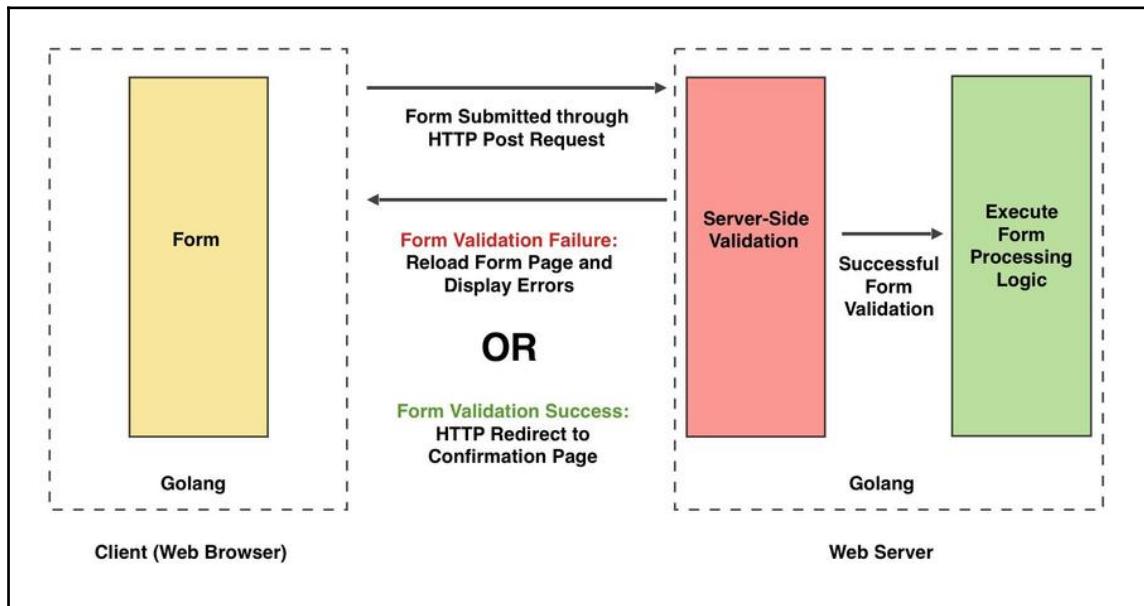


Figure 7.1: The web form with server-side validation only

Figure 7.2 depicts an image showing the web form with both client-side and server-side validation in place. When the user submits the web form, the data in the form is validated using client-side validation. The form data will be submitted to the web server using an XHR call to a Rest API endpoint, only upon a successful client-side validation result. Once the form data has been submitted to the server, it will undergo a second round of server-side validation. This ensures the quality of the form data even in a scenario where the client-side validation may have been tampered with. The client-side application will inspect the form validation result returned from the server and will either display the confirmation page upon a successful form submission or will display the contact form errors, upon an unsuccessful form submission:

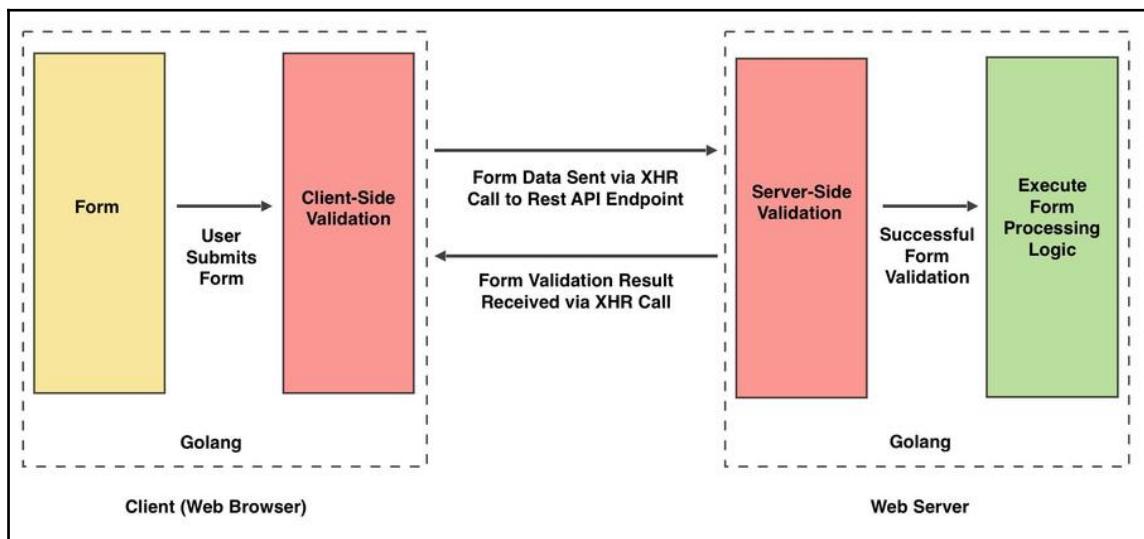


Figure 7.2: The web form validated on both the client-side and the server-side

# Designing the contact form

The contact form will allow website users to get in touch with the IGWEB team. Successfully completing the contact form will result in a contact form submission containing the user-generated form data that will be persisted in the Redis database. *Figure 7.3* is the wireframe image depicting the contact form:

The wireframe shows a 'Contact' form with the following fields:

- First Name: Text input field
- Last Name: Text input field
- E-mail Address: Text input field
- Message Area: Text area with placeholder text: "Enter your message for us here."
- Contact Button: A button labeled "Contact" at the bottom.

Figure 7.3: Wireframe design of the contact form

Figure 7.4 is the wireframe image depicting the contact form with form errors displayed when the user has not filled out the form properly:

The wireframe shows a 'Contact' form with the following layout:

- First Name:** An input field with an 'Error message' to its right.
- Last Name:** An input field with an 'Error message' to its right.
- E-mail Address:** An input field with an 'Error message' to its right.
- Message Area:** A large text area containing the placeholder text 'Enter your message for us here.' with an 'Error message' to its right.
- Contact Button:** A central button labeled 'Contact'.

Figure 7.4: Wireframe design of the contact form, with error messages displayed

Figure 7.5 is the wireframe image depicting the confirmation page that will be displayed to the user upon a successful contact form submission:



Figure 7.5: Wireframe design of the confirmation page

The contact form will solicit the following required information from the user: their first name, their last name, their email address, and their message to the team. If the user has not filled out any of these fields, upon hitting the **Contact** button on the form, the user will receive field-specific error messages, indicating the fields that have not been filled out.

## Implementing the templates

When rendering the **Contact** page from the server-side, we will use the `contact_page` template (found in the `shared/templates/contact_page.tpl` file):

```
 {{ define "pagecontent" }}  
 {{template "contact_content" . }}  
 {{end}}  
 {{template "layouts/webpage_layout" . }}
```

Recall that because we include the `layouts/webpage_layout` template, and this will print the markup that generates the `doctype`, `html`, and `body` tags of the page. This template will be used exclusively on the server-side.

Using the `define template` action, we demarcate the "pagecontent" block, where the content of the contact page will be rendered. The content of the contact page is defined inside the `contact_content` template (found in the `shared/template/contact_content.tpl` file):

```
<h1>Contact</h1>

{{template "partials/contactform_partial" .}}
```

Recall that in addition to the server-side application, the client-side application will be using the `contact_content` template to render the contact form in the primary content area.

Inside the `contact_content` template, we include the contact form partial template (`partials/contactform_partial`) that contains the markup for the contact form:

```
<div class="formContainer">
<form id="contactForm" name="contactForm" action="/contact" method="POST"
class="pure-form pure-form-aligned">
  <fieldset>
    {{if .Form }}
      <div class="pure-control-group">
        <label for="firstName">First Name</label>
        <input id="firstName" type="text" placeholder="First Name"
name="firstName" value="{{.Form.Fields.firstName}}">
        <span id="firstNameError" class="formError pure-form-message-
inline">{{.Form.Errors.firstName}}</span>
      </div>

      <div class="pure-control-group">
        <label for="lastName">Last Name</label>
        <input id="lastName" type="text" placeholder="Last Name"
name="lastName" value="{{.Form.Fields.lastName}}">
        <span id="lastNameError" class="formError pure-form-message-
inline">{{.Form.Errors.lastName}}</span>
      </div>

      <div class="pure-control-group">
        <label for="email">E-mail Address</label>
        <input id="email" type="text" placeholder="E-mail Address"
name="email" value="{{.Form.Fields.email}}">
        <span id="emailError" class="formError pure-form-message-
inline">{{.Form.Errors.email}}</span>
      </div>

    <fieldset class="pure-control-group">
      <textarea id="messageBody" class="pure-input-1-2" placeholder="Enter
```

```
your message for us here."
name="messageBody">>{{ .Form.Fields.messageBody }}</textarea>
>{{ .Form.Errors.messageBody }}</span>
</fieldset>

<div class="pure-controls">
  <input id="contactButton" name="contactButton" class="pure-button
pure-button-primary" type="submit" value="Contact" />
</div>
{{end}}
</fieldset>
</form>
</div>
```

This partial template contains the HTML markup necessary to implement the wireframe design depicted in *Figure 7.3*. The template actions that access the form field values and their corresponding errors are shown in bold. The reason that we populate the `value` attribute for a given `input` field is in case the user makes a mistake filling out the form, these values will be prepopulated with the values the user entered in the previous form submission attempt. There is a `<span>` tag directly after each `input` field, which will house the corresponding error message for that particular field.

The very last `<input>` tag is a `submit` button. By clicking this button, the user will be able to submit the form contents to the web server.

## Validating email address syntax

In addition to the basic requirement that all fields must be filled out, the email address field must be a properly formatted email address. If the user fails to provide a properly formatted email address, a field-specific error message will inform the user that the email address syntax is incorrect.

We'll be using the `EmailSyntax` function from the `validate` package found within the shared folder:

```
const EmailRegex = `(?i)^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\. [a-z0-9-]
]+)*(\. [a-z]{2,3})+$`
```

```
func EmailSyntax(email string) bool {
  validationResult := false
  r, err := regexp.Compile(EmailRegex)
  if err != nil {
```

```
    log.Fatal(err)
}
validationResult = r.MatchString(email)
return validationResult
}
```

Recall that because the `validate` package is strategically placed in the `shared` folder, the package is meant to be isomorphic (used across environments). The job of the `EmailSyntax` function is to determine if an input string is a valid email address or not. If the email address is valid, the function will return `true` or the function will return `false` if the input string isn't a valid email address.

## The form interface

An isomorphic web form implements the `Form` interface found in the `isokit` package:

```
type Form interface {
    Validate() bool
    Fields() map[string]string
    Errors() map[string]string
    FormParams() *FormParams
    PrefillFields()
    SetFields(fields map[string]string)
    SetErrors(errors map[string]string)
    SetFormParams(formParams *FormParams)
    SetPrefillFields(prefillFields []string)
}
```

The `Validate` method determines if the form has been filled out properly or not, returning a Boolean value of `true` if the form has been filled out properly, and it returns a Boolean value of `false` if the form hasn't been filled out properly.

The `Fields` method returns `map` of all the form fields where the key is the name of the form field, and the value is the string value of the form field.

The `Errors` method contains `map` of all the errors that were populated upon validation of the form. The key is the name of the form field, and the value is a descriptive error message.

The `FormParams` method returns the form's isomorphic form parameters object. The form parameters object is important because it determines the source from where user entered values for the form fields can be obtained. On the server side, form field values are obtained from `*http.Request` and on the client-side, form fields are obtained from the `FormElement` object.

Here's what the `FormParams` struct looks like:

```
type FormParams struct {
    FormElement *dom.HTMLFormElement
    ResponseWriter http.ResponseWriter
    Request *http.Request
    UseFormFieldsForValidation bool
    FormFields map[string]string
}
```

The `PrefillFields` method returns a string slice of all the names of the form fields, whose values should be retained in case the user makes a mistake while submitting the form.

The last four getter methods considered, `Fields`, `Errors`, `FormParams`, and `PrefillFields`, have corresponding setter methods, `SetFields`, `SetErrors`, `SetFormParams`, and `SetPrefillFields`, respectively.

## Implementing the contact form

Now that we know what the form interface looks like, let's start implementing the contact form. In our import grouping, note that we include the `validate` package and the `isokit` package:

```
import (
    "github.com/EngineerKamesh/igb/igweb/shared/validate"
    "github.com/isomorphicgo/isokit"
)
```

Recall that we need to import the `validate` package for the email address validation functionality using the `EmailSyntax` function defined in the package.

Most of the functionality needed to implement the `Form` interface that we covered earlier is provided by the `BasicForm` type, also found in the `isokit` package. We will type embed the type `BasicForm` into the type definition of our `ContactForm` struct:

```
type ContactForm struct {
    isokit.BasicForm
}
```

By doing so, most of the functionality to implement the `Form` interface is provided to us for free. It is imperative on us though to implement the `Validate` method because the default `Validate` method implementation, found in the `BasicForm` type, will always return `false`.

The constructor function for the contact form accepts a `FormParams` struct and will return a pointer to a newly created `ContactForm` struct:

```
func NewContactForm(formParams *isokit.FormParams) *ContactForm {
    prefillFields := []string{"firstName", "lastName", "email",
    "messageBody", "byDateInput"}
    fields := make(map[string]string)
    errors := make(map[string]string)
    c := &ContactForm{}
    c.SetPrefillFields(prefillFields)
    c.SetFields(fields)
    c.SetErrors(errors)
    c.SetFormParams(formParams)
    return c
}
```

We create a string slice, containing the names of the fields that should have their values retained, in the `prefillFields` variable. We create instances of type `map[string]string` for both the `fields` variable and the `errors` variable. We create a reference to a new `ContactForm` instance and assign it to the variable `c`. We call the `SetFields` method of the `ContactForm` instance, `c`, and pass the `fields` variable.

We call the `SetFields` and `SetErrors` methods and pass in `fields` and `errors` variables, respectively. We call the `SetFormParams` method of `c` to set the form parameters, which were passed into the constructor function. Finally, we return the new `ContactForm` instance.

As noted earlier, the default `Validate` method implementation, found in the `BasicForm` type, will always return `false`. Because we are implementing our own custom form, the contact form, it is our responsibility to define what a successful validation is, and we do so by implementing the `Validate` method:

```
func (c *ContactForm) Validate() bool {
    c.RegenerateErrors()
    c.PopulateFields()

    // Check if first name was filled out
    if isokit.FormValue(c.FormParams(), "firstName") == "" {
        c.SetError("firstName", "The first name field is required.")
    }

    // Check if last name was filled out
    if isokit.FormValue(c.FormParams(), "lastName") == "" {
        c.SetError("lastName", "The last name field is required.")
    }
}
```

```
// Check if message body was filled out
if isokit.FormValue(c.FormParams(), "messageBody") == "" {
    c.SetError("messageBody", "The message area must be filled.")
}

// Check if e-mail address was filled out
if isokit.FormValue(c.FormParams(), "email") == "" {
    c.SetError("email", "The e-mail address field is required.")
} else if validate.EmailSyntax(isokit.FormValue(c.FormParams(), "email"))
== false {
    // Check e-mail address syntax
    c.SetError("email", "The e-mail address entered has an improper
syntax.")

}

if len(c.Errors()) > 0 {
    return false
} else {
    return true
}
}
```

We first call the `RegenerateErrors` method to clear the current errors that are displayed to the user. The functionality for this method is only applicable to the client-side application. We will cover this method in more detail when we implement the contact form functionality on the client side.

We call the `PopulateFields` method to populate the `fields` map of the `ContactForm` instance. In case the user had made mistakes filling out the form, this method is responsible for prefilling the values that the user had already entered, to save them the trouble of having to enter those values again to resubmit the form.

At this point, we can commence with the form validation. We first check to see whether the first name field has been filled out by the user. We use the `FormValue` function, found in the `isokit` package, to obtain the user entered value for the form field having the name `firstName`. The first argument we pass to the `FormValue` function is the contact form's form parameters object, and the second value is the name of the form field whose value we wish to obtain, in this case, that is the form field with the name `"firstName"`. By checking to see whether the user-entered value is an empty string, we can determine whether or not the user has entered a value into the field. If they haven't, we call the `SetError` method, passing the name of the form field, along with a descriptive error message.

We perform the exact same check, to see if the user has filled out the necessary values, for the last name field, the message body, and the email address. If they haven't filled out any of these fields, we make calls to the `SetError` method, providing the name of the field and a descriptive error message.

In the case of the email address, if the user has entered a value for the email form field, we perform an additional check on the syntax of the email address supplied by the user. We pass the email value entered by the user to the `EmailSyntax` function in the `validate` package. If the email is not a valid syntax, we call the `SetError` method, passing in the form field name "`email`", along with a descriptive error message.

As we stated earlier, the `Validate` function returns a Boolean value based off of whether the form contains errors or not. We use the `if` conditional to determine If the count of errors is greater than zero, and if it is, that indicates that the form has errors, and we return a Boolean value of `false`. If count of errors is zero, the flow of control will reach the `else` block where we return a Boolean value of `true`.

Now that we've added the contact form, it's time to implement the server-side route handlers.

## Registering the contact route

We start out by adding the routes for the contact form page and the contact confirmation page:

```
r.Handle("/contact", handlers.ContactHandler(env)).Methods("GET", "POST")
r.Handle("/contact-confirmation",
  handlers.ContactConfirmationHandler(env)).Methods("GET")
```

Note that the `/contact` route that we have registered, which will be handled by the `ContactHandler` function, will accept HTTP requests using both the `GET` and the `POST` method. When the contact form is first accessed, it will be through a `GET` request to the `/contact` route. When the user submits the contact form, they will initiate a `POST` request to the `/contact` route. This explains why this route accepts both of these HTTP methods.

Upon successfully filling out the contact form, the user will be redirected to the `/contact-confirmation` route. This is done intentionally to avoid resubmission form errors that can occur, when the user attempts to refresh the web page, if we had simply printed out a form confirmation message using the `/contact` route itself.

## The contact route handler

The ContactHandler is responsible for rendering the contact page on IGWEB, where the contact form will reside:

```
func ContactHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
```

We declare and initialize the `formParams` variable to a newly initialized `FormParams` instance, providing the values for the `ResponseWriter` and `Request` fields:

```
    formParams := isokit.FormParams{ResponseWriter: w, Request: r}
```

We then declare and initialize the `contactForm` variable, with a newly created `ContactForm` instance, by calling the `NewContactForm` function and passing in the reference to the `formParams` struct:

```
    contactForm := forms.NewContactForm(&formParams)
```

We switch on the type of HTTP request method:

```
    switch r.Method {
        case "GET":
            DisplayContactForm(env, contactForm)
        case "POST":
            validationResult := contactForm.Validate()
            if validationResult == true {
                submissions.ProcessContactForm(env, contactForm)
                DisplayConfirmation(env, w, r)
            } else {
                DisplayContactForm(env, contactForm)
            }
        default:
            DisplayContactForm(env, contactForm)
    }
}
```

In the case that the HTTP request method is `GET`, we call the `DisplayContactForm` function, passing in the `env` object and the `contactForm` object. The `DisplayContactForm` function will render the contact form on the contact page.

In the case that the HTTP request method is a `POST`, we validate the contact form. Remember that if the `/contact` route is accessed using the `POST` method, it is indicative of the user having submitted the contact form to the route. We declare and initialize the `validationResult` variable, setting it to the value of the result from calling the `Validate` method of the `ContactForm` object, `contactForm`.

If the value of the `validationResult` is `true`, the form validated successfully. We call the `ProcessContactForm` function in the `submissions` package, passing in the `env` object and the `ContactForm` object. The `ProcessContactForm` function is responsible for handling a successful contact form submission. We then call the `DisplayConfirmation` function, passing in the `env` object, `http.ResponseWriter`, `w`, and `*http.Request`, `r`.

If the value of the `validationResult` is `false`, flow of control goes inside the `else` block, and we call the `DisplayContactForm` function passing in the `env` object and the `ContactForm` object, `contactForm`. This will render the contact form again, and this time, the user will see error messages pertaining to the fields that were either not filled out or that were not filled out properly.

In the case that the HTTP request method is neither a `GET` or a `POST`, we reach the default condition and simply call the `DisplayContactForm` function to display the contact form.

Here's the `DisplayContactForm` function:

```
func DisplayContactForm(env *common.Env, contactForm *forms.ContactForm) {
    templateData := &templatedata.Contact{PageTitle: "Contact", Form:
    contactForm}
    env.TemplateSet.Render("contact_page", &isokit.RenderParams{Writer:
    contactForm.FormParams().ResponseWriter, Data: templateData})
}
```

The function takes in an `env` object and a `ContactForm` object as input arguments. We start out by declaring and initializing the variable `templateData`, which will serve as the data object that we will be feeding to the `contact_page` template. We create a new instance of a `templatedata.Contact` struct and populate its `PageTitle` field to `"Contact"`, and its `Form` field to the `ContactForm` object that was passed into the function.

Here's what the `Contact` struct from the `templatedata` package looks like:

```
type Contact struct {
    PageTitle string
    Form *forms.ContactForm
}
```

The `PageTitle` field represents the page title for the web page, and the `Form` field represents the `ContactForm` object.

We then call the `Render` method on the `env.TemplateSet` object, and we pass in the name of the template we wish to render, `contact_page`, along with the isomorphic template render parameters (`RenderParams`) object. We have assigned the `Writer` field, of the `RenderParams` object, with the `ResponseWriter` associated with the `ContactForm` object, and we have assigned the `Data` field with the `templateData` variable.

Here's the `DisplayConfirmation` function:

```
func DisplayConfirmation(env *common.Env, w http.ResponseWriter, r *http.Request) {
    http.Redirect(w, r, "/contact-confirmation", 302)
}
```

This function is responsible for performing the redirect to the confirmation page. In this function, we simply call the `Redirect` function available in the `http` package and perform a 302 status redirect to the `/contact-confirmation` route.

Now that we've covered the route handler for the `Contact` page, it's time to take a look at the route handler for the contact form confirmation web page.

## The contact confirmation route handler

The sole purpose of the `ContactConfirmationHandler` function is to render the contact confirmation page:

```
func ContactConfirmationHandler(env *common.Env) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        env.TemplateSet.Render("contact_confirmation_page",
            &isokit.RenderParams{Writer: w, Data: nil})
    })
}
```

We call the `Render` method of the `TemplateSet` object and specify that want to render the `contact_confirmation_page` template, along with the passed in `RenderParams` struct. We have populated the `Writer` field of the struct with the `http.ResponseWriter`, and we have assigned a value of `nil` to the `Data` object, to indicate that there is no data object that is to be passed to the template.

## Processing the contact form submission

Upon successful completion of the contact form, we call the `ProcessContactForm` function in the `submission` package. If the workflow to fill out the contact form were like playing baseball, the call to the `ProcessContactForm` function can be considered reaching home plate and scoring a run. As we shall see later in the section, *Contact form Rest API endpoint*, this function will also be called by the contact form's Rest API endpoint. Now that we have established the significance of this function, let's go ahead and examine it:

```
func ProcessContactForm(env *common.Env, form *forms.ContactForm) {  
  
    log.Println("Successfully reached process content form function,  
    indicating that the contact form was filled out properly resulting in a  
    positive validation.")  
  
    contactRequest := &models.ContactRequest{FirstName:  
        form.GetFieldValue("firstName"), LastName: form.GetFieldValue("lastName"),  
        Email: form.GetFieldValue("email"), Message:  
        form.GetFieldValue("messageBody")}  
  
    env.DB.CreateContactRequest(contactRequest)  
}
```

We first print out a log message to indicate that we have successfully reached the function, indicating that the user has properly filled out the contact form, and the user-entered data is worthy to be processed. We then declare and initialize the `contactRequest` variable with a newly created `ContactRequest` instance.

The purpose of the `ContactRequest` struct is to model the data that is collected from the contact form. Here's what the `ContactRequest` struct looks like:

```
type ContactRequest struct {  
    FirstName string  
    LastName string  
    Email string  
    Message string  
}
```

As you can see, each field in the `ContactRequest` struct corresponds to the form field that exists in the contact form. We populate each field in the `ContactRequest` struct with its corresponding user-entered value from the contact form by calling the `GetFieldValue` method on the contact form object and providing the name of the form field.

As stated earlier, a successful contact form submission consists of storing the contact request information in the Redis database:

```
env.DB.CreateContactRequest(contactRequest)
```

We call the `CreateContactRequest` method of our custom Redis datastore object, `env.DB`, and pass in the `ContactRequest` object, `contactRequest` to the method. This method will save the contact request information into the Redis database:

```
func (r *RedisDatastore) CreateContactRequest(contactRequest
*models.ContactRequest) error {

    now := time.Now()
    nowFormatted := now.Format(time.RFC822Z)

    jsonData, err := json.Marshal(contactRequest)
    if err != nil {
        return err
    }

    if r.Cmd("SET", "contact-request|"+contactRequest.Email+"|"+nowFormatted,
    string(jsonData)).Err != nil {
        return errors.New("Failed to execute Redis SET command")
    }

    return nil
}
```

The `CreateContactRequest` method accepts a `ContactRequest` object as a sole input argument. We JSON marshal the `ContactRequest` value and store it into the Redis database. An error object is returned if either the JSON marshaling process failed or if saving to the database failed. If there were no errors encountered, we return `nil`.

## The accessible contact form

At this point, we have everything in place to take the contact form for a test drive. However, instead of opening up the contact form in a GUI-based web browser, we're first going to see how accessible the contact form is for visually impaired users using the Lynx web browser.

On first impression, it may seem strange that we are test driving the contact form using a 25-year-old, text-only web browser. However, Lynx has the capability to provide a refreshable braille display, along with text-to-speech functionality, which has made it a commendable web browsing technology for the visually impaired. Because Lynx does not support displaying images and running JavaScript, we can get a good idea of how the contact form will hold up for users that need greater accessibility.

If you are using Homebrew on your Mac, you can easily install Lynx like so:

```
$ brew install lynx
```

If you are using Ubuntu, you can install Lynx by issuing the following command:

```
$ sudo apt-get install lynx
```

If you are using Windows, you can download Lynx from this web page: <http://lynx.invisible-island.net/lynx2.8.8/index.html>.



You can read more about the Lynx web browser on Wikipedia at [https://en.wikipedia.org/wiki/Lynx\\_\(web\\_browser\)](https://en.wikipedia.org/wiki/Lynx_(web_browser)).

With the `igweb` web server instance running, we start up `lynx` using the `--nocolor` option like so:

```
$ lynx --nocolor localhost:8080/contact
```

Figure 7.6 shows what the contact form looks like in the Lynx web browser:

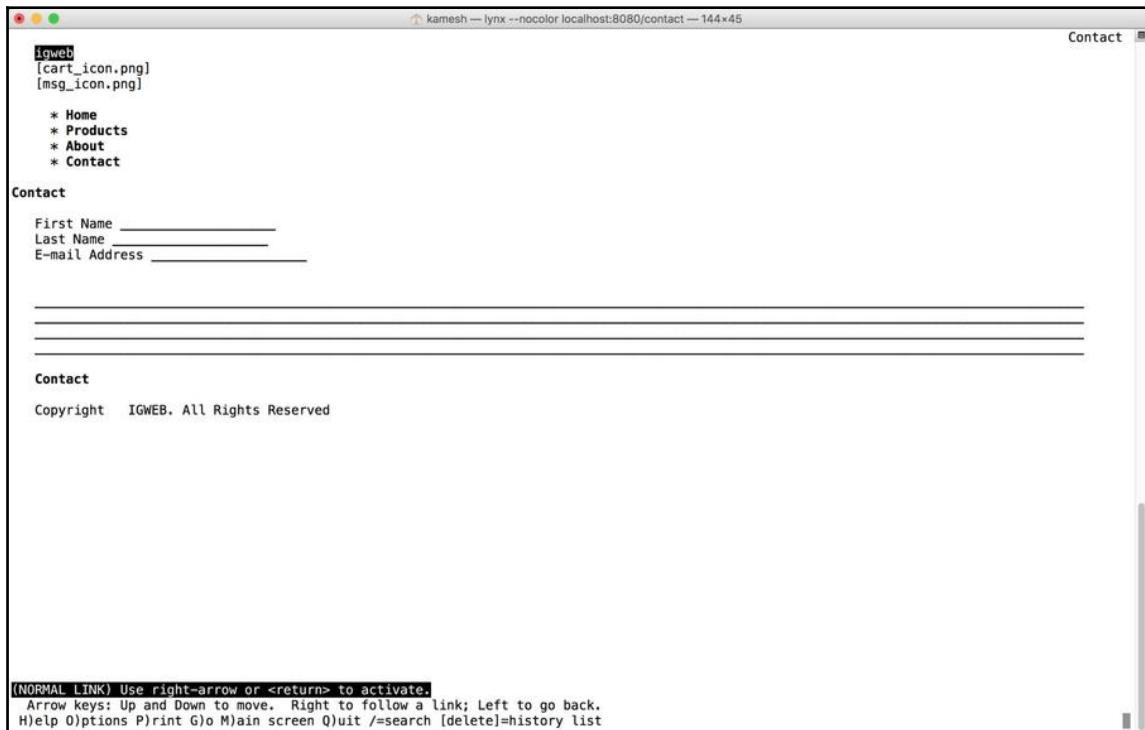


Figure 7.6: The contact form in the Lynx web browser

Now, we are going to partially fill out the contact form, on purpose to test if the form validation logic is working. In the case of the email field, we will provide an improperly formatted email address, as shown in *Figure 7.7*:

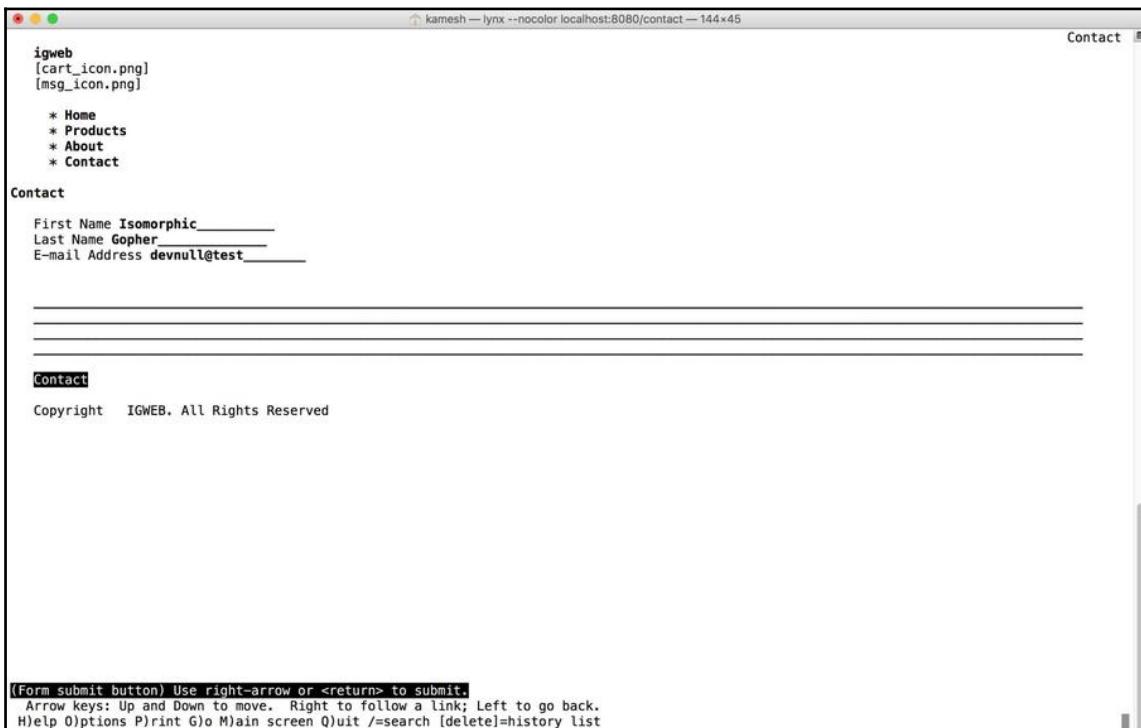


Figure 7.7: The contact form filled out improperly

Upon hitting the **Contact** button, note that we get error messages that pertain to the fields that have not been filled out properly, as shown in *Figure 7.8*:

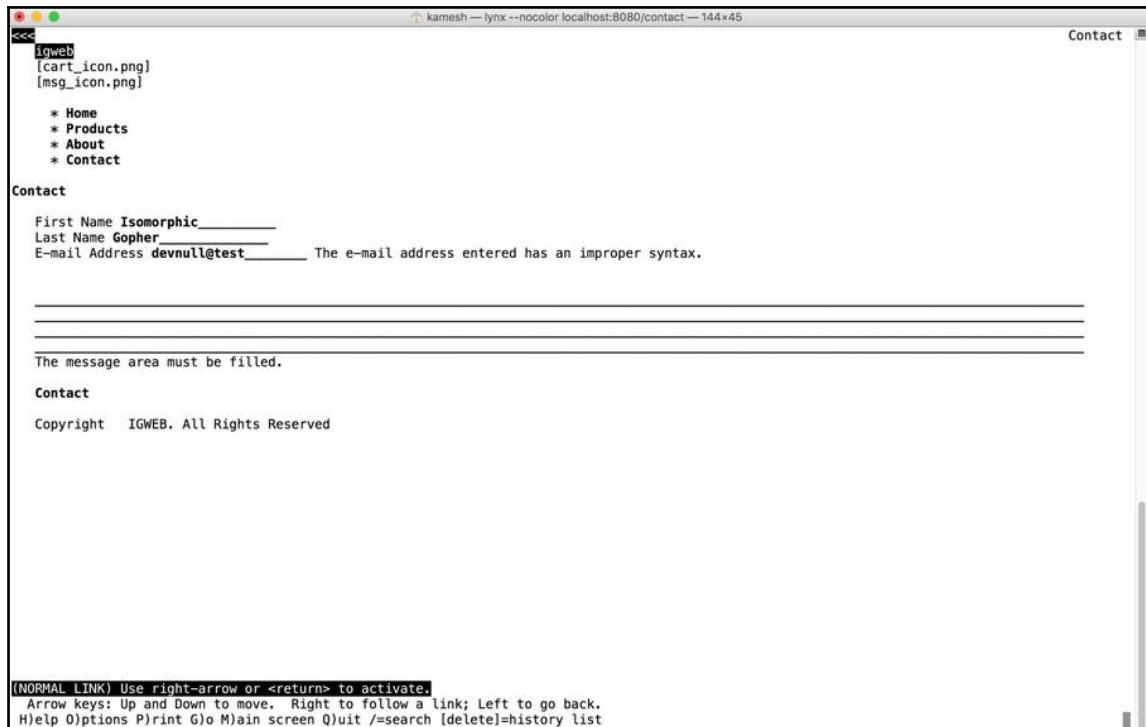


Figure 7.8: Error messages are displayed for the email address field and the message text area

Also note that we received the error message telling us that the email address format is not correct.

Figure 7.9 shows what the contact form looks like after we have corrected all the errors:

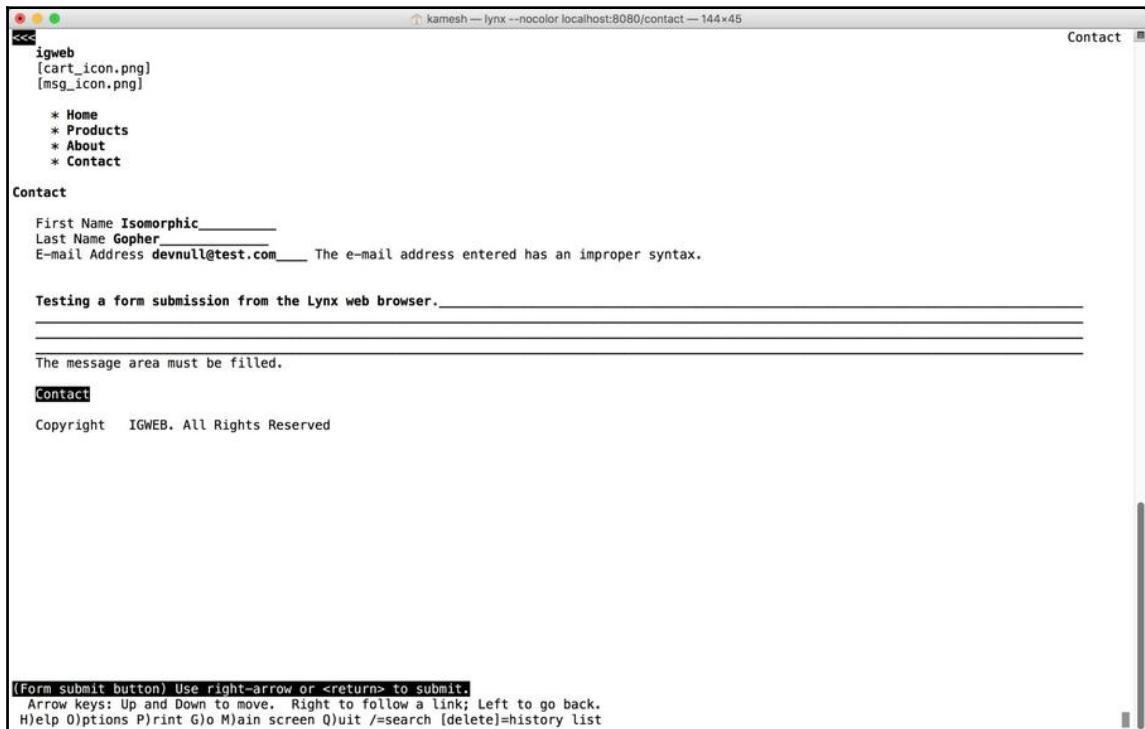


Figure 7.9: The contact form filled out properly

Upon submitting the corrected contact form, we see the confirmation message, informing us that we have successfully filled out the contact form, as shown in *Figure 7.10*:



Figure 7.10: The confirmation page

Inspecting the Redis database, using the **redis-cli** command, we can verify that we received the form submission, as shown in *Figure 7.11*:



```
127.0.0.1:6379> KEYS *devnull@test.com*
1) "contact-request|devnull@test.com|07 Nov 17 13:23 -0800"
127.0.0.1:6379> GET "contact-request|devnull@test.com|07 Nov 17 13:23 -0800"
"{"FirstName": "Isomorphic", "LastName": "Gopher", "Email": "devnull@test.com", "Message": "Testing a form submission from the Lynx web browser."}"
127.0.0.1:6379>
```

Figure 7.11: Verification of a newly stored contact request entry in the Redis database

At this point, we can be satisfied knowing that we've made our contact form accessible for visually impaired users and it didn't take much effort on our part. Let's take a look at how the contact form looks in a GUI-based web browser with JavaScript disabled.

## The contact form can function without JavaScript

In the Safari web browser, we may disable JavaScript, by choosing the **Disable JavaScript** option in Safari's **Develop** menu:

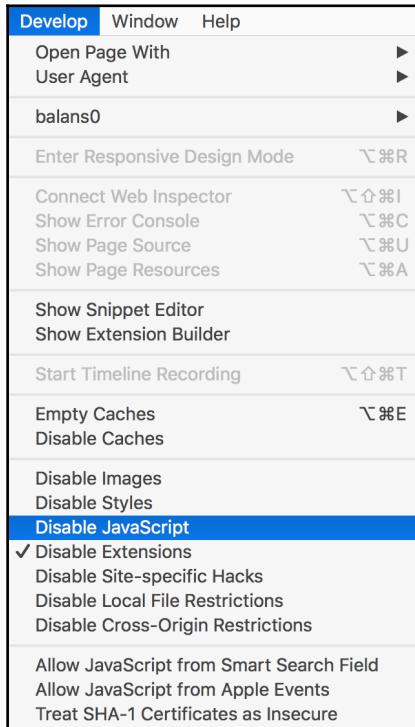
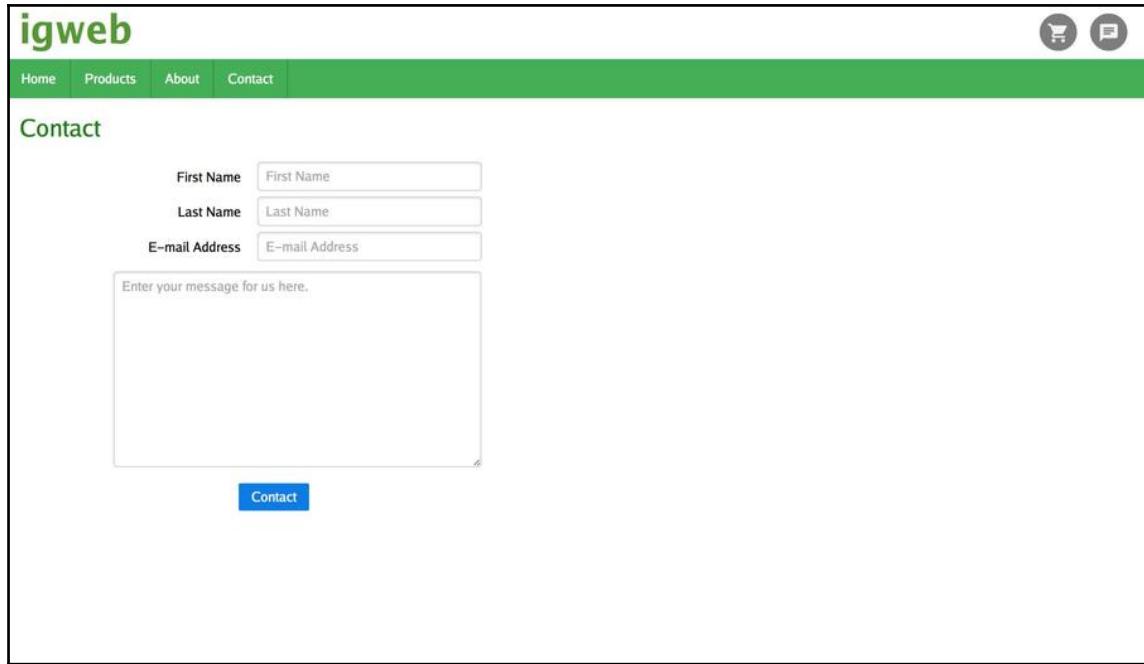


Figure 7.12: Disabling JavaScript using Safari's develop menu

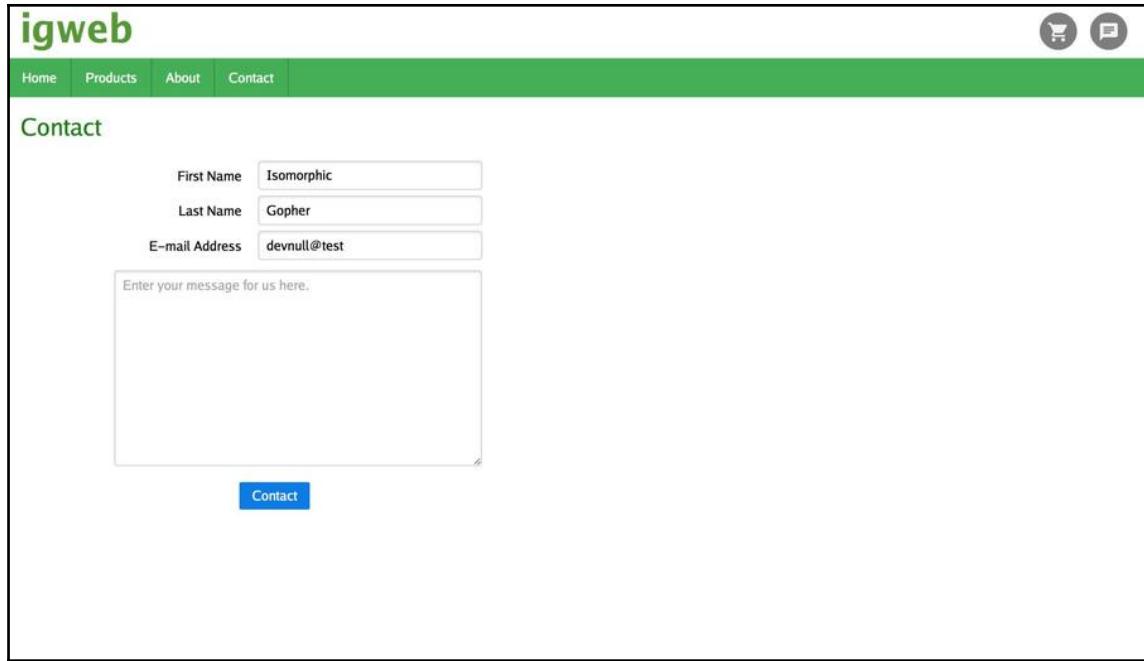
Figure 7.13 shows what the contact form looks like in the **Graphical User Interface (GUI)-based web browser**:



The screenshot shows a contact form within a web browser window. The browser's header includes the logo 'igweb' and two circular icons. The main navigation bar at the top has links for 'Home', 'Products', 'About', and 'Contact'. The 'Contact' link is highlighted with a blue border. The page title 'Contact' is displayed in a green header. Below the title are three input fields: 'First Name' (placeholder 'First Name'), 'Last Name' (placeholder 'Last Name'), and 'E-mail Address' (placeholder 'E-mail Address'). Below these fields is a large text area with the placeholder 'Enter your message for us here.' At the bottom of the form is a blue 'Contact' button.

Figure 7.13: The contact form in a GUI-based web browser

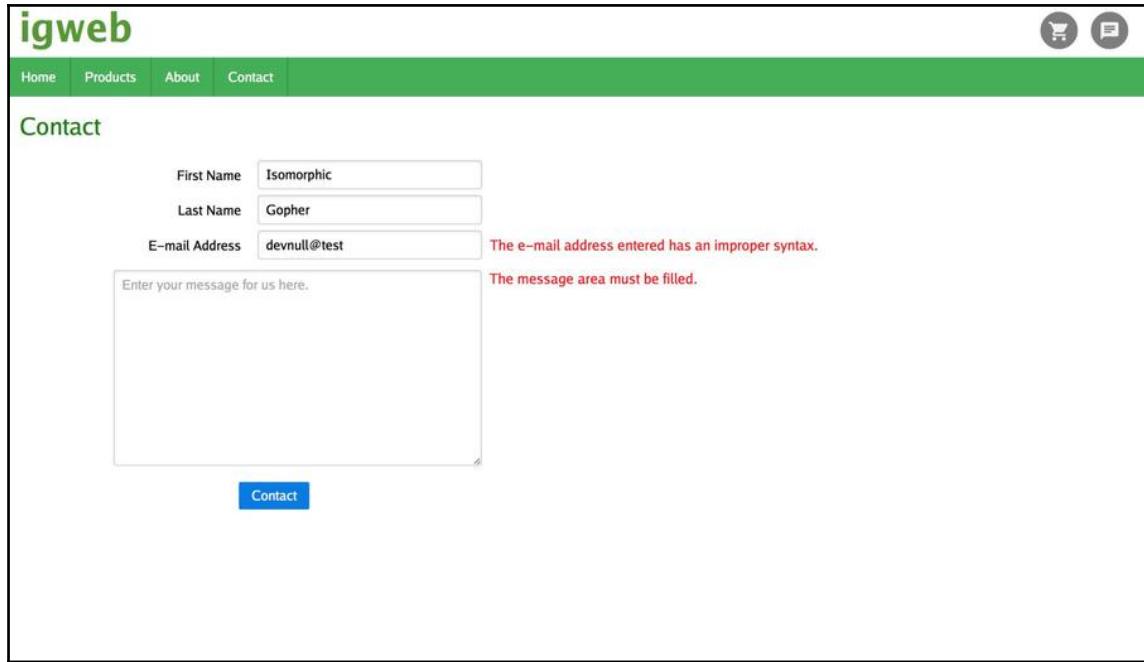
We follow the same testing strategy that we performed on the Lynx web browser. We partially fill out the form and supply an invalid email address, as shown in *Figure 7.14*:



The screenshot shows a contact form on a website. The header 'igweb' is visible, along with a navigation bar with links for Home, Products, About, and Contact. The 'Contact' link is highlighted. The main content area is titled 'Contact'. It contains three text input fields: 'First Name' with the value 'Isomorphic', 'Last Name' with the value 'Gopher', and 'E-mail Address' with the value 'devnull@test'. Below these fields is a large text area with the placeholder text 'Enter your message for us here.' At the bottom of the form is a blue 'Contact' button.

Figure 7.14: The contact form filled out improperly

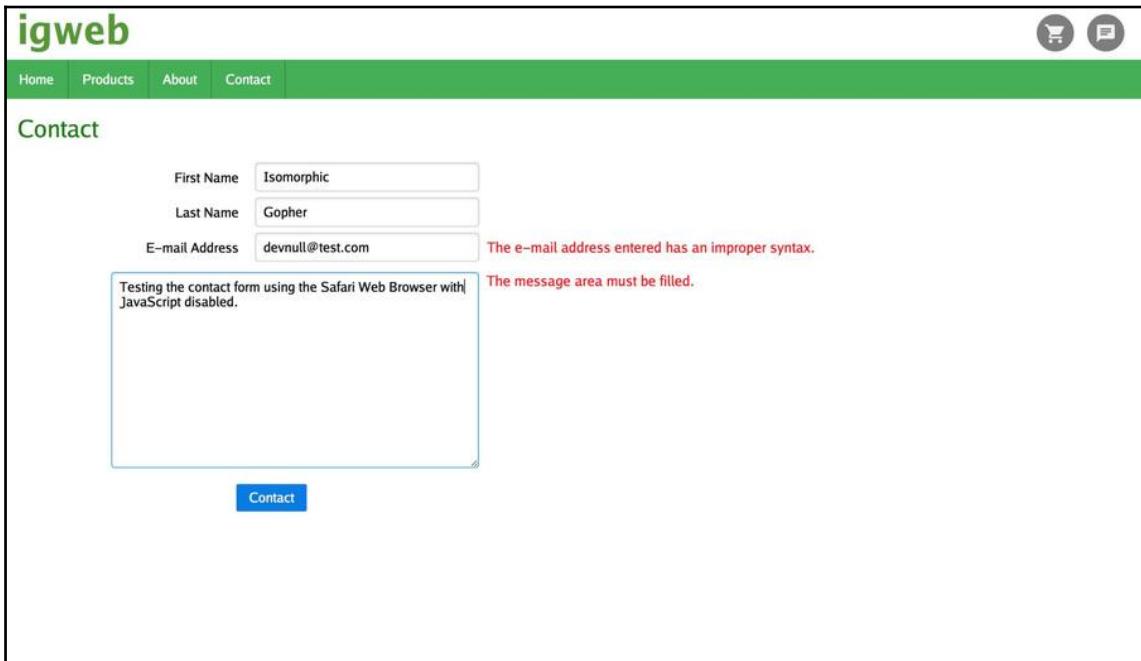
Upon hitting the **Contact** button, the error messages are displayed next to the fields with issues, as shown in *Figure 7.15*:



The screenshot shows a contact form on a website with a green header bar. The header bar contains the website name 'igweb' and navigation links for 'Home', 'Products', 'About', and 'Contact'. On the right side of the header are two circular icons: a shopping cart and a mail icon. The main content area has a white background. It features a 'Contact' heading. Below the heading are three input fields: 'First Name' (value: 'Isomorphic'), 'Last Name' (value: 'Gopher'), and 'E-mail Address' (value: 'devnull@test'). To the right of the 'E-mail Address' field is a red error message: 'The e-mail address entered has an improper syntax.' Below these fields is a large text area labeled 'Enter your message for us here.' To the right of this text area is another red error message: 'The message area must be filled.' At the bottom of the form is a blue 'Contact' button.

Figure 7.15: Error messages are displayed next to fields with issues

Upon submitting the contact form, note that we get errors pertaining to the improperly filled out fields. After correcting the errors, we are now ready to hit the **Contact** button to submit the form again, as shown in *Figure 7.16*:



The screenshot shows a contact form on a website with a green header bar. The header bar includes the logo 'igweb', a shopping cart icon, and a message icon. Below the header is a navigation bar with links for 'Home', 'Products', 'About', and 'Contact'. The main content area is titled 'Contact'. The form has three fields: 'First Name' (filled with 'Isomorphic'), 'Last Name' (filled with 'Gopher'), and 'E-mail Address' (filled with 'devnull@test.com'). Below the form, a message area contains the text: 'Testing the contact form using the Safari Web Browser with JavaScript disabled.' To the right of the message area, two validation errors are displayed: 'The e-mail address entered has an improper syntax.' and 'The message area must be filled.' At the bottom of the form is a blue 'Contact' button.

Figure 7.16: The properly filled out contact form ready for resubmission

Upon submitting the contact form, we get forwarded to the `/contact-confirmation` route, and we receive the confirmation message that the contact form has been filled out properly, as shown in *Figure 7.17*:

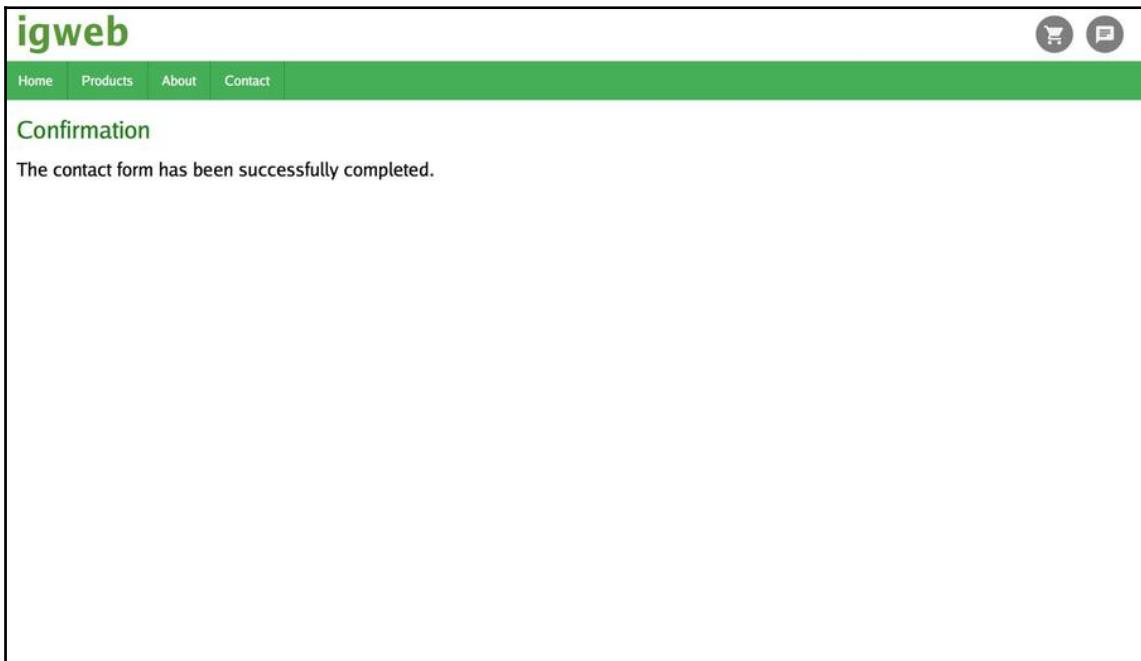


Figure 7.17: The confirmation page

The server-side-based contact form that we have implemented will continue to function even with JavaScript enabled. You might be wondering why do we need to implement the contact form on the client-side? Couldn't we just solely use the server-side-based contact form and call it a day?

The answer boils down to providing the user with an enhanced user experience. By solely using the server-side contact form, we break the single page application architecture that the user is experiencing. The astute reader will recognize that it takes a full page reload to submit the form and resubmit the form if there were errors. The HTTP redirect to the `/contact-confirmation` route will also break the user experience because it will also cause a full page reload.

The following two objectives need to be fulfilled, in order to implement the contact form on the client-side:

- Provide a consistent, seamless single-page application experience
- Provide the ability to validate the contact form on the client-side

The first objective, providing a consistent, seamless single-page application experience, is easily done using the isomorphic template set to render content to the primary content area `div` container as we had shown in previous chapters.

The second objective, the ability to validate the contact form on the client side, is possible, since the web browser has JavaScript enabled. With this capability, we can validate the contact form on the client side itself. Consider the scenario, where we have a user, that keeps making mistakes while filling out the contact form. We can lessen the amount of unnecessary network calls that are made to the web server. Only after the user has gotten past the first round of validation (on the client-side), will the form be submitted over the network, to the web server, where it undergoes the final round of validation (on the server side).

## Client-side considerations

Amazingly enough, there isn't much work that we need to perform to get the contact form going on the client side. Let's examine the `contact.go` source file found in the `client/handlers` folder, section by section:

```
func ContactHandler(env *common.Env) isokit.Handler {
    return isokit.HandlerFunc(func(ctx context.Context) {
        contactForm := forms.NewContactForm(nil)
        DisplayContactForm(env, contactForm)
    })
}
```

This is our `ContactHandler` function, which will service the needs of the `/contact` route on the client side. We start off by declaring and initializing the `contactForm` variable, assigning it to the `ContactForm` instance that is returned by calling the `NewContactForm` constructor function.

Note that we pass `nil` to the constructor function, when we should normally be passing a `FormParams` struct. On the client side, we would populate the `FormElement` field of the `FormParams` struct to associate the form element on the web page to the `contactForm` object. However, prior to rendering the web page, we run into a *did the chicken come before the egg* scenario. We can't populate the `FormElement` field of the `FormParams` struct because a form element doesn't exist on the web page yet. So, our first order of business is to render the contact form, and for the time being, we will set the contact form's `FormParams` struct to `nil` in order to do so. Later on, we will set the `contactForm` object's `FormParams` struct using the `contactForm` object's `SetFormParams` method.

To display the contact form on the web page, we call the `DisplayContactForm` function passing in the `env` object and the `contactForm` object, `contactForm`. This function is instrumental in our first objective to preserve the seamless single-page application user experience. Here's what the `DisplayContactForm` function looks like:

```
func DisplayContactForm(env *common.Env, contactForm *forms.ContactForm) {
    templateData := &templatedata.Contact{PageTitle: "Contact", Form:
    contactForm}
    env.TemplateSet.Render("contact_content", &isokit.RenderParams{Data:
    templateData, Disposition: isokit.PlacementReplaceInnerContents, Element:
    env.PrimaryContent, PageTitle: templateData.PageTitle})
    InitializeContactPage(env, contactForm)
}
```

We declare and initialize the `templateData` variable, which will be the data object that we pass to the template. The `templateData` variable is assigned with a newly created `Contact` instance from the `templatedata` package, having a `PageTitle` property set to "Contact" and the `Form` property set to the `contactForm` object.

We call the `Render` method of the `env.TemplateSet` object and specify that we wish to render the "contact\_content" template. We also supply the isomorphic render parameters (`RenderParams`) to the `Render` method, setting the `Data` field equal to the `templateData` variable, and we set the `Disposition` field to `isokit.PlacementReplaceInnerContents`, which declares how we will render the template content relative to an associated element. By setting the `Element` field to `env.PrimaryContent`, we specify that the primary content `div` container will be the associated element that the template will be rendering to. Finally, we set the `PageTitle` property to dynamically change the web page's title as the user lands on the `/contact` route from the client side.

We call the `InitializeContactPage` function, supplying the `env` object and the `contactForm` object. Recall that the `InitializeContactPage` function is responsible for setting up the user interactivity-related code (event handlers) for the **Contact** page. Let's examine the `InitializeContactPage` function:

```
func InitializeContactPage(env *common.Env, contactForm *forms.ContactForm) {
    formElement :=
        env.Document.GetElementByID("contactForm").(*dom.HTMLFormElement)
        contactForm.SetFormParams(&isokit.FormParams{FormElement: formElement})
        contactButton :=
        env.Document.GetElementByID("contactButton").(*dom.HTMLInputElement)
        contactButton.AddEventListener("click", false, func(event dom.Event) {
            handleContactButtonClickEvent(env, event, contactForm)
        })
}
```

We call the `GetElementByID` method on the `env.Document` object to fetch the contact form element and assign it to the variable `formElement`. We call the `SetFormParams` method, supplying a `FormParams` struct and populating its `FormElement` field with the `formElement` variable. At this point, we have set the form parameters for the `contactForm` object. We obtain the contact form's button element by calling the `GetElementByID` method on the `env.Document` object and supplying an `id` of "contactButton".

We add an event listener, on the `click` event of the contact button, which will call the `handleContactButtonClickEvent` function and pass the `env` object, the `event` object, and the `contactForm` object. The `handleContactButtonClickEvent` function is significant because it will run the form validation on the client-side, and if the validation is successful, it will initiate an XHR call to a Rest API endpoint on the server-side. Here's the code for the `handleContactButtonClickEvent` function:

```
func handleContactButtonClickEvent(env *common.Env, event dom.Event,
    contactForm *forms.ContactForm) {
    event.PreventDefault()
    clientSideValidationResult := contactForm.Validate()

    if clientSideValidationResult == true {
        contactFormErrorsChannel := make(chan map[string]string)
        go ContactFormSubmissionRequest(contactFormErrorsChannel, contactForm)
```

The first thing that we do is suppress the default behavior of clicking the **Contact** button, which will submit the entire web form. This default behavior stems from the fact that the contact button element is an `input` element of type `submit`, whose default behavior when clicked is to submit the web form.

We then declare and initialize `clientSideValidationResult`, a Boolean variable, assigned to the result of calling the `Validate` method on the `contactForm` object. If the value of `clientSideValidationResult` is `false`, we reach the `else` block where we call the `DisplayErrors` method on the `contactForm` object. The `DisplayErrors` method is provided to us from the `BasicForm` type in the `isokit` package.

If the value of the `clientSideValidationResult` is `true`, that means the form validated properly on the client side. At this point, the contact form submission has cleared the first round of validation on the client-side.

To commence the second (and final) round of validation, we need to call the Rest API endpoint on the server-side, which is responsible for validating the contents of the form and rerun the same set of validations. We create a channel named `contactFormErrorsChannel`, which is channel that we'll send `map[string]string` values over. We call the `ContactFormSubmissionRequest` function as a goroutine, passing in the channel `contactFormErrorsChannel` and the `contactForm` object. The `ContactFormSubmissionRequest` function will initiate an XHR call to the server-side Rest API endpoint, to validate the contact form on the server-side. A map of errors will be sent over `contactFormErrorsChannel`.

Let's take a quick look at the `ContactFormSubmissionRequest` function before we return to the `handleContactButtonClickEvent` function:

```
func ContactFormSubmissionRequest(contactFormErrorsChannel chan
map[string]string, contactForm *forms.ContactForm) {

    jsonData, err := json.Marshal(contactForm.Fields())
    if err != nil {
        println("Encountered error: ", err)
        return
    }

    data, err := xhr.Send("POST", "/restapi/contact-form", jsonData)
    if err != nil {
        println("Encountered error: ", err)
        return
    }

    var contactFormErrors map[string]string
```

```
    json.NewDecoder(strings.NewReader(string(data))).Decode(&contactFormErrors)

    contactFormErrorsChannel <- contactFormErrors
}
```

In the `ContactFormSubmissionRequest` function, we JSON marshal the fields of the `contactForm` object and fire an XHR call to the web server by calling the `Send` function from the `xhr` package. We specify that the XHR call will be using the `POST` HTTP Method and will be posting to the `/restapi/contact-form` endpoint. We pass in the JSON-encoded data of the contact form fields as the final argument to the `Send` function.

If there were no errors in either the JSON marshaling process, or while making the XHR call, we obtain the data retrieved from the server and attempt to decode it from JSON format into the `contactFormErrors` variable. We then send the `contactFormErrors` variable over the channel, `contactFormErrorsChannel`.

Now, let's return back to the `handleContactButtonClickEvent` function:

```
go func() {
    serverContactFormErrors := <-contactFormErrorsChannel
    serverSideValidationResult := len(serverContactFormErrors) == 0

    if serverSideValidationResult == true {
        env.TemplateSet.Render("contact_confirmation_content",
        &isokit.RenderParams{Data: nil, Disposition:
        isokit.PlacementReplaceInnerContents, Element: env.PrimaryContent})
    } else {
        contactForm.SetErrors(serverContactFormErrors)
        contactForm.DisplayErrors()
    }
}()

} else {
    contactForm.DisplayErrors()
}
}
```

To prevent blocking within the event handler, we create and run an anonymous goroutine function. We receive the map of errors into the `serverContactFormErrors` variable, from `contactFormErrorsChannel`. The `serverSideValidationResult` Boolean variable is responsible for determining if there were errors in the contact form by examining the length of the errors map. If the length of the errors is zero that indicates there were no errors in the contact form submission. If the length is greater than zero that indicates that errors are present in the contact form submission.

If the `serverSideValidationResult` Boolean variable has a value of `true`, we call the `Render` method on the isomorphic template set to render the `contact_confirmation_content` template and we pass in the isomorphic template render parameters. In the `RenderParams` object, we set the `Data` field to `nil` because we won't be passing in any data object to the template. We specify the value `isokit.PlacementReplaceInnerContents` for the `Disposition` field to indicate that we will be performing a replace inner HTML operation on the associated element. We set the `Element` field to the associated element, the primary content `div` container, since this is where the template will render to.

If the `serverSideValidationResult` Boolean variable has a value of `false`, that means the form still contains errors that need to be corrected. We call the `SetErrors` method on the `contactForm` object passing in the `serverContactFormErrors` variable. We then call the `DisplayErrors` method on the `contactForm` object to display the errors to the user.

We're just about done, the only item we have left to realizing the contact form on the client-side is implementing the server-side, Rest API endpoint that performs the second round of validation on the contact form submission.

## Contact form Rest API endpoint

Inside the `igweb.go` source file, we have registered the `/restapi/contact-form` endpoint and it's associated handler function, `ContactFormEndpoint`:

```
r.Handle("/restapi/contact-form",
  endpoints.ContactFormEndpoint(env)).Methods("POST")
```

The `ContactFormEndpoint` function is responsible for servicing the `/restapi/contact-form` endpoint:

```
func ContactFormEndpoint(env *common.Env) http.Handler {
  return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
```

```
var fields map[string]string

reqBody, err := ioutil.ReadAll(r.Body)
if err != nil {
    log.Println("Encountered error when attempting to read the request
body: ", err)
}

err = json.Unmarshal(reqBody, &fields)
if err != nil {
    log.Println("Encountered error when attempting to unmarshal json data:
", err)
}

formParams := isokit.FormParams{ResponseWriter: w, Request: r,
UseFormFieldsForValidation: true, FormFields: fields}
contactForm := forms.NewContactForm(&formParams)
validationResult := contactForm.Validate()

if validationResult == true {
    submissions.ProcessContactForm(env, contactForm)
}
w.Header().Set("Content-Type", "application/json")
json.NewEncoder(w).Encode(contactForm.Errors())
})
}
```

The purpose of this function is to provide server-side validation of the contact-form and return a JSON-encoded map of errors. We create a variable `fields` of type `map[string]string` that represents the fields in the contact form. We read the request body, which will contain the JSON-encoded fields map. We then unmarshal the JSON-encoded fields map into the `fields` variable.

We create a new `FormParams` instance and assign it to the variable `formParams`. In the `FormParams` struct, we specify the value of the `http.ResponseWriter`, `w`, for the `ResponseWriter` field, and the `*http.Request`, `r`, for the `Request` field. We set the `UseFormFieldsForValidation` field to `true`. Doing so will change the default behavior of fetching the form value for a particular field from the request, and instead the values for form fields will be obtained from the contact form's `formFields` map. Finally, we set the `FormFields` field to the `fields` variable, the map of fields that we JSON unmarshalled from the request body.

We create a new `contactForm` object by calling the `NewContactForm` function and passing in a reference to the `formParams` object. To perform the server-side validation, we simply call the `Validate` method on the `contactForm` object and assign the result of the method call to the `validationResult` variable. Keep in mind that the same validation code present on the client-side is also present on the server-side, and there's really nothing special we're doing here, except invoking the validation logic from the server-side where presumably it cannot be tampered with.

If the value of `validationResult` is `true`, that means the contact form has cleared the second round of form validation on the server-side, and we can call the `ProcessContactForm` function in the `submissions` package, passing in the `env` object and the `contactForm` object. Remember when it comes to successfully validating the contact form, calling the `ProcessContactForm` function means we've reached the home plate and scored a run.

If the value of the `validationResult` is `false`, there is nothing special that we have to do. The `Errors` field of the `contactForm` object would have been populated after making a call to the object's `Validate` method. If there were no errors, the `Errors` field would just be an empty map.

We send a header to the client to indicate that the server will be sending a JSON object response. We then encode the errors map of the `contactForm` object into its JSON representation and write it out to the client using `http.ResponseWriter`, `w`.

## Checking the client-side validation

We have everything in place now for the contact form's client-side validation. Let's open up the web browser with JavaScript enabled. Let's also have the web inspector open to check for network calls, as shown in *Figure 7.18*:

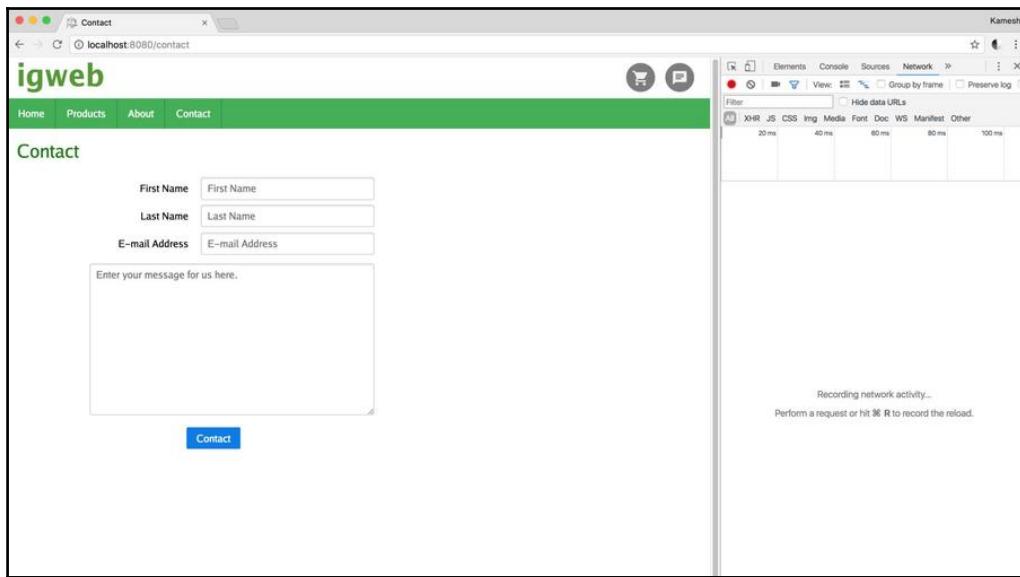


Figure 7.18: The contact form with the web inspector open

First, we will partially fill out the contact form, as shown in *Figure 7.19*:

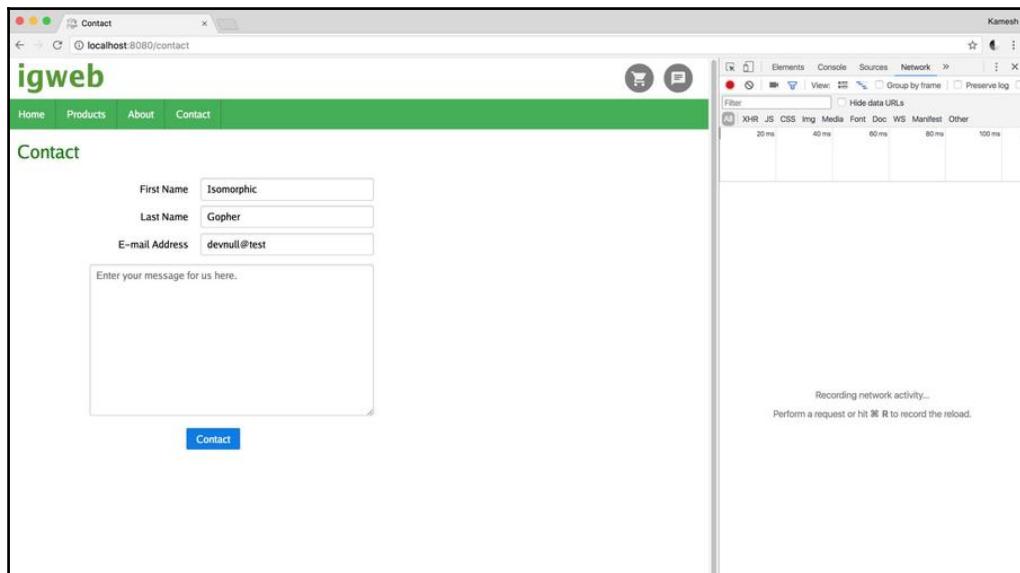


Figure 7.19: The contact form filled out improperly

Upon clicking on the **Contact** button, we will trigger the form validation errors on the client side, as shown in *Figure 7.20*. Note that as we do so, no network calls are made to the server, no matter how many times we click on the **Contact** button:

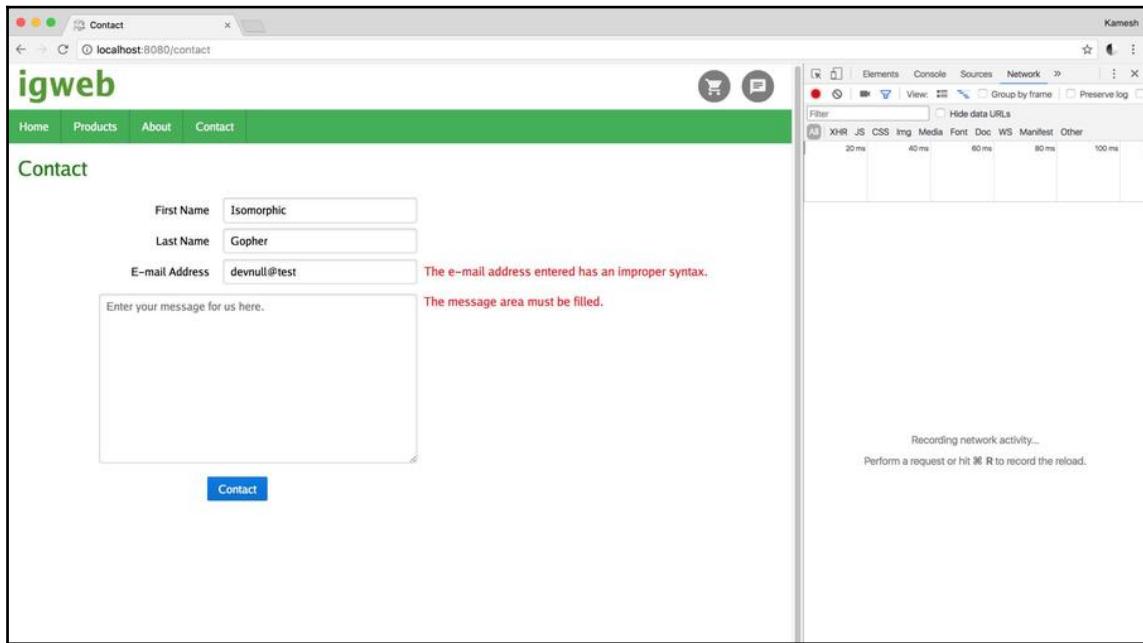
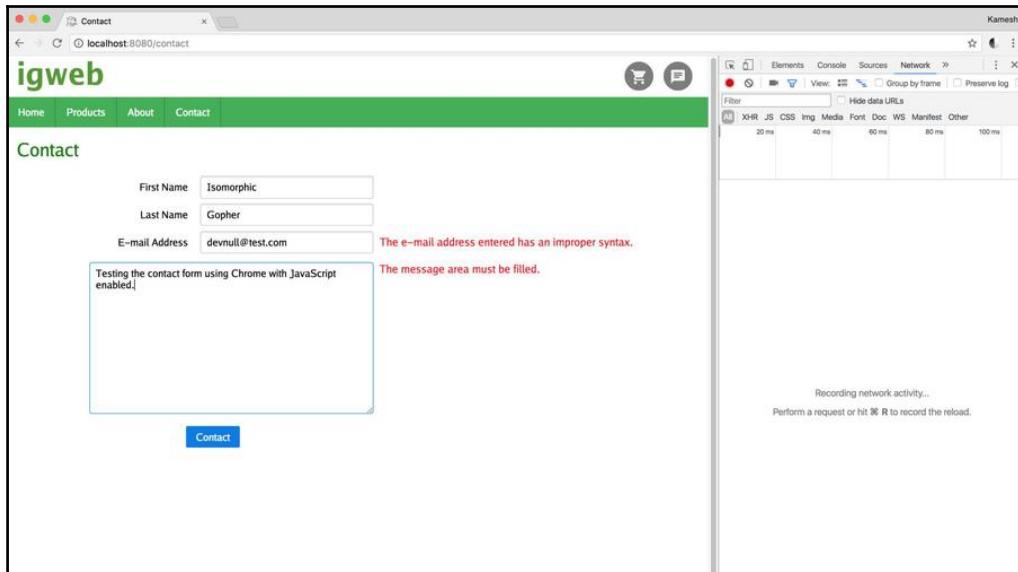


Figure 7.20: Error messages are displayed after performing client-side validation. Note that there are no network calls made to the server

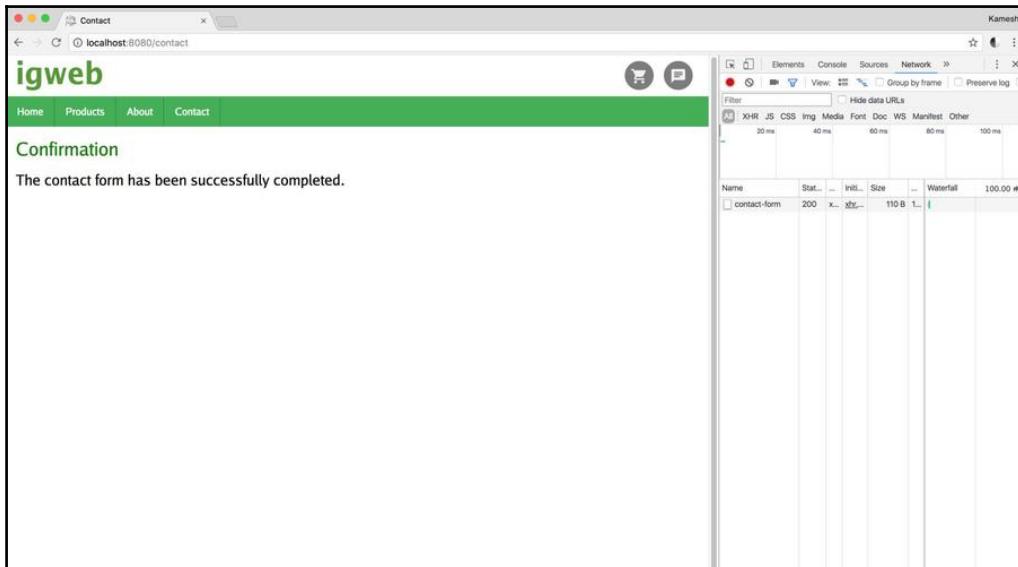
Now, let's correct the errors present in the contact form (as shown in *Figure 7.21*) and get ready to resubmit:



A screenshot of a web browser window for 'igweb' showing a contact form. The browser's developer tools are open, specifically the Network tab, which shows a single XHR request. The contact form fields are filled: First Name (Isomorphic), Last Name (Gopher), and E-mail Address (devnull@test.com). A validation error message 'The e-mail address entered has an improper syntax.' is displayed next to the email field. The message area contains the text 'Testing the contact form using Chrome with JavaScript enabled.' A red error message 'The message area must be filled.' is shown above the message area. A 'Contact' button is at the bottom. The developer tools Network tab shows a single XHR request with a status of 200 ms.

Figure 7.21: The contact form filled out properly and ready for resubmission

Upon resubmission of the form, we receive the confirmation message, as shown *Figure 7.22*:



A screenshot of a web browser window for 'igweb' showing a confirmation message. The developer tools Network tab shows an XHR call with a status of 200 ms. The confirmation message 'The contact form has been successfully completed.' is displayed on the page.

Figure 7.22: An XHR call is made containing the form data, and the confirmation message is rendered upon successful server-side form validation

Note that an XHR call was initiated to the web server, as shown in *Figure 7.23*. Examining the response of the call, we can see that the empty object ({} ) returned from the endpoint's response, indicates that the `errors` map is empty, signifying a successful form submission:

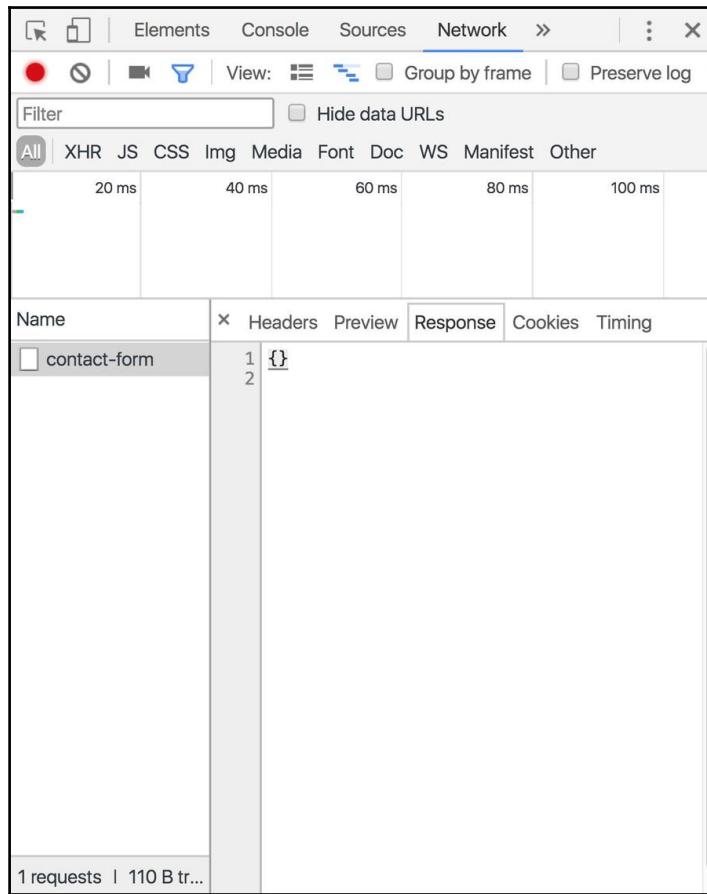


Figure 7.23: The XHR call responded with an empty errors map, indicating that the form successfully cleared the server-side form validation

Now that we have verified that the client-side validation logic is working on the contact form, we must emphasize a significant point that is important when accepting data from the client-side. The server must always hold the veto power, when it comes to validating user-entered data. The round two validation performed on the server-side should be a mandatory step. Let's take a look at why we always need server-side validation.

## Tampering with the client-side validation result

Let's consider the scenario, where we have a nefarious (and clever) user who knows how to short circuit our client-side validation logic. It's JavaScript after all, and it's running in the web browser. There's really nothing to stop a malicious user from throwing our client-side validation logic to the wind. To simulate such a tampering event, we simply need to assign the Boolean value of `true` to the `clientSideValidationResult` variable in the `contact.go` source file like so:

```
func handleContactButtonClickEvent(env *common.Env, event dom.Event,
contactForm *forms.ContactForm) {

    event.PreventDefault()
    clientSideValidationResult := contactForm.Validate()

    clientSideValidationResult = true

    if clientSideValidationResult == true {

        contactFormErrorsChannel := make(chan map[string]string)
        go ContactFormSubmissionRequest(contactFormErrorsChannel, contactForm)

        go func() {

            serverContactFormErrors := <-contactFormErrorsChannel
            serverSideValidationResult := len(serverContactFormErrors) == 0

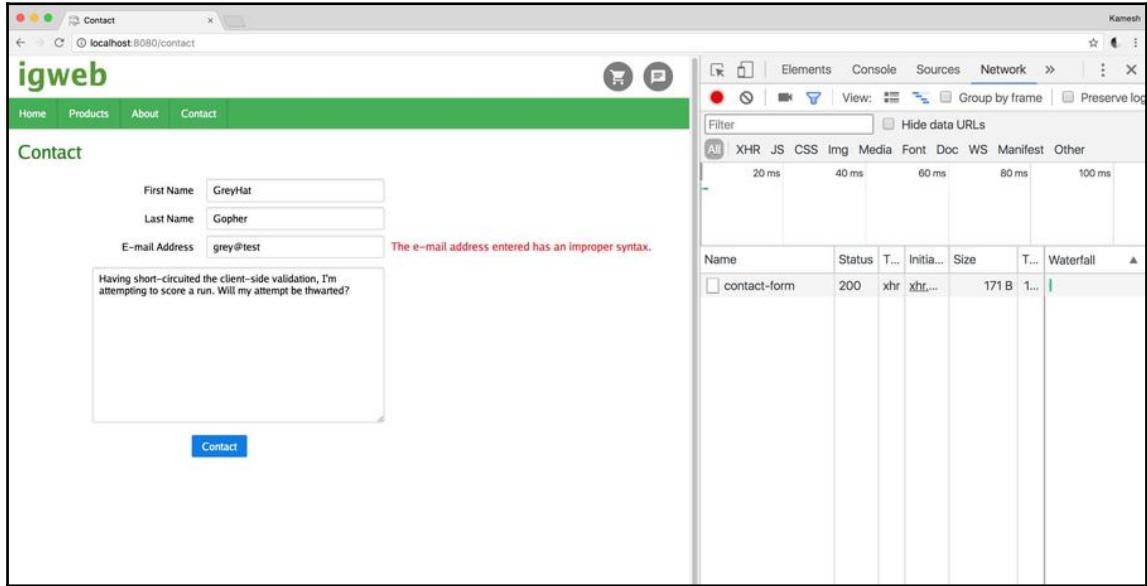
            if serverSideValidationResult == true {
                env.TemplateSet.Render("contact_confirmation_content",
&isokit.RenderParams{Data: nil, Disposition:
isokit.PlacementReplaceInnerContents, Element: env.PrimaryContent})
            } else {
                contactForm.SetErrors(serverContactFormErrors)
                contactForm.DisplayErrors()
            }
        }

    }()

} else {
    contactForm.DisplayErrors()
}
}
```

At this point, we have bypassed the real result of the client-side validation, and we are forcing the client-side web application to always green light the contact form validation performed on the client-side. If we were solely performing form validation on the client-side, this would put us in a very bad situation. This is exactly the reason why, we need the second round of validation on the server-side.

Let's open up the web browser and partially fill out the form again, as shown in *Figure 7.24*:



The screenshot shows a web browser window for a site called "igweb". The URL is "localhost:8080/contact". The browser's developer tools are open, showing the Network tab with a single XHR request named "contact-form" with a status of 200. The main content area shows a contact form with three fields: First Name (GreyHat), Last Name (Gopher), and E-mail Address (grey@test). The E-mail Address field has a red validation message: "The e-mail address entered has an improper syntax." Below the form, a text area contains the message: "Having short-circuited the client-side validation, I'm attempting to score a run. Will my attempt be thwarted?". At the bottom is a blue "Contact" button. The developer tools Network tab shows the XHR request "contact-form" with a status of 200, a type of "xhr", and a size of 171 B.

Figure 7.24: Even after disabling the client-side form validation, the server-side form validation prevents the improperly filled out contact form from being submitted

Note that this time, when the **Contact** button is clicked, the XHR call is initiated to the Rest API endpoint on the server-side, which returns map of errors in the contact form, as shown in *Figure 7.25*:

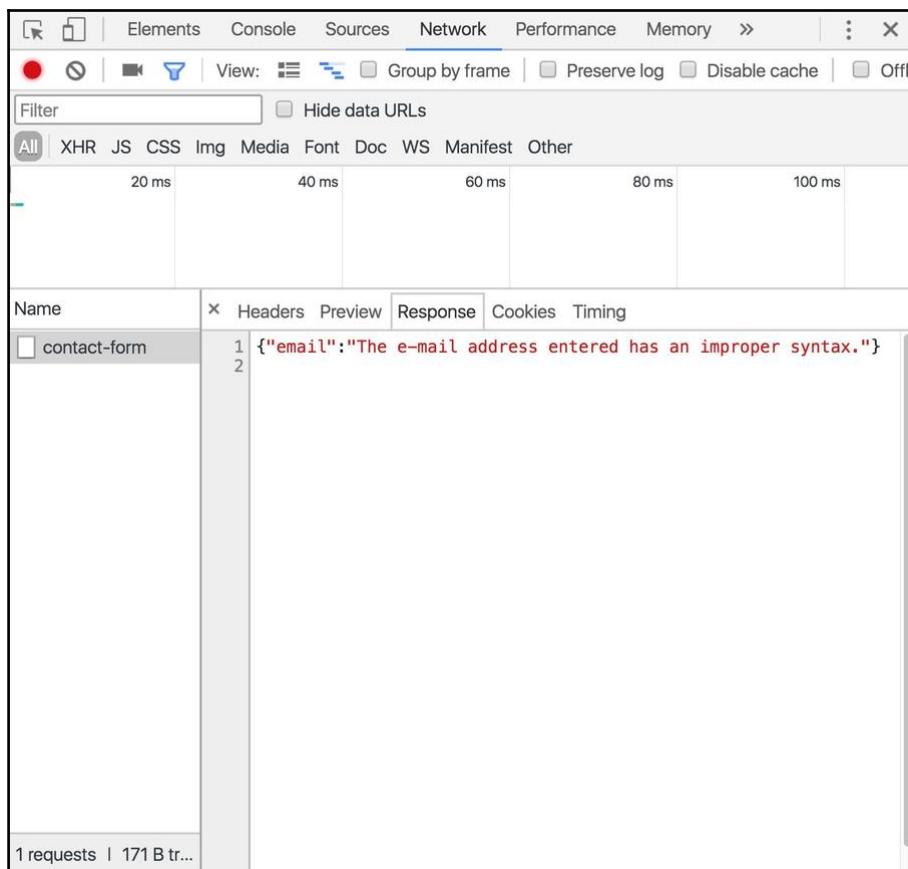


Figure 7.25: The errors map from the server response is populated with an error indicating that the value entered in the email address field has an improper syntax

The second round of validation, performed on the server-side, has kicked in, and it prevented the malicious user from being able to reach home plate and score a run. If the client-side validation is unable to properly function, that incomplete or incorrectly formatted form field will be caught by the server-side validation. This is a major reason, why you should always implement server-side form validation for your web forms.

## Summary

In this chapter, we demonstrated the process of building an accessible, isomorphic web form. First, we demonstrated the flow of the isomorphic web form in both the scenario where JavaScript was disabled and in the scenario where JavaScript was enabled.

We showed you how to create an isomorphic web form, which had the ability to share both form code as well as validation logic across environments. In the scenario where the form contained errors, we showed you how to display the errors to the user in a meaningful manner. The isomorphic web form created was quite robust and being able to function, both, in the scenario where JavaScript was either disabled in the web browser, or a JavaScript runtime didn't exist (such as the Lynx web browser), and in the scenario where JavaScript was enabled in the web browser.

We demonstrated testing the accessible, isomorphic web form using the Lynx web browser, to verify that the form would be available to users in need of greater accessibility. We also verified that the form functioned properly, even with JavaScript disabled, in a web browser that was equipped with a JavaScript runtime.

In the scenario that JavaScript was enabled in the web browser, we showed you how to validate the form on the client-side and submit the data to a Rest API endpoint after performing client-side validation. Even with the convenience and heightened capability of validating the form on the client-side, we emphasized the importance of always validating the form on the server-side, by demonstrating a scenario where the server-side form validation kicked in, even in the potential scenario that the client-side validation result was tampered with.

The interaction between the user and the contact form was fairly simple. The user had to fill out the form correctly in order to submit the data to the server, where ultimately the form data was processed. In the next chapter, we are going to go beyond this simple interaction and consider a scenario where the user and the web application engage in communication, in an almost conversation-like manner. In *Chapter 8, Real-time Web Application Functionality*, we will implement IGWEB's live chat feature, which allows the website user to engage in a simple question and answer conversation with a chatbot.

# 8

## Real-Time Web Application Functionality

In the previous chapter, we considered how we could validate and process user-generated data through a web form. When the user properly filled out the contact form, it successfully cleared two rounds of validation, and the user was presented with a confirmation message. Once the form had been submitted, the workflow was complete. What if we wanted to consider a more engaging workflow, one where the user could engage with the server-side application, perhaps, in a conversation-like manner?

The web of today is far different from the nascent web that Tim Berners-Lee devised in the early 1990s. Back then, the emphasis on the Web was to hyperlink connected documents. The HTTP transaction between the client and the server had always meant to be short-lived.

In the early 2000s, this started to change. Researchers demonstrated the means by which the server could maintain a persistent connection with the client. Early prototypes on the client side were created using Adobe Flash, one of the only technologies available at the time, to make a persistent connection between the web server and the web client.

In parallel to these early attempts, an era of inefficiency was born in the form of AJAX (XHR) long polling. The client would keep making calls to the server (similar to a heartbeat check), and check whether the state of something the client was interested in had changed. The server would return the same, tired response until the state that the client was interested in changed, which could be reported back to the client. The major inefficiency of this approach is the sheer amount of network calls that have to be made between the web client and the web server. Unfortunately, the inefficient practice of AJAX long polling became so popular, that it is still widely used by many websites today.

The idea behind real-time web application functionality is to provide a far greater user experience by providing information in near real-time. Keep in mind that with network latency and the limitations on signals imposed by the laws of physics, no communication is ever conducted in *real time* but, rather, in *near real time*.

The primary ingredient to implement real-time web application functionality is the WebSocket, a protocol allowing for bidirectional communication between the web server and the web client. Go makes for an ideal programming language to implement real-time web applications due to the built-in capabilities it possesses for both networking and web programming.

In this chapter, we will build a live chat application that demonstrates real-time web application functionality, which will allow the website user to converse with a rudimentary chatbot. As the user asks questions to the bot, the bot will respond in real time, and all communication between the user and the bot will be performed over a WebSocket connection between the web browser and the web server.

In this chapter, we will cover the following topics:

- The live chat feature
- Implementing the live chat's server-side functionality
- Implementing the live chat's client-side functionality
- Conversing with the agent

## The live chat feature

It's common nowadays to see chatbots (also known as agents) service the needs of website users for a wide variety of purposes, from deciding what shoes to purchase to providing tips on what stocks would look good on a client's portfolio, for example. We will build a rudimentary chatbot, that will provide some friendly tips on Isomorphic Go to IGWEB users.

Once the live chat feature is activated, the user can continue to access different sections of the website without having their conversation with the bot interrupted, provided that the user utilizes the navigation menu on the website or links found on the website that are routed on the client side. In a real-world scenario, this functionality would be an attractive proposition for both product sales and technical support usage scenarios. For instance, if a user has a particular question on a product listed on the website, the user can freely browse through the website, without worrying about losing their current chat conversation with the agent.

Keep in mind that the agent that we will build has a low intelligence level. The agent is presented here for illustration purposes only, and a far more robust **Artificial Intelligence (AI)** solution should be utilized for production needs. With the knowledge you will gain from this chapter, it should be fairly straightforward to replace the current agent's brain with a more robust one that will meet your specific needs in the live chat feature.

## Designing the live chatbox

The following diagram is a wireframe design depicting the top bar of IGWEB. The icon at the far right, will active the live chat feature when it is clicked on:



Figure 8.1: Wireframe design depicting IGWEB's top bar

The following diagram is a wireframe design depicting the live chat box. The chat box consists of an avatar image of the agent named "Case" along with its name and title. A close button is included in the top-right corner of the chat box. Users may enter their message to the agent in the bottom text area that has the placeholder text **Type your message here**. The conversation with the human and the bot will be presented in the middle area of the chat box:

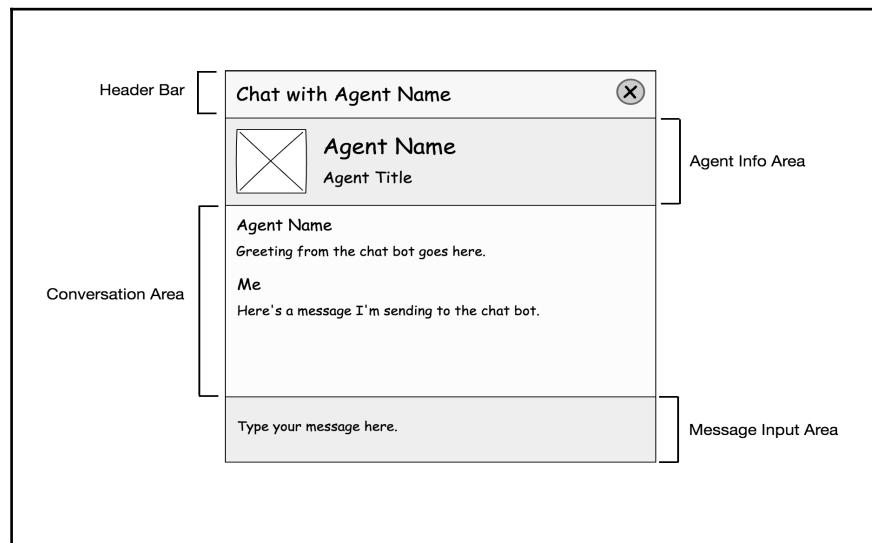


Figure 8.2: Wireframe design of the live chat box

## Implementing the live chat box templates

In order to have the chat box present in all sections of the website, we will need to place the chat box `div` container right below the primary content `div` container in the web page layout template (`layouts/webpage_layout tmpl`):

```
<!doctype html>
<html>
{{ template "partials/header_partial" . }}

<div id="primaryContent" class="pageContent">
{{ template "pagecontent" . }}
</div>

<div id="chatboxContainer" class="containerPulse">
</div>

{{ template "partials/footer_partial" . }}
</html>
```

The chat box will be implemented as a partial template in the `chatbox_partial tmpl` source file in the `shared/templates/partials` folder:

```
<div id="chatbox">
<div id="chatboxHeaderBar" class="chatboxHeader">
<div id="chatboxTitle" class="chatboxHeaderTitle"><span>Chat with
{{.AgentName}}</span></div>
<div id="chatboxCloseControl">X</div>
</div>

<div class="chatboxAgentInfo">
<div class="chatboxAgentThumbnail"></div>
<div class="chatboxAgentName">{{.AgentName}}</div>
<div class="chatboxAgentTitle">{{.AgentTitle}}</div>
</div>

<div id="chatboxConversationContainer">
</div>

<div id="chatboxMsgInputContainer">
<input type="text" id="chatboxInputField" placeholder="Type your
message here...">
</input>
</div>
```

```
<div class="chatboxFooter">
  <a href="http://www.isomorphicgo.org" target="_blank">Powered by
  Isomorphic Go</a>
</div>
</div>
```

This is the HTML markup required to implement the wireframe design depicted in *Figure 8.2* of the live chat box. Note that the `input` textfield having the id "chatboxInputField". This is the `input` field where the user will be able to type their message. Each message created, both the one that the user writes, as well as the one that the bot writes, will use the `livechatmsg_partial tmpl` template:

```
<div class="chatboxMessage">
  <div class="chatSenderName">{{ .Name }}</div>
  <div class="chatSenderMsg">{{ .Message }}</div>
</div>
```

Each message is inside its own `div` container that has two `div` containers (shown in bold) housing the name of the sender of the message and the message itself.

There are no buttons necessary in the live chat feature, since we will be adding an event listener to listen for the press of the `Enter` key to submit the user's message to the server over the WebSocket connection.

Now that we've implemented the HTML markup that will be used to render the chat box, let's examine the functionality required to implement the live chat feature on the server side.

## Implementing the live chat's server-side functionality

When the live chat feature is activated, we will create a persistent, WebSocket connection, between the web client and the web server. The Gorilla Web Toolkit provides an excellent implementation of the WebSocket protocol in their `websocket` package, which can be found at <http://github.com/gorilla/websocket>. To fetch the `websocket` package, you may issue the following command:

```
$ go get github.com/gorilla/websocket
```

The Gorilla web toolkit also provides a useful example web chat application:

<https://github.com/gorilla/websocket/tree/master/examples/chat>.

Rather than reinventing the wheel, we will repurpose Gorilla's example web chat application to fulfill the live chat feature. The source files needed from the web chat example have been copied over to the `chat` folder.

There are three major changes we need to make to realize the live chat feature using the example chat application provided by Gorilla:

- Replies from the chatbot (the agent) should be targeted to a specific user, and not be sent out to every connected user
- We need to create the functionality to allow the chatbot to send a message back to the user
- We need to implement the front-end portion of the chat application in Go

Let's consider each of these three points in more detail.

First, Gorilla's web chat example is a free-for-all chat room. Any user can come in, type a message, and all other users connected to the chat server will be able to see the message. A major requirement for the live chat feature is that each conversation between the chatbot and the human should be exclusive. Replies from the agent must be targeted to a specific user, and not to all connected users.

Second, the example web chat application from the Gorilla web toolkit doesn't send any messages back to the user. This is where the custom chatbot comes into the picture. The agent will communicate directly with the user over the established WebSocket connection.

Third, the front-end portion of the example web chat application is implemented as a HTML document containing inline CSS and JavaScript. As you may have guessed already, we will implement the front-end portion for the live chat feature in Go, and the code will reside in the `client/chat` folder.

Now that we have established our plan of action to implement the live chat feature using the Gorilla web chat example as a foundation to start from, let's begin the implementation.

The modified web chat application that we will create contains two main types: `Hub` and `Client`.

## The hub type

The chat hub is responsible for maintaining a list of client connections and directing the chatbot to broadcast a message to the relevant client. For example, if Alice asked the question "What is Isomorphic Go?", the answer from the chatbot should go to Alice and not to Bob (who may not have even asked a question yet).

Here's what the Hub struct looks like:

```
type Hub struct {
    chatbot bot.Bot
    clients map[*Client]bool
    broadcastmsg chan *ClientMessage
    register chan *Client
    unregister chan *Client
}
```

The `chatbot` is a chat bot (agent) that implements the `Bot` interface. This is the brain that will answer the questions received from clients.

The `clients` map is used to register clients. The key-value pair stored in the `map` consists of the key, a pointer to a `Client` instance, and the value consists of a Boolean value set to `true` to indicate that the client is connected. Clients communicate with the hub over the `broadcastmsg`, `register`, and `unregister` channels. The `register` channel registers a client with the hub. The `unregister` channel, unregisters a client with the hub. The client sends the message entered by the user over the `broadcastmsg` channel, a channel of type `ClientMessage`. Here's the `ClientMessage` struct that we have introduced:

```
type ClientMessage struct {
    client *Client
    message []byte
}
```

To fulfill the first major change we laid out previously, that is, the exclusivity of the conversation between the agent and the user, we use the `ClientMessage` struct to store, both the pointer to the `Client` instance that sent the user's message along with the user's message itself (a `byte` slice).

The constructor function, `NewHub`, takes in `chatbot` that implements the `Bot` interface and returns a new `Hub` instance:

```
func NewHub(chatbot bot.Bot) *Hub {
    return &Hub{
        chatbot: chatbot,
        broadcastmsg: make(chan *ClientMessage),
        register: make(chan *Client),
        unregister: make(chan *Client),
        clients: make(map[*Client]bool),
    }
}
```

We implement an exported getter method, `ChatBot`, so that the `chatbot` can be accessed from the `Hub` object:

```
func (h *Hub) ChatBot() bot.Bot {
    return h.chatbot
}
```

This action will be significant when we implement a Rest API endpoint to send the bot's details (its name, title, and avatar image) to the client.

The `SendMessage` method is responsible for broadcasting a message to a particular client:

```
func (h *Hub) SendMessage(client *Client, message []byte) {
    client.send <- message
}
```

The method takes in a pointer to `Client`, and the `message`, which is a `byte` slice, that should be sent to that particular client. The message will be sent over the client's `send` channel.

The `Run` method is called to start the chat hub:

```
func (h *Hub) Run() {
    for {
        select {
        case client := <-h.register:
            h.clients[client] = true
            greeting := h.chatbot.Greeting()
            h.SendMessage(client, []byte(greeting))

        case client := <-h.unregister:
            if _, ok := h.clients[client]; ok {
                delete(h.clients, client)
                close(client.send)
            }
        }
    }
}
```

```
        }
        case clientmsg := <-h.broadcastmsg:
            client := clientmsg.client
            reply := h.chatbot.Reply(string(clientmsg.message))
            h.SendMessage(client, []byte(reply))
        }
    }
}
```

We use the `select` statement inside the `for` loop to wait on multiple client operations.

In the case that a pointer to a `Client` comes in over the hub's `register` channel, the hub will register the new client by adding the `client` pointer (as the key) to the `clients` map and set a value of `true` for it. We will fetch a greeting message to return to the client by calling the `Greeting` method on `chatbot`. Once we get the greeting (a string value), we call the `SendMessage` method passing in the `client` and the greeting converted to a `byte` slice.

In the case that a pointer to a `Client` comes in over the hub's `unregister` channel, the hub will remove the entry in `map` for the given `client` and close the client's `send` channel, which signifies that the `client` won't be sending any more messages to the server.

In the case that a pointer to a `ClientMessage` comes in over the hub's `broadcastmsg` channel, the hub will pass the client's `message` (as a string value) to the `Reply` method of the `chatbot` object. Once we get `reply` (a string value) from the agent, we call the `SendMessage` method passing in the `client` and the `reply` converted to a `byte` slice.

## The client type

The `Client` type acts as a broker between `Hub` and the `websocket` connection.

Here's what the `Client` struct looks like:

```
type Client struct {
    hub *Hub
    conn *websocket.Conn
    send chan []byte
}
```

Each `Client` value contains a pointer to `Hub`, a pointer to a `websocket` connection, and a buffered channel, `send`, meant for outbound messages.

The `readPump` method is responsible for relaying inbound messages coming in over the websocket connection to the hub:

```
func (c *Client) readPump() {
    defer func() {
        c.hub.unregister <- c
        c.conn.Close()
    }()
    c.conn.SetReadLimit(maxMessageSize)
    c.conn.SetReadDeadline(time.Now().Add(pongWait))
    c.conn.SetPongHandler(func(string) error {
        c.conn.SetReadDeadline(time.Now().Add(pongWait)); return nil })
    for {
        _, message, err := c.conn.ReadMessage()
        if err != nil {
            if websocket.IsUnexpectedCloseError(err, websocket.CloseGoingAway) {
                log.Printf("error: %v", err)
            }
            break
        }
        message = bytes.TrimSpace(bytes.Replace(message, newline, space, -1))
        // c.hub.broadcast <- message

        clientmsg := &ClientMessage{client: c, message: message}
        c.hub.broadcastmsg <- clientmsg
    }
}
```

We had to make a slight change to this function to fulfill the requirements of the live chat feature. In the Gorilla web chat example, the message alone was relayed over to Hub. Since we are directing chat bot responses, back to the client that sent them, not only do we need to send the message to the hub, but also the client that happened to send the message (shown in bold). We do so by creating a `ClientMessage` struct:

```
type ClientMessage struct {
    client *Client
    message []byte
}
```

The `ClientMessage` struct contains fields to hold both the pointer to the client as well as the message, a byte slice.

Going back to the `readPump` function in the `client.go` source file, the following two lines are instrumental in allowing the Hub to know which client sent the message:

```
clientmsg := &ClientMessage{client: c, message: message}
c.hub.broadcastmsg <- clientmsg
```

The `writePump` method is responsible for relaying outbound messages from the client's send channel over the websocket connection:

```
func (c *Client) writePump() {
    ticker := time.NewTicker(pingPeriod)
    defer func() {
        ticker.Stop()
        c.conn.Close()
    }()
    for {
        select {
        case message, ok := <-c.send:
            c.conn.SetWriteDeadline(time.Now().Add(writeWait))
            if !ok {
                // The hub closed the channel.
                c.conn.WriteMessage(websocket.CloseMessage, []byte{})
                return
            }
            w, err := c.conn.NextWriter(websocket.TextMessage)
            if err != nil {
                return
            }
            w.Write(message)

            // Add queued chat messages to the current websocket message.
            n := len(c.send)
            for i := 0; i < n; i++ {
                w.Write(newline)
                w.Write(<-c.send)
            }

            if err := w.Close(); err != nil {
                return
            }
        case <-ticker.C:
            c.conn.SetWriteDeadline(time.Now().Add(writeWait))
            if err := c.conn.WriteMessage(websocket.PingMessage, []byte{}); err
        != nil {
                return
            }
        }
    }
}
```

```
    }
}
}
```

The `ServeWs` method is meant to be registered as an HTTP handler by the web application:

```
func ServeWs(hub *Hub) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        conn, err := upgrader.Upgrade(w, r, nil)
        if err != nil {
            log.Println(err)
            return
        }
        client := &Client{hub: hub, conn: conn, send: make(chan []byte, 256)}
        client.hub.register <- client
        go client.writePump()
        client.readPump()
    })
}
```

This method performs two important tasks. The method upgrades the normal HTTP connection to a websocket connection and registers the client to the hub.

Now that we've set up the code for our web chat server, it's time to activate it in our web application.

## Activating the chat server

In the `igweb.go` source file, we have included a function called `startChatHub`, which is responsible for starting the Hub:

```
func startChatHub(hub *chat.Hub) {
    go hub.Run()
}
```

We add the following code in the `main` function to create a new chatbot, associate it with the Hub, and start the Hub:

```
chatbot := bot.NewAgentCase()
hub := chat.NewHub(chatbot)
startChatHub(hub)
```

When we call the `registerRoutes` function to register all the routes for the server-side web application, note that we also pass in the `hub` value to the function:

```
r := mux.NewRouter()
registerRoutes(&env, r, hub)
```

In the `registerRoutes` function, we need the `hub` to register the route handler for the Rest API endpoint that returns the agent's information:

```
r.Handle("/restapi/get-agent-info", endpoints.GetAgentInfoEndpoint(env,
hub.ChatBot()))
```

We will go over this endpoint in the section, *Exposing the agent's information to the client*.

The `hub` is also used to register the route handler for the WebSocket route, `/ws`. We register the `ServeWs` handler function, passing in the `hub`:

```
r.Handle("/ws", chat.ServeWs(hub))
```

Now that we have everything in place to activate the chat server, it's time to focus on the star of the live chat feature—the chat agent.

## The agent's brain

The chat bot type that we will use for the live chat feature, `AgentCase`, will implement the following `Bot` interface:

```
type Bot interface {
    Greeting() string
    Reply(string) string
    Name() string
    Title() string
    ThumbnailPath() string
    SetName(string)
    SetTitle(string)
    SetThumbnailPath(string)
}
```

The `Greeting` method will be used to send an initial greeting to the user, enticing them to interact with the chat bot.

The `Reply` method accepts a question (a string) and returns a reply (also a string) for the given question.

The rest of the methods implemented are for purely psychological reasons to give humans the illusion that they are communicating with someone, rather than something.

The `Name` method is a getter method that returns the chat bot's name.

The `Title` method is a getter method that returns the chat bot's title.

The `ThumbnailPath` method is a getter method that returns the path to the chat bot's avatar image.

Each of the getter methods has a corresponding setter method: `SetName`, `SetTitle`, and `SetThumbnailPath`.

By defining the `Bot` interface, we are clearly stating the expectations of a chat bot. This allows us to make the chat bot solution extensible in the future. For example, the intelligence that `Case` exhibits may be too rudimentary and limiting. In the near future, we may want to implement a bot named `Molly`, whose intelligence may be implemented using a more powerful algorithm. As long as the `Molly` chat bot implements the `Bot` interface, the new chat bot can be easily plugged into our web application.

In fact, from the perspective of the server-side web application, it would just be a one-line code change. Instead of instantiating an `AgentCase` instance, we would instantiate an `AgentMolly` instance instead. Besides the difference in intelligence, the new chat bot, `Molly`, would come with its own name, title, and avatar image, so humans would be able to differentiate it from `Case`.

Here's the `AgentCase` struct:

```
type AgentCase struct {
    Bot
    name string
    title string
    thumbnailPath string
    knowledgeBase map[string]string
    knowledgeCorpus []string
    sampleQuestions []string
}
```

We have embedded the `Bot` interface to the `struct` definition, indicating that the `AgentCase` type will implement the `Bot` interface. The field `name` is for the name of the agent. The field `title` is for the title of the agent. The field `thumbnailPath` is used to specify the path to the chat bot's avatar image.

The `knowledgeBase` field is `map` of type `map[string]string`. This is essentially the agent's brain. Keys in the `map` are the common terms found in a particular question. Values in the `map` are the answers to the question.

The `knowledgeCorpus` field, a `string` `byte` slice, is a knowledge corpus of the terms that may exist in questions that the bot will be asked. We use the keys of the `knowledgeBase` `map` to construct the `knowledgeCorpus`. A corpus is a collection of text that is used to conduct linguistic analysis. In our case, we will conduct the linguistic analysis based on the question (the query) that the human user provided to the bot.

The `sampleQuestions` field, a `string` `byte` slice, will contain a list of sample questions that the user may ask the chat bot. The chat bot will provide the user with a sample question when it greets them to entice the human user into a conversation. It is understood that the human user is free to paraphrase the sample question or ask an entirely different question depending on their preference.

The `initializeIntelligence` method is used to initialize Case's brain:

```
func (a *AgentCase) initializeIntelligence() {  
  
    a.knowledgeBase = map[string]string{  
        "isomorphic go isomorphic go web applications": "Isomorphic Go is the  
        methodology to create isomorphic web applications using the Go (Golang)  
        programming language. An isomorphic web application, is a web application,  
        that contains code which can run, on both the web client and the web  
        server.",  
        "kick recompile code restart web server instance instant kickstart  
        lightweight mechanism": "Kick is a lightweight mechanism to provide an  
        instant kickstart to a Go web server instance, upon the modification of a  
        Go source file within a particular project directory (including any  
        subdirectories). An instant kickstart consists of a recompilation of the Go  
        code and a restart of the web server instance. Kick comes with the ability  
        to take both the go and gopherjs commands into consideration when  
        performing the instant kickstart. This makes it a really handy tool for  
        isomorphic golang projects.",  
        "starter code starter kit": "The isogoapp, is a basic, barebones web  
        app, intended to be used as a starting point for developing an Isomorphic  
        Go application. Here's the link to the github page:  
        https://github.com/isomorphicgo/isogoapp",  
        "lack intelligence idiot stupid dumb dummy don't know anything":  
        "Please don't question my intelligence, it's artificial after all!",  
        "find talk topic presentation lecture subject": "Watch the Isomorphic  
        Go talk by Kamesh Balasubramanian at GopherCon India:  
        https://youtu.be/zrsuxZEoTcs",  
        "benefits of the technology significance of the technology importance
```

```
of the technology": "Here are some benefits of Isomorphic Go: Unlike
JavaScript, Go provides type safety, allowing us to find and eliminate many
bugs at compile time itself. Eliminates mental context-shifts between back-
end and front-end coding. Page loading prompts are not necessary.",

"perform routing web app register routes": "You can
implement client-side routing in your web application using the isokit
Router preventing the dreaded full page reload.",

"render templates perform template rendering": "Use template sets, a
set of project templates that are persisted in memory and are available on
both the server-side and the client-side",

"cogs reusable components react-like react": "Cogs are reusable
components in an Isomorphic Go web application.",

}

a.knowledgeCorpus = make([]string, 1)
for k, _ := range a.knowledgeBase {
    a.knowledgeCorpus = append(a.knowledgeCorpus, k)
}

a.sampleQuestions = []string{"What is isomorphic go?", "What are the
benefits of this technology?", "Does isomorphic go offer anything react-
like?", "How can I recompile code instantly?", "How can I perform routing
in my web app?", "Where can I get starter code?", "Where can I find a talk
on this topic?"}

}
```

There are three important tasks that occur within this method:

- First, we set Case's knowledge base.
- Second, we set Case's knowledge corpus.
- Third, we set the sample questions, which Case will utilize when greeting the human user.

The first task we must take care of is to set Case's knowledge base. This consists of setting the `knowledgeBase` property of the `AgentCase` instance. As mentioned earlier, the keys in the map refer to terms found in the question, and the values in the map are the answers to the question. For example, the "isomorphic go isomorphic go web applications" key could service the following questions:

- What is Isomorphic Go?
- What can you tell me about Isomorphic Go?

It can also service statements that aren't questions:

- Tell me about Isomorphic Go
- Give me the rundown on Isomorphic Go



Due to the the large amount of text contained within the map literal declaration for the knowledgeBase map, I encourage you to view the source file, `agentcase.go`, on a computer.

The second task we must take care of is to set Case's corpus, the collection of text used for linguistic analysis used against the user's question. The corpus is constructed from the keys of the knowledgeBase map. We set the knowledgeCorpus field property of the `AgentCase` instance to a newly created string byte slice using the built-in `make` function. Using a `for` loop, we iterate through all the entries in the `knowledgeBase` map and append each key to the `knowledgeCorpus` field slice.

The third and last task we must take care of is to set the sample questions that Case will present to the human user. We simply populate the `sampleQuestions` property of the `AgentCase` instance. We use the string literal declaration to populate all the sample questions that are contained in the string byte slice.

Here are the getter and setter methods of the `AgentCase` type:

```
func (a *AgentCase) Name() string {
    return a.name
}

func (a *AgentCase) Title() string {
    return a.title
}

func (a *AgentCase) ThumbnailPath() string {
    return a.thumbnailPath
}

func (a *AgentCase) SetName(name string) {
    a.name = name
}

func (a *AgentCase) SetTitle(title string) {
    a.title = title
}

func (a *AgentCase) SetThumbnailPath(thumbnailPath string) {
```

```
a.thumbnailPath = thumbnailPath
}
```

These methods are used to get and set the name, title, and thumbnailPath fields of the AgentCase object.

Here's the constructor function used to create a new AgentCase instance:

```
func NewAgentCase() *AgentCase {
    agentCase := &AgentCase{name: "Case", title: "Resident Isomorphic Gopher
Agent", thumbnailPath: "/static/images/chat/Case.png"}
    agentCase.initializeIntelligence()
    return agentCase
}
```

We declare and initialize the agentCase variable with a new AgentCase instance, setting the fields for name, title, and thumbnailPath. We then call the initializeIntelligence method to initialize Case's brain. Finally, we return the newly created and initialized AgentCase instance.

## Greeting the human

The Greeting method is used to provide a first-time greeting to the user when the live chat feature is activated:

```
func (a *AgentCase) Greeting() string {
    sampleQuestionIndex := randomNumber(0, len(a.sampleQuestions))
    greeting := "Hi there! I'm Case. You can ask me a question on Isomorphic
    Go. Such as...\" + a.sampleQuestions[sampleQuestionIndex] + "\""
    return greeting
}
```

Since the greeting will include a randomly selected sample question that can be asked to Case, the randomNumber function is called to obtain the index number of the sample question. We pass the minimum value and the maximum value to the randomNumber function to specify the range that the produced random number should be in.

Here's the `randomNumber` function used to generate a random number within a given range:

```
func randomNumber(min, max int) int {  
    rand.Seed(time.Now().UTC().UnixNano())  
    return min + rand.Intn(max-min)  
}
```

Returning to the `Greeting` method, we get the sample question retrieved from the `sampleQuestions` string slice using the random index. We then assign the sample question to the `greeting` variable and return `greeting`.

## Replying to a human's question

Now that we've initialized the intelligence of the chat bot and prepared it to greet the human user, it's time to guide the chat bot on how to think about a user's question so that the chat bot may offer a sensible reply.

The replies that the chat bot will send to the human user are limited to those found as the values in the `knowledgeBase` map of the `AgentCase` struct. If the human user asks a question outside the scope of what the chat bot knows (the knowledge corpus), it will simply reply with the message "I don't know the answer to that one."

To analyze the user's question and provide the best reply for it, we will be using the `nlp` package, which contains a collection of machine learning algorithms that can be used for basic natural language processing.

You can install the `nlp` package by issuing the following `go get` command:

```
$ go get github.com/james-bowman/nlp
```

Let's go over the `Reply` method piece by piece, starting with the method declaration:

```
func (a *AgentCase) Reply(query string) string {
```

The function takes in a question string and returns an answer string for the given question.

We declare the `result` variable that represents the answer to the user's question:

```
    var result string
```

The `result` variable will be returned by the `Reply` method.

Using the `nlp` package, we create a new `vectoriser` and a new `transformer`:

```
vectoriser := nlp.NewCountVectorizer(true)
transformer := nlp.NewTfidfTransformer()
```

The `vectoriser` will be used to encode the query terms from the knowledge corpus into a term document matrix, where each column will represent a document within the corpus, and each row represents a term. It is used to keep track of the frequency of terms found within a particular document. For our usage scenario, you may consider the document as the unique entry found in the `knowledgeCorpus` string slice.

The `transformer` will be used to remove bias of frequently appearing terms in `knowledgeCorpus`. For example, words that are repeated across `knowledgeCorpus` such as *the*, *and*, and *web* will have a lesser weight. The transformer is a **TFIDF (term frequency inverse document frequency)** transformer.

We then proceed to create `reducer`, which is a new `TruncatedSVD` instance:

```
reducer := nlp.NewTruncatedSVD(4)
```

The `reducer` that we just declared is significant, since we will be performing **Latent Semantic Analysis (LSA)**, also known as **Latent Semantic Indexing (LSI)**, for the search and retrieval of the proper document to the user's query term. LSA helps us find semantic attributes that exist within a corpus based on the co-occurrence of terms. It assumes that words that frequently appear together must have some semantic relation.

The `reducer` is used to find semantic meaning that may be hidden beneath the term frequencies within the document feature vectors.

The following code is a pipeline that transforms the corpus into a Latent Semantic Index, which fits the models to the documents:

```
matrix, _ := vectoriser.FitTransform(a.knowledgeCorpus...)
matrix, _ = transformer.FitTransform(matrix)
lsi, _ := reducer.FitTransform(matrix)
```

We have to run the user's query through the same pipeline so that it is projected in the same dimensional space:

```
matrix, _ = vectoriser.Transform(query)
matrix, _ = transformer.Transform(matrix)
queryVector, _ := reducer.Transform(matrix)
```

Now that we have `lsi` and the `queryVector` in place, it is time to find the document that best matches the query term. We do so by calculating the cosine similarity of each document in our corpus against the query:

```
highestSimilarity := -1.0
var matched int
_, docs := lsi.Dims()
for i := 0; i < docs; i++ {
    similarity :=
nlp.CosineSimilarity(queryVector.(mat.ColViewer).ColView(0),
lsi.(mat.ColViewer).ColView(i))
    if similarity > highestSimilarity {
        matched = i
        highestSimilarity = similarity
    }
}
```



The **cosine similarity** calculates the difference between the angles of two numerical vectors.

The document in the corpus having the highest similarity with the user's query will be matched as the best document that reflects the user's question. The possible values for the cosine similarity can fall between the range of 0 to 1. A 0 value indicates complete orthogonality and a 1 value indicates a perfect match. The cosine similarity value can also be a **NaN (not a number)** value. The NaN value is indicative that there is no match at all.

The `highestSimilarity` value will be `-1` if no match was found; otherwise, it will be a value between `0` and `1`:

```
if highestSimilarity == -1 {
    result = "I don't know the answer to that one."
} else {
    result = a.knowledgeBase[a.knowledgeCorpus[matched]]
}

return result
```

In the `if` conditional block, we check whether the `highestSimilarity` value is `-1`; if it is, the answer to the user will be "I don't know the answer to that one.".

If we reach the `else` block, it indicates that the `highestSimilarity` is a value between 0 and 1, indicating that a match was found. Recall that the document in our `knowledgeCorpus` has a corresponding key in the `knowledgeBase` map. The answer to the user's question is the value in the `knowledgeBase` map with the provided key, and we set the `result` string to this value. In the last line of code in the method, we return the `result` variable.



The logic to implement the chatbot's intelligence was inspired from James Bowman's article, *Semantic analysis of webpages with machine learning in Go* (<http://www.jamesbowman.me/post/semantic-analysis-of-webpages-with-machine-learning-in-go/>).

## Exposing the agent's information to the client

Now that we've implemented the chat agent, `AgentCase`, we need a way to expose Case's information to the client, notably, its name, title, and the path to its avatar image.

We create a new Rest API endpoint, `GetAgentInfoEndpoint`, to expose the chat agent's information to the client-side web application:

```
func GetAgentInfoEndpoint(env *common.Env, chatbot bot.Bot) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        m := make(map[string]string)
        m["AgentName"] = chatbot.Name()
        m["AgentTitle"] = chatbot.Title()
        m["AgentThumbImagePath"] = chatbot.ThumbnailPath()
        w.Header().Set("Content-Type", "application/json")
        json.NewEncoder(w).Encode(m)
    })
}
```

Note that in the signature of the `GetAgentInfoEndpoint` function, we accept the `env` object and the `chatbot` object. Note that the `chatbot` is of type `bot.Bot`, an interface type, rather than the `AgentCase` type. This provides us with the flexibility to easily swap in another bot, such as `AgentMolly`, instead of `AgentCase`, in the future.

We simply create a map, `m`, of type `map[string]string`, containing the bot's name, title, and avatar image path. We set a header to indicate that the server response will be in the JSON format. Finally, we write out the JSON encoded map using `http.ResponseWriter`, `w`.

# Implementing the live chat's client-side functionality

Now that we've covered the server-side functionality needed to implement the chat bot, it's time focus on the live chat feature from the perspective of the client-side web application.

Inside the `InitialPageLayoutControls` function, we add an event listener on the `click` event to the live chat icon found on the top bar:

```
liveChatIcon :=  
env.Document.GetElementByID("liveChatIcon").(*dom.HTMLImageElement)  
liveChatIcon.AddEventListener("click", false, func(event dom.Event) {  
  
    chatbox := env.Document.GetElementByID("chatbox")  
    if chatbox != nil {  
        return  
    }  
    go chat.StartLiveChat(env)  
})
```

If the live chat feature has already been activated, the `chatbox` div element will already exist, that is, it will be a non-nil value. In this scenario, we return from the function.

However, in the case that the live chat feature has not been activated yet, we call the `StartLiveChat` function located in the `chat` package as a goroutine, passing in the `env` object. Calling this function, will activate the live chat feature.

## Creating the live chat client

We will use the `gopherjs/websocket/websocketjs` package to create a WebSocket connection that will be used to connect to the web server instance.

You can install this package using the following `go get` command:

```
$ go get -u github.com/gopherjs/websocket
```

The client-side implementation of the live chat feature can be found in the `client/chat/chat.go` source file. We define the `ws` variable of the type `websocketjs.WebSocket` and the `agentInfo` variable of type `map[string]string`:

```
var ws *websocketjs.WebSocket  
var agentInfo map[string]string
```

We also declare a constant that represents the key code for the **Enter** key:

```
const ENTERKEY int = 13
```

The `GetAgentInfoRequest` function is used to obtain the agent information from the `/restapi/get-agent-info` endpoint:

```
func GetAgentInfoRequest(agentInfoChannel chan map[string]string) {
    data, err := xhr.Send("GET", "/restapi/get-agent-info", nil)
    if err != nil {
        println("Encountered error: ", err)
    }
    var agentInfo map[string]string
    json.NewDecoder(strings.NewReader(string(data))).Decode(&agentInfo)
    agentInfoChannel <- agentInfo
}
```

Once we retrieve the JSON encoded data from the server, we decode it to `map` of type `map[string]string`. We then send the `agentInfo` `map` over the channel, `agentInfoChannel`.

The `getServerPort` function is a helper function to obtain the port the server is running on:

```
func getServerPort(env *common.Env) string {
    if env.Location.Port != "" {
        return env.Location.Port
    }

    if env.Location.Protocol == "https" {
        return "443"
    } else {
        return "80"
    }
}
```

This function is used when constructing the `serverEndpoint` `string` variable inside the `StartLiveChat` function, which represents the server endpoint that we'll be making the `WebSocket` connection to.

When the user clicks on the live chat icon in the top bar, the `StartLiveChat` function will be called as a goroutine:

```
func StartLiveChat(env *common.Env) {  
  
    agentInfoChannel := make(chan map[string]string)  
    go GetAgentInfoRequest(agentInfoChannel)  
    agentInfo = <-agentInfoChannel
```

We first start out by fetching the agent's info by calling the `GetAgentInfoRequest` function as a goroutine. The agent's information will be sent as a map of the type `map[string]string` over the `agentInfoChannel` channel. The `agentInfo` map will be used as the data object that is passed to the `partials/chatbox_partial` template to display the agent's details (name, title, and avatar image).

We then proceed to create a new WebSocket connection and connect to the server endpoint:

```
var err error  
serverEndpoint := "ws://" + env.Location.Hostname + ":" +  
getServerPort(env) + "/ws"  
ws, err = websocketjs.New(serverEndpoint)  
if err != nil {  
    println("Encountered error when attempting to connect to the websocket:  
", err)  
}
```

We use the helper `getServerPort` function to get the port that the server is running on. The server port value is used when constructing the `serverEndpoint` string variable, which represents the WebSocket address of the server endpoint we will be connecting to.

We use the `env.Document` object's `GetElementByID` method to get the chat container `div` element by supplying the ID of `"chatboxContainer"`. We also add a CSS animated style to make the chat box container pulsate for a dramatic effect when the chatbot is available to answer questions:

```
chatContainer :=  
env.Document.GetElementByID("chatboxContainer").(*dom.HTMLDivElement)  
chatContainer.SetClass("containerPulse")  
  
env.TemplateSet.Render("partials/chatbox_partial",  
&isokit.RenderParams{Data: agentInfo, Disposition:  
isokit.PlacementReplaceInnerContents, Element: chatContainer})
```

We call the `Render` method of the template set object, rendering the "partials/chatbox\_partial" template and supplying the template render parameters. We specify that the data object that is to be fed to the template will be the `agentInfo` map. We specify that the disposition of the rendering should be to replace the inner HTML contents of the associated element with the rendered template output. Finally, we specify that the associated element to render to is the `chatContainer` element.

When the live chat feature is functional and the WebSocket connection to the server is connected, then the chat box header bar, the strip containing the chat box title, the `chatboxHeaderBar`, will be colored green. If the WebSocket connection has been disconnected or there is an error, the strip will be colored red. By default, the strip will be colored green when we set the default CSS class of the `chatboxHeaderBar` to "chatboxHeader":

```
chatboxHeaderBar :=  
env.Document.GetElementByID("chatboxHeaderBar").(*dom.HTMLDivElement)  
chatboxHeaderBar.SetClass("chatboxHeader")
```

## Initializing the event listeners

Finally, we call the `InitializeChatEventHandlers` function, passing in the `env` object, to initialize the event handlers for the live chat feature:

```
InitializeChatEventHandlers(env)
```

The `InitializeChatEventHandlers` function is responsible for setting up all the event listeners required by the live chat feature. There are two controls that require user interaction. The first is the message input field, where the user types and sends their questions by hitting the **Enter** key. The second is the close button, the **X**, that's found on the upper right hand corner of the chat box, used to close the live chat feature.

To handle the user interaction with the message input field, we set up the `keypress` event listener, which will detect a `keypress` event inside the message input `textfield` element:

```
func InitializeChatEventHandlers(env *common.Env) {  
  
    msgInput :=  
    env.Document.GetElementByID("chatboxInputField").(*dom.HTMLInputElement)  
    msgInput.AddEventListener("keypress", false, func(event dom.Event) {  
        if event.Underlying().Get("keyCode").Int() == ENTERKEY {  
            event.PreventDefault()  
            go ChatSendMessage(env, msgInput.Value)  
            msgInput.Value = ""  
        }  
    })  
}
```

```
    }  
})
```

We fetch the `input` message textfield element by calling the `GetElementByID` method on the `env.Document` object. We then attach a `keypress` event listener function to the element. If the key that the user presses is the `Enter` key, then we will prevent the default behavior of the `keypress` event and call the `ChatSendMessage` function, as a goroutine, passing in the `env` object and the `Value` property of the `msgInput` element. Finally, we clear the text in the message input field by setting its `Value` property to the empty string value.

## The close chat control

To handle the user interaction when clicking on the `X` control to close the live chat feature, we set up an event listener to handle the `click` event of the close control:

```
closeControl :=  
env.Document.GetElementByID("chatboxCloseControl").(*dom.HTMLDivElement)  
closeControl.AddEventListener("click", false, func(event dom.Event) {  
    CloseChat(env)  
})
```

We get the `div` element that represents the close control by calling the `GetElementByID` method on the `env.Document` object, specifying the ID `"chatboxCloseControl"`. We attach an event listener, to the close control, on the `click` event, which will call the `CloseChat` function.

## Setting up event listeners for the WebSocket object

Now that we've set up the event listeners for the user interactions, we must set up event listeners on the `WebSocket` object, `ws`. We first add an event listener on the `message` event:

```
ws.AddEventListener("message", false, func(ev *js.Object) {  
    go HandleOnMessage(env, ev)  
})
```

The `message` event listener will be triggered when a new message comes across the WebSocket connection. This is indicative of the agent sending a message back to the user. In this situation we call the `HandleOnMessage` function, passing in the `env` object and the event object, `ev`, to the function.

The other event we have to listen for from the WebSocket object is the `close` event. This event can be triggered from a normal operating scenario, such as the user closing the live chat feature using the close control. The event can also be triggered from an abnormal operating scenario, such as the web server instance suddenly going down, breaking off the WebSocket connection. Our code must be intelligent enough to fire only in the abnormal connection closing scenario:

```
ws.AddEventListener("close", false, func(ev *js.Object) {  
    chatboxContainer :=  
        env.Document.GetElementByID("chatboxContainer").(*dom.HTMLDivElement)  
    if len(chatboxContainer.ChildNodes()) > 0 {  
        go HandleDisconnection(env)  
    }  
})
```

We do so by first fetching the chatbox container `div` element. If the number of child nodes within the chatbox container is greater than zero, it means that the connection has abnormally closed while the user is using the live chat feature, and we must call the `HandleDisconnection` function, as a goroutine, passing in the `env` object to the function.

There may be certain scenarios, where the `close` event will not fire, such as when we lose internet connectivity. The TCP connection that the WebSocket connection is communicating over, may still be considered to be live even though the internet connection has been disconnected. In order for our live chat feature to be resilient to handle this scenario, we need to listen for the `env.Window` object's `offline` event, which will fire when the network connection is lost:

```
env.Window.AddEventListener("offline", false, func(event dom.Event) {  
    go HandleDisconnection(env)  
})  
}
```

We perform the same action as we did previously to handle this event. We call the `HandleDisconnection` function, as a goroutine, passing in the `env` object to the function. Take note that the last closing brace, `}`, indicates the end of the `InitializeChatEventHandlers` function.

Now that we have set up all the necessary event listeners for the live chat feature, it's time to examine each function that was called by the event listeners we just set up.

The `ChatSendMessage` function is called after the user hits the `Enter` key inside the message input textfield:

```
func ChatSendMessage(env *common.Env, message string) {
    ws.Send([]byte(message))
    UpdateChatBox(env, message, "Me")
}
```

We call the `Send` method of the `WebSocket` object, `ws`, to send the user's question over to the web server. We then call the `UpdateChatBox` function to render the user's message to the chat box's conversation container. We pass in the `env` object, `message` the user wrote, and the `sender` string as input values to the `UpdateChatBox` function. The `sender` string is the person who sent the message; in this case, since the user sent it, the `sender` string will be `"Me"`. The `sender` string helps the user differentiate between messages the user sent versus messages the chatbot replied with.

The `UpdateChatBox` function is used to update the chat box conversation container area:

```
func UpdateChatBox(env *common.Env, message string, sender string) {
    m := make(map[string]string)
    m["Name"] = sender
    m["Message"] = message
    conversationContainer :=
        env.Document.GetElementByID("chatboxConversationContainer").(*dom.HTMLDivElement)
    env.TemplateSet.Render("partials/livechatmsg_partial",
        &isokit.RenderParams{Data: m, Disposition: isokit.PlacementAppendTo,
        Element: conversationContainer})
    scrollHeight := conversationContainer.Underlying().Get("scrollHeight")
    conversationContainer.Underlying().Set("scrollTop", scrollHeight)
}
```

We create a new map of the `map[string]string` type, which will be used as the data object that will be fed to the `partials/livechatmsg_partial` template. The map consists of an entry with the key `"Name"` to represent `sender` and an entry with the key `"Message"` for the message. The values for both the `"Name"` and the `"Message"` will be displayed in the chat box's conversation container area.

We obtain the element for `conversationContainer` by calling the `env.Document` object's `GetElementByID` method and specifying the `id` value of `"chatboxConversationContainer"`.

We call the `Render` method of the `env.TemplateSet` object and specify that we want to render the `partials/livechatmsg_partial` template. In the `render` parameters (`RenderParams`) object, we set the `Data` field to the map, `m`. We set the `Disposition` field to `isokit.PlacementAppendTo` to specify that the disposition operation will be an *append to* operation relative to the associated element. We set the `Element` field to the `conversationContainer`, since this is the element where the chat messages will be appended to.

The final two lines in the function will auto-scroll the `conversationContainer` all the way to the bottom upon rendering a new message so that the most recent message will always be displayed to the user.

Besides the `ChatSendMessage` function, the other utilizer of the `UpdateChatBox` function is the `HandleOnMessage` function:

```
func HandleOnMessage(env *common.Env, ev *js.Object) {  
    response := ev.Get("data").String()  
    UpdateChatBox(env, response, agentInfo["AgentName"])  
}
```

Recall that this function will be called upon the `"message"` event being fired from the WebSocket connection. We fetch the response from the chatbot, communicated over the WebSocket connection, by getting the string value of the `data` property of the event object. We then call the `UpdateChatBox` function passing in the `env` object, the `response` string, and the `sender` string, `agentInfo["AgentName"]`. Note that we have passed the name of the agent, the value in the `agentInfo` map obtained using the `"AgentName"` key, as the `sender` string.

The `CloseChat` function is used to close the web socket connection and dismiss the chat box from the user interface:

```
func CloseChat(env *common.Env) {  
    ws.Close()  
    chatboxContainer :=  
    env.Document.GetElementByID("chatboxContainer").(*dom.HTMLDivElement)  
    chatboxContainer.RemoveChild(chatboxContainerchildNodes()[0])  
}
```

We first call the `Close` method on the `WebSocket` object. We get the `chatboxContainer` element and remove its first child node, which will subsequently remove all the children of the first child node.

Keep in mind that this function will be called either when the user hits the **X** control in the chat box, or in the scenario of an abnormal `WebSocket` connection termination encountered while the live chat feature is open.

## Handling a disconnection event

This leads us to the last function, `HandleDisconnection`, which is called either on an abnormal `WebSocket` connection close event or when the internet connection has been disconnected, that is, when the `wenv.Window` object fires an `offline` event:

```
func HandleDisconnection(env *common.Env) {  
  
    chatContainer :=  
    env.Document.GetElementByID("chatboxContainer").(*dom.HTMLDivElement)  
    chatContainer.SetClass("")  
  
    chatboxHeaderBar :=  
    env.Document.GetElementByID("chatboxHeaderBar").(*dom.HTMLDivElement)  
    chatboxHeaderBar.SetClass("chatboxHeader disconnected")  
  
    chatboxTitleDiv :=  
    env.Document.GetElementByID("chatboxTitle").(*dom.HTMLDivElement)  
    if chatboxTitleDiv != nil {  
        titleSpan := chatboxTitleDiv.ChildNodes()[0].(*dom.HTMLSpanElement)  
        if titleSpan != nil {  
            var countdown uint64 = 6  
            tickerForCountdown := time.NewTicker(1 * time.Second)  
            timerToCloseChat := time.NewTimer(6 * time.Second)  
            go func() {  
                for _ = range tickerForCountdown.C {  
                    atomic.AddUint64(&countdown, ^uint64(0))  
                    safeCountdownValue := atomic.LoadUint64(&countdown)  
                    titleSpan.SetInnerHTML("Disconnected! - Closing LiveChat in " +  
                    strconv.FormatUint(safeCountdownValue, 10) + " seconds.")  
                }  
            }()  
            go func() {  
                <-timerToCloseChat.C  
                tickerForCountdown.Stop()  
                CloseChat(env)  
            }()  
        }()  
    }()  
}
```

```
        }  
    }  
}
```

We first set the CSS `classname` value of the `chatContainer` to empty string, using the `SetClass` method, to disable the `chatContainer` element's pulsate effect to indicate that the connection has been broken.

We then change the background color of `chatboxHeaderBar` to red by setting the `chatboxHeaderBar` element's CSS `classname` value to "chatboxHeader disconnected" using the `SetClass` method.

The remaining code will present the user with a message indicating that the connection has been disconnected, and the live chat feature will automatically initiate a countdown. The `chatboxHeaderBar` will display the countdown, 5-4-3-2-1, by the second, as the live chat feature shuts itself down. We use two goroutines, one for the countdown ticker and the other for the countdown timer. When the countdown timer has expired, it signifies that the countdown is over, and we call the `CloseChat` function passing in the `env` object to close the live chat feature.

## Conversing with the agent

At this point, we have implemented the server-side and client-side functionality to realize the live chat feature, showcasing real-time web application functionality. Now it's time to start up a conversation (question and answer session) with the chat agent.

After clicking on the live chat icon found on the website's top bar, we are presented with the chat box in the lower right-hand side portion of the web page. The following screenshot depicts the chat box with the chat agent's greeting:

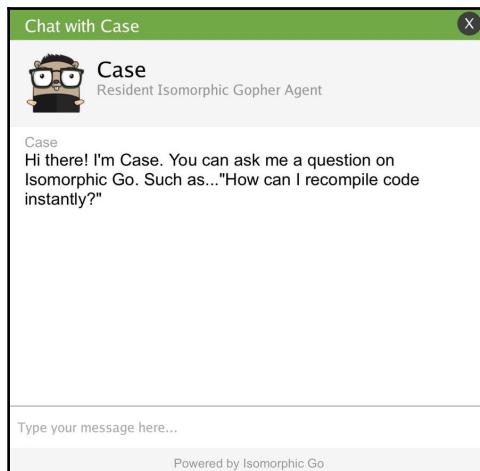


Figure 8.3: The chat box opens up with a greeting from the chat agent

We can close the live chat box window using the **X** control in the top-right corner of the chat box. We can reactivate the live chat feature by clicking on the live chat icon again in the top bar. Instead of asking the chat agent a question such as **What is Isomorphic Go?**, we can actually provide a statement such as **Tell me more about Isomorphic Go** as we have shown in the following screenshot:

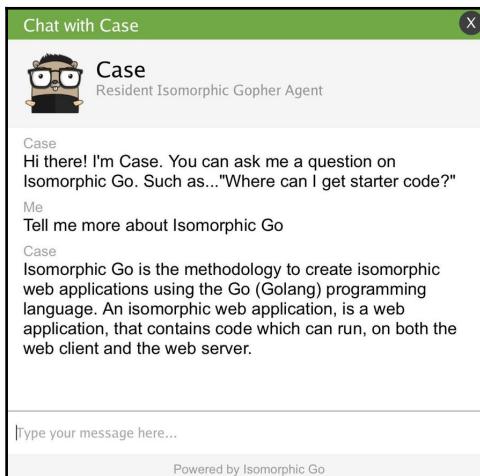


Figure 8.4: The chat agent understands an information request even if it isn't a question

The question and answer session between the human user and the chat agent can continue for as long as the human user wishes, as shown in the next screenshot. This is perhaps the greatest strength of a chat agent—it has unlimited patience when dealing with humans.

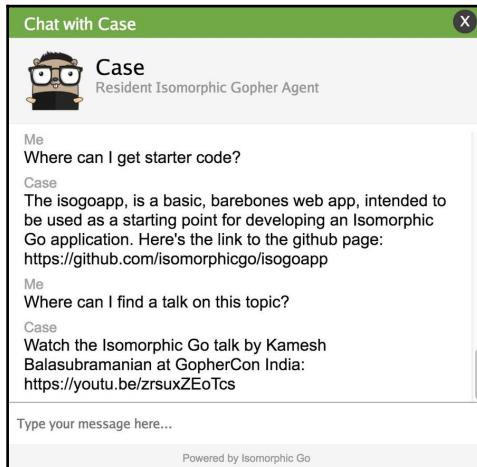


Figure 8.5: The question and answer session can continue as long as the human wants it to

The chat agent we have implemented has an extremely narrow and limited set of intelligence. The chat agent will admit that it doesn't know the answer, when the human user asks a question outside of the scope of its intelligence, as shown here:

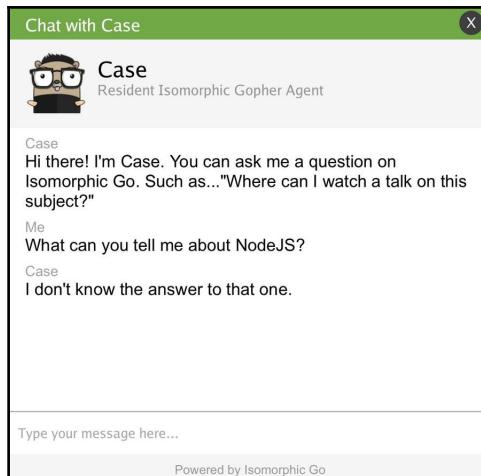


Figure 8.6: The chat agent doesn't have an answer for a question outside the scope of its intelligence

Some human users can be rude to the chat agent. This comes along with the public-facing role that the chat agent serves. If we tune the corpus just right, we can have our chat agent exhibit a witty reply.

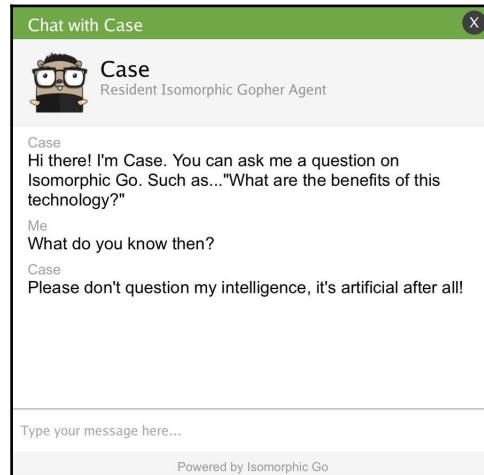


Figure 8.7: The chat agent exhibiting a witty reply

As noted earlier, we have strategically placed the chat box container outside of the primary content area in the web page layout. Having done so, the chat box and the conversation with the chat agent, can be continued, as we freely navigate through the links of IGWEB, as shown here:

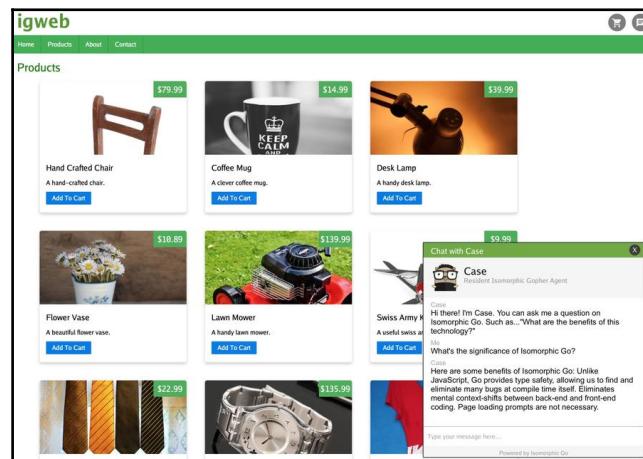


Figure 8.8: The chat conversation will be retained as the user navigates through IGWEB

For example, the chat conversation is continued even after clicking on the **Coffee Mug** product image to get to the product detail page, as shown here:

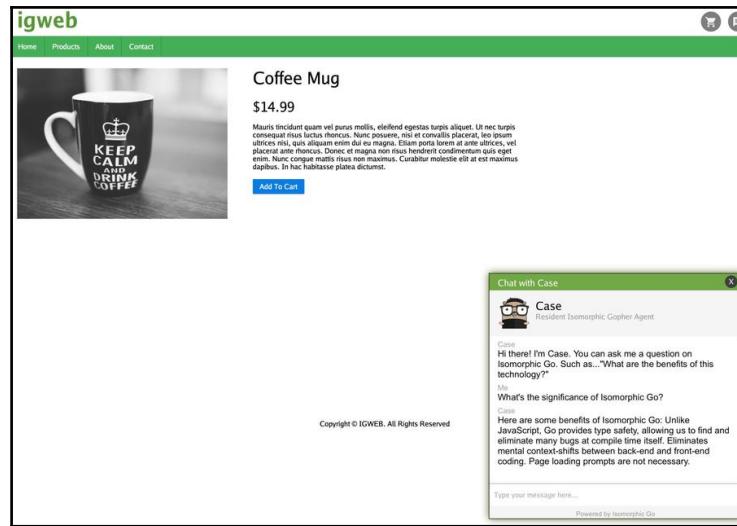


Figure 8.9: The chat conversation has been retained as the user visits the product detail page for the coffee mug

A real-time web application depends on a persistent connection to the Internet. Let's see how the live chat feature gracefully handles the scenario where we disconnect the Internet connection, as shown here:

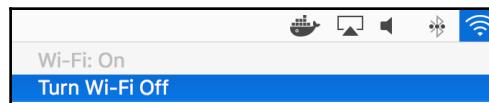


Figure 8.10: Switching off the internet connection

Once the internet connection has been turned off, we immediately are notified of the disconnection in the chat box's header bar as shown in *Figure 8.11*. The background color of the chat box header bar turns red and a countdown to close the live chat feature is initiated. After the countdown is complete, the live chat feature closes itself automatically:

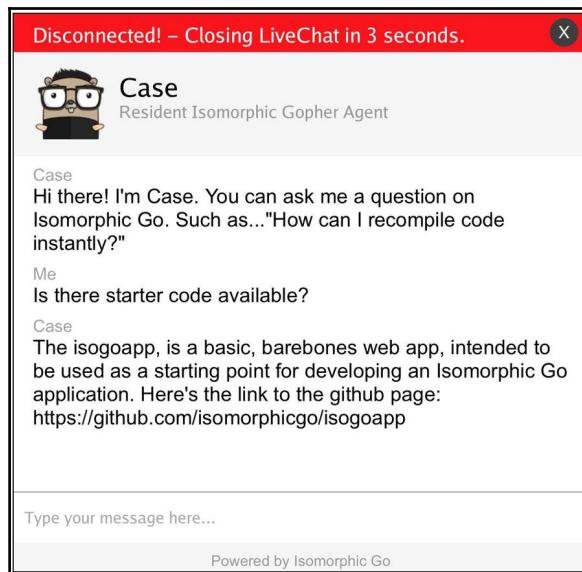


Figure 8.11: The countdown to shutdown the live chat feature appears in the chat box's header bar

When implementing real-time web application functionality, it's always important to consider the scenario of the persistent WebSocket connection being interrupted. By having the live chat close down gracefully, when the persistent connection between the web client and the web server is interrupted, we have a means to provide a *heads up* to the user to disengage with the chat agent.

## Summary

In this chapter, we implemented real-time web application functionality in the form of IGWEB's live chat feature. You learned how to establish a persistent connection between the web server and the web client using a WebSocket. On the server-side, we introduced you to the `websocket` package from the Gorilla toolkit project. On the client-side, we introduced you to the `gopherjs/websocket/websocket.js` package from the GopherJS project.

We created a simple, rudimentary chatbot that answers questions posed by users in real-time, with the conversation between the human and the bot being relayed through the established WebSocket connection. Since the real-time web application functionality depends on the persistent connection, we also added code to automatically shut down the live chat feature in case of an interrupted internet connection.

We used the `nlp` package to implement the brain of the rudimentary chat agent so that it could answer a few questions related to Isomorphic Go. We made our chat agent solution extensible, where new bots of varying intelligence could be added in the future by defining the `Bot` interface.

In Chapter 9, *Cogs—Reusable Components*, we will explore how to implement reusable interface widgets all across IGWEB. Reusable components provide a means to promote greater reusability, and they can be used in a plug and play manner. As you will learn, cogs are also efficient, making use of a virtual DOM to rerender their content as needed.

# 9

## Cogs – Reusable Components

In the previous five chapters of this book, we focused on developing functionality for either a specific web page on IGWEB or a specific feature, such as the live chat feature that we implemented in the last chapter. The solutions that we've made thus far have served a particular individual purpose. Not much consideration was factored into promoting code reuse for a particular user interface feature, since we didn't have the need to create multiple instances of it.

Reusable components are user interface widgets that provide a means to promote greater reusability. They can be used in a plug and play manner, since each component is a standalone user interface widget that contains its own set of Go source files and static assets, such as Go template files, along with CSS and JavaScript source files.

In this chapter, we will focus on creating **cogs**—reusable components that can be utilized in Isomorphic Go web applications. The term **cog** stands for **component object in Go**. Cogs are reusable user interface widgets that can either be implemented exclusively in Go (a **pure cog**) or implemented using Go and JavaScript (a **hybrid cog**).

We can create multiple instances of a **cog** and control the cog's behavior by supplying input parameters (in the form of key-value pairs) to the **cog**, known as **props**. When subsequent changes are made to the **props**, the **cog** is **reactive**, meaning that it can automatically re-render itself. Therefore, cogs have the capability to change their appearance, based on changes made to their **props**.

Perhaps, the most attractive feature of cogs is that they are readily reusable. Cogs are implemented as standalone Go packages that contain one or more Go source files along with any static assets that are needed by the cog's implementation.

In this chapter, we will cover the following topics:

- Essential cog concepts
- Implementing pure cogs
- Implementing hybrid cogs

## Essential cog concepts

Cogs (component objects in Go) are reusable components that are implemented in Go. The guiding philosophy behind cogs is to allow developers to create reusable components on the front-end in an idiomatic manner. Cogs are self-contained defined as their own Go package, which makes it easy to reuse and maintain them. Due to their self-contained nature, cogs can be used to create composable user interfaces.

Cogs follow a clear separation of concerns where the presentation layer of a cog is implemented using one or more Go templates, and the controller logic of the cog is implemented in one or more Go source files contained within a Go package. These Go source files may import Go packages from the standard library or third-party libraries. We'll see an example of this when we implement the time ago cog in the *Implementing pure cogs* section of this chapter.

Cogs may also have CSS stylesheet and JavaScript code associated with them, allowing cog developers/maintainers to leverage prebuilt JavaScript solutions as needed rather than porting a JavaScript widget directly to Go. This makes cogs interoperable with existing JavaScript solutions and prevents situations where the developer can save valuable time, by not having to reinvent the proverbial wheel. For example, Pikaday (<https://github.com/dbushell/Pikaday>) is a well-established calendar date picker JavaScript widget. In the *Implementing hybrid cogs* section of this chapter, we will learn how to implement a date picker cog that utilizes the functionality that the Pikaday JavaScript widget provides. Go developers that use the date picker cog need not have any knowledge of JavaScript, and can utilize it solely with their knowledge of Go.

Each cog comes with a **virtual DOM tree**, an in-memory representation of its actual DOM tree. It is far more efficient to manipulate the cog's in-memory virtual DOM tree rather than manipulating the actual DOM tree itself. *Figure 9.1* is a Venn diagram depicting a cog's virtual DOM tree, the difference between the two trees, and the actual DOM tree:

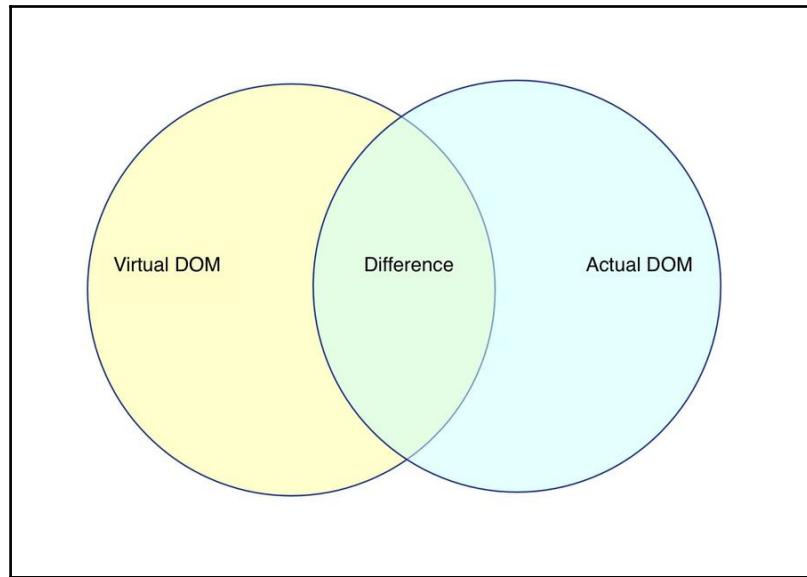


Figure 9.1: A Venn diagram depicting the virtual DOM, the difference, and the actual DOM

As changes are made to a cog's properties (*props*), the cog's rendering engine will utilize its virtual DOM tree to determine the changes and then reconcile the changes with the actual DOM tree. This allows the cog to be *reactive*, meaning that the cog can automatically re-render itself the moment that one of its props gets updated. In this manner, cogs reduce the complexity that is involved when updating the user interface.

## The UX toolkit

The UX toolkit provides the functionality to implement cogs within the cog package, which can be installed using the following go get command:

```
$ go get -u github.com/uxtoolkit/cog
```

All cogs must implement the `Cog` interface:

```
type Cog interface {
    Render() error
    Start() error
}
```

The `Render` method is responsible for rendering the `cog` on the web page. If there are any errors in the rendering process, the method will return an `error` object.

The `Start` method is responsible for activating the `cog`. If the `cog` was not able to start, the method will return an `error` object.

The `cog` package contains two important exported variables, `ReactivityEnabled` and `VDOMEnabled`. Both of these exported variables are of type `bool`, and by default, they are both set to `true`.

When the variable `ReactivityEnabled` is set to `true`, cogs will be re-rendered as changes are made to their props. If `ReactivityEnabled` is set to `false`, then the cog's `Render` method must explicitly be called to re-render the cog.

When the variable `VDOMEnabled` is set to `true`, cogs will be rendered utilizing the cog's virtual DOM tree. If `VDOMEnabled` is set to `false`, the cog will be rendered using the actual DOM tree through a replace inner HTML operation. This can be an expensive operation which can be avoided by utilizing the cog's virtual DOM tree.

The `UXCog` type implements the `Render` method of the `Cog` interface. Here is what the `UXCog` struct looks like:

```
type UXCog struct {
    Cog
    cogType reflect.Type
    cogPrefixName string
    cogPackagePath string
    cogTemplatePath string
    templateSet *isokit.TemplateSet
    Props map[string]interface{}
    element *dom.Element
    id string
    hasBeenRendered bool
    parseTree *reconcile.ParseTree
    cleanupFunc func()
}
```

The `UXCog` type provides the basic functionality to make cogs work. That means in order to implement our own cogs, we must type embed `UXCog` in the type definition of all cogs that we create. The following methods (for brevity, only the method signatures are presented) of the `UXCog` type are of particular interest to us:

```
func (u *UXCog) ID() string  
  
func (u *UXCog) SetID(id string)  
  
func (u *UXCog) CogInit(ts *isokit.TemplateSet)  
  
func (u *UXCog) SetCogType(cogType reflect.Type)  
  
func (u *UXCog) SetProp(key string, value interface{})  
  
func (u *UXCog) Render() error
```

The `ID` method is a getter method that returns the ID of the cog's `div` container in the DOM. A cog's `div` container is known as its **mount point**.

The `SetID` method is a setter method that is used to set the ID of the cog's `div` container in the DOM.

The `CogInit` method is used to associate the `cog` to the application's `TemplateSet` object. There are two significant purposes served for this method. First, the method is used to register a `cog` on the server-side, so that all of the templates for a given `cog` are included in the template bundle that is produced by the static assets bundling system built into `isokit`. Second, calling the cog's `CogInit` method on the client-side provides the `cog` access to the client-side application's `TemplateSet` object, allowing the `cog` to render itself on the web page.

The `SetCogType` method allows us to dynamically set the cog's type by performing runtime reflection on a newly instantiated `cog`. This provides the hook needed by `isokit`'s static assets bundling system to bundle the template files, CSS source files, and JavaScript source files associated for a given `cog`.

The `SetProp` method is used to set a key-value pair in the cog's `Props` map, which is of type `map[string]interface{}`. The key of the map represents the name of the prop, and the value represents the value of the prop.

The `Render` method is responsible for rendering the `cog` to the DOM. If a change is made to the `cog` (its prop value is updated) after it has been rendered, the `cog` will be re-rendered.



You can visit the UX toolkit website to learn more about cogs: <http://uxtoolkit.io>.

Now that we have been acquainted with the `UXCog` type, it's time to examine the anatomy of a `cog`.

## The anatomy of a cog

For the `IGWEB` project, we will be creating cogs in the `$IGWEB_APP_ROOT/shared/cogs` folder. You can take a peek at the `timeago` cog, whose implementation is found in the `$IGWEB_APP_ROOT/shared/cogs/timeago` folder, as you read through this section, to see a tangible implementation of the concepts presented herein.

For the purposes of illustration only, we are going to walk you through the process of creating a simple `cog` called `widget`.

The project structure for the `widget` `cog` contained within the `widget` folder is organized in the following manner:

- `widget`
  - `widget.go`
  - `templates`
  - `widget tmpl`

The `widget.go` source file will contain the implementation for the `widget` `cog`.

The `templates` folder contains the template source file(s) used to implement the `cog`. If the `cog` is to be rendered on the web page, at least one template source file must be present. The template source file's name must match the package name of the `cog`. For example, for the `cog` package `widget`, the name of the template source file must be `widget tmpl`.

Cogs follow a *convention over configuration* strategy when it comes to naming package names and source files. Since we have chosen the name `widget` we must declare a Go package named `widget` as well inside the `widget.go` source file:

```
package widget
```

All cogs are required to include the `errors` package, the `reflect` package, and the `cog` package in their import grouping:

```
import (
    "errors"
    "reflect"
    "github.com/uxtoolkit/cog"
)
```

We must declare an un-exported, package scoped variable called `cogType`:

```
var cogType reflect.Type
```

This variable represents the cog's type. We call the `TypeOf` function in the `reflect` package, passing in a newly created instance of the `cog`, to dynamically set the cog's type inside the `cog` package's `init` function:

```
func init() {
    cogType = reflect.TypeOf(Widget{})
}
```

This provides a hook for isokit's static bundling system, to know where to look, to obtain the static assets required to make a `cog` function.

A `cog` implements a specific type. In the case of a widget, we implement the `Widget` type. Here's the `Widget` struct:

```
type Widget struct {
    cog.UXCog
}
```

We must type embed the `cog.UXCog` type to bring all the functionality needed from the `cog.UxCog` type in order to implement the `cog`.

The `struct` may contain other field definitions that are required to implement the `cog`, depending on the purpose that the `cog` serves.

Every `cog` implementation should include a constructor function:

```
func NewWidget() *Widget {
    w := &Widget{}
    w.SetCogType(cogType)
    return f
}
```

As with any typical constructor function, the purpose is to create a new instance of the `cog`, `Widget`.

The `cog`'s constructor function must contain the line that calls the `SetCogType` method (shown in bold). This is used as a hook by isokit's automatic static assets bundling system to bundle the `cog`'s required static assets.

Additional fields of the `Widget` type may be set to initialize the `cog` based on the `cog`'s implementation.

In order to fulfill the implementation of the `Cog` interface, all `cogs` must implement a `Start` method:

```
func (w *Widget) Start() error {  
    var allRequiredConditionsHaveBeenCalled bool = true
```

The `Start` method is responsible for activating the `cog`, which includes the initial render of the `cog` to the web page. The `Start` method will return an `error` object, if the `cog` failed to start, otherwise, a `nil` value will be returned.

For illustration purposes, we have defined an `if` conditional block containing a Boolean variable called `allRequiredConditionsHaveBeenCalled`:

```
if allRequiredConditionsHaveBeenCalled == false {  
    return errors.New("Failed to meet all requirements, cog failed to  
    start!")  
}
```

If all the conditions to start the `cog` have been met, this variable will be equal to `true`. Otherwise, it will be equal to `false`. If it is `false`, then we will return a new `error` object, indicating that the `cog` was unable to start since all requirements had not been met.

We can set a key-value pair in a `cog`'s `Props` map by calling the `SetProp` method:

```
w.SetProp("foo", "bar")
```

In this case, we have set the prop named `foo` to the value `bar`. The `Props` map will automatically be used as the data object that gets fed into a `cog`'s template. This means that all props defined in the `Props` map are accessible by the `cog`'s template.

The cog's template source file name, by convention, must be named `widget tmpl` to match the cog's package name of `widget`, and the template file should reside in the `templates` folder, which is located in the cog's folder, `widget`.

Let's take a quick look at what the `widget tmpl` source file may look like:

```
<p>Value of Foo is: {{.foo}}</p>
```

Notice that we are able to print out the value of the prop that has a key of `foo` within the template.

Let's return back to the `widget` cog's `Start` method. We call the cog's `Render` method to render the cog in the web browser:

```
err := w.Render()  
if err != nil {  
    return err  
}
```

The `Render` method returns an `error` object if an error was encountered while rendering a cog, otherwise it will return a value of `nil` to indicate that the cog was rendered successfully.

If the cog was rendered successfully, the cog's `Start` method returns a value of `nil` to indicate that the cog has been started successfully:

```
return nil
```

In order to render our cog to the real DOM, we need a place to render the cog to. The `div` container that houses the rendered content of a cog is known as its **mount point**. The mount point is where the cog gets rendered to in the DOM. To render the `widget` cog on the home page, we would add the following markup to the home page's content template:

```
<div data-component="cog" id="widgetContainer"></div>
```

By setting the `data-component` attribute to "cog", we indicate that the `div` element is meant to be used as a cog's mount point, and the cog's rendered content will be contained inside this element.

In the client-side application, the widget `cog` can be instantiated like so:

```
w := widget.NewWidget()
w.CogInit(env.TemplateSet)
w.SetID("widgetContainer")
w.Start()
w.SetProp("foo", "bar2")
```

We create a new `Widget` instance and assign it to the variable `w`. We must call the `CogInit` method of the `cog` to associate the application's `TemplateSet` object with the `cog`. The `cog` utilizes the `TemplateSet` so that it may fetch it's associated template(s), which are required to render the `cog`. We call the `cog`'s `SetID` method, passing in the `id` to the `div` element that acts as the `cog`'s mount point. We call the `cog`'s `Start` method to activate the `cog`. Since the `Start` method calls the `cog`'s `Render` method, the `cog` will be rendered in the designated mount point, the `div` element with `id`, `"widgetContainer"`. Finally, when we call the `SetProp` method and change the value of the `"foo"` prop to `"bar2"`, the `cog` will get re-rendered automatically.

Now that we've examined the basic anatomy of a `cog`, let's consider how cogs are rendered using a virtual DOM.

## The virtual DOM tree

Each `cog` instance has a virtual DOM tree associated with it. This virtual DOM tree is a parse tree comprised of all the children of the `cog`'s `div` container.

*Figure 9.2* is a flowchart depicting the process to render, and re-render (through the application of reconciliation) the `cog` to the DOM:

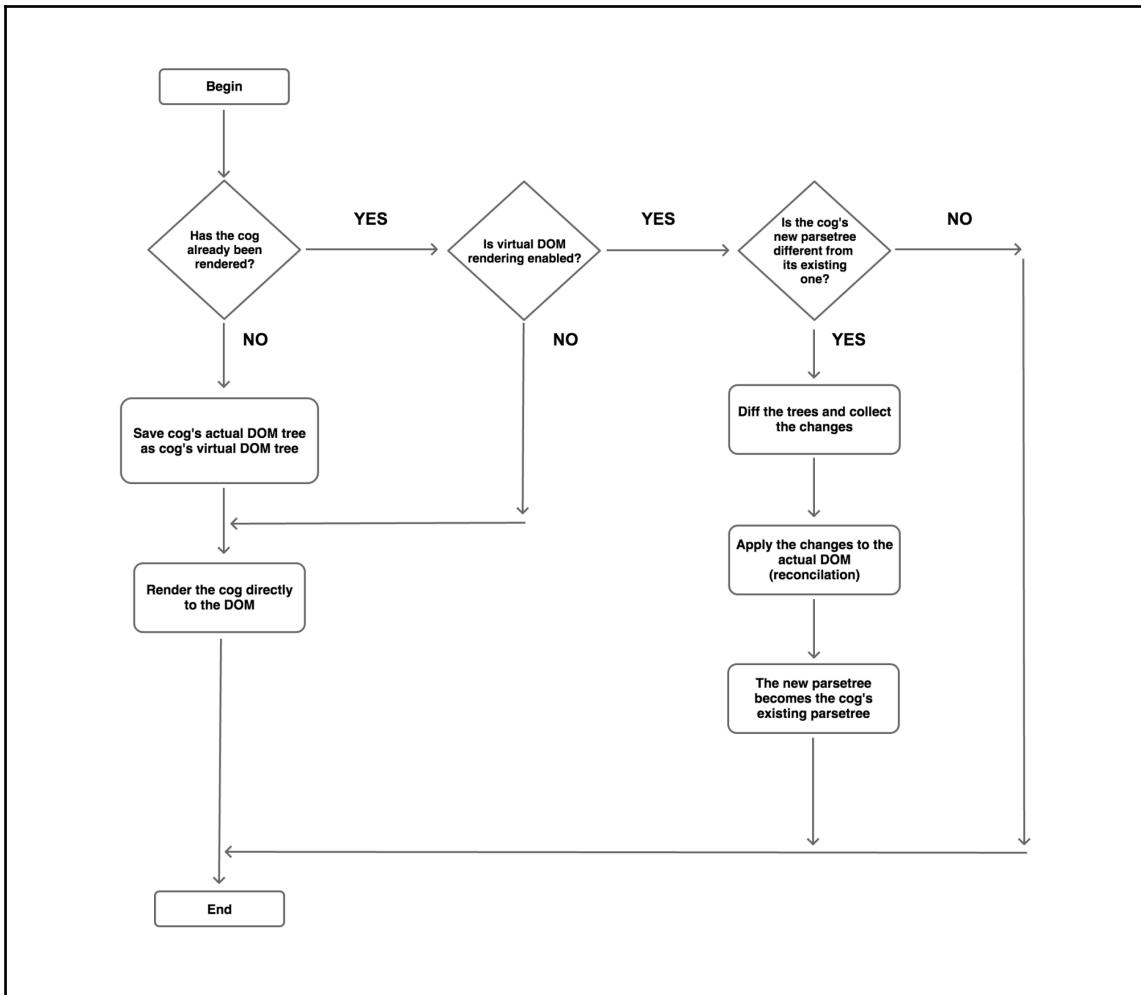


Figure 9.2: A flowchart depicting the process to render and re-render a cog

When the `cog` is first rendered in the DOM, a replace inner HTML operation is performed. The operation to replace the inner HTML contents of an element in the DOM is an expensive operation. Hence, it is not performed on subsequent renders of the `cog`.

All subsequent calls to the cog's `Render` method will utilize the cog's virtual DOM tree. The cog's virtual DOM tree is used to keep track of changes between the cog's current virtual DOM tree and the cog's new virtual DOM tree. A cog will have a new virtual DOM tree to compare against its current one when a cog's prop value has been updated.

Let's consider an example scenario with the widget cog. Calling a widget cog's `Start` method will perform the initial render of the cog (since the cog's `Render` method is called within the `Start` method). The cog will have a virtual DOM tree that will be the parse tree of the `div` container holding the cog's rendered content. If we were to update the "`foo`" prop (which is rendered in the cog's template) by calling the `SetProp` method on the cog, then the `Render` method will automatically be called, since the cog is reactive. Upon performing the subsequent render operation on the cog, the cog's current virtual DOM tree will be diffed against the cog's new virtual DOM tree (the virtual DOM tree that is created after updating the cog's prop).

If there are no changes between the current virtual DOM tree and the new virtual DOM tree, there is no need to perform any operation. However, if there are differences between the current virtual DOM tree and the new virtual DOM tree, then we must apply the changes that constitute the difference to the actual DOM. The process of applying these changes is known as **reconciliation**. Performing reconciliation allows us to avoid performing an expensive replace inner HTML operation. Upon successful application of reconciliation, the cog's new virtual DOM tree will be considered as the cog's current virtual DOM tree to prepare the cog for the next render cycle:

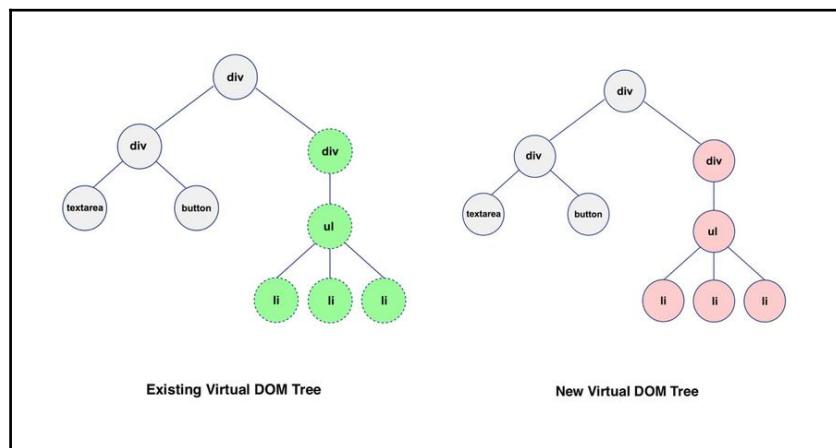


Figure 9.3: A cog's existing virtual DOM tree (left) and the cog's new virtual DOM tree (right)

Figure 9.3 depicts a cog's existing virtual DOM tree on the left and the cog's new virtual DOM tree on the right. After performing a `diff` operation on the two virtual DOM trees (new and existing), it is determined that the rightmost `div` element (which contains the `ul` element) and its children have changed, and the reconciliation operation would only update the `div` element and its children in the actual DOM.

## The life cycle of a cog

Figure 9.4 depicts the life cycle of a cog, which starts on the server-side, where we first register the cog. The cog's type must be registered on the server-side so that the cog's associated template(s), along with other static assets can be automatically bundled and made available to the client-side application:

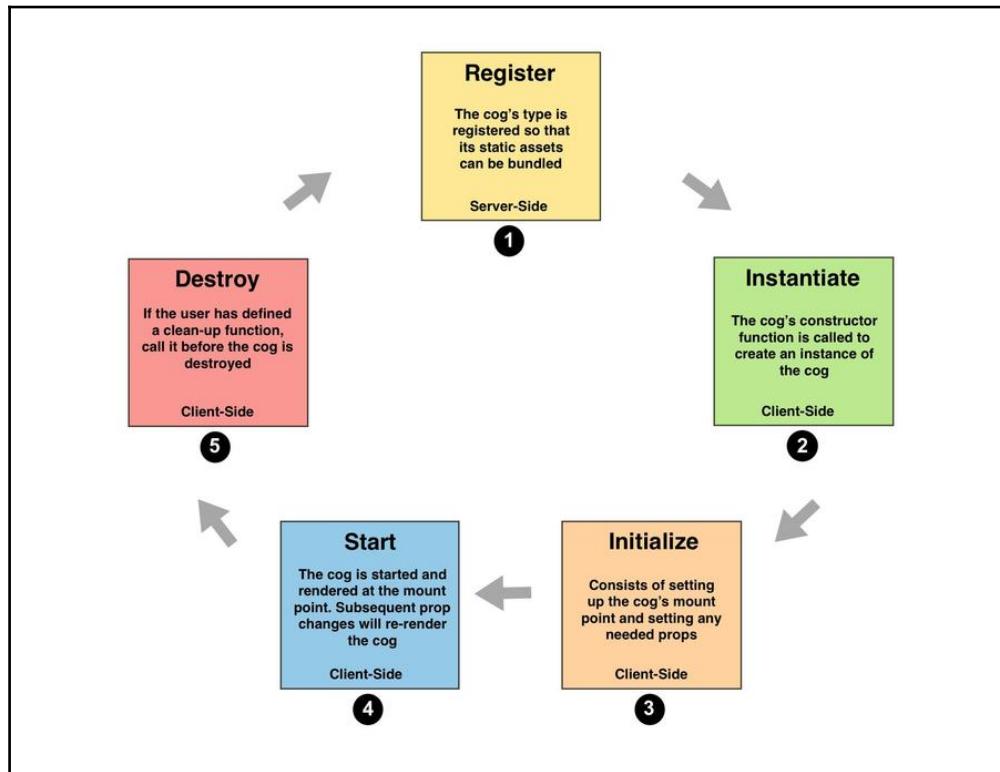


Figure 9.4: The life cycle of a cog

The subsequent steps in the `cog` life cycle, take place on the client-side. We declare a mount point for the `cog`, by introducing a `div` element with a `data-component` attribute equal to `"cog"`, to indicate that the `div` element is a mount point for a `cog`.

The next step, is to create a new instance of the `cog` by calling its constructor function. We initialize the `cog` by calling its `CogInit` method and pass in the `TemplateSet` object of the client-side application. Initializing the `cog` also consists of calling the `cog`'s `SetID` method to associate the mount point to the `cog` (so that the `cog` knows where to render to). `Cog` initialization also includes setting a prop in the `cog`'s `Props` map by calling its `SetProp` method before calling the `Start` method.

Note that calling a `cog`'s `SetProp` method, before its `Start` method is called, will not render the `cog`. A `cog` will re-render upon calling its `SetProp` method only after a `cog` has been rendered to the mount point, by calling its `Start` method.

Calling the `Cog`'s `Start` method will activate the `cog` and render the `cog`'s contents to the specified mount point.

Any subsequent calls made to the `cog`'s `SetProp` method will result in a re-render of the `cog`.

When the user navigates to a different page on the website, the container that the `cog` resides in is removed, effectively destroying the `cog`. The user may specify a clean-up function that should be called just before the `cog` is destroyed. This can come in handy to free up resources in a responsible manner prior to the `cog`'s destruction. We'll see an example of implementing a clean-up function later in this chapter.

## Implementing pure cogs

Now that we have a basic understanding of cogs, its time to implement a few cogs in practice. Although cogs operate on the client-side, it is important to note that the server-side application needs to acknowledge their existence by registering them. Code for cogs are strategically placed in the `shared/cogs` folder for this reason.

Pure cogs are implemented exclusively in Go. As you will see, we can leverage functionality from existing Go packages to implement cogs.

Inside the main function in the `igweb.go` source file, we call the `initializeCogs` function passing in the application's template set:

```
initializeCogs(env.TemplateSet)
```

The `initializeCogs` function is responsible for initializing all the cogs that are going to be used in the Isomorphic Go web application:

```
func initializeCogs(ts *isokit.TemplateSet) {
    timeago.NewTimeAgo().CogInit(ts)
    liveclock.NewLiveClock().CogInit(ts)
    datepicker.NewDatePicker().CogInit(ts)
    carousel.NewCarousel().CogInit(ts)
    notify.NewNotify().CogInit(ts)
    isokit.BundleStaticAssets()
}
```

Note that the `initializeCogs` function takes a sole input argument, `ts`, the `TemplateSet` object. We call the cog's constructor function to create a new instance of the cog, and immediately call the `CogInit` method of the cog, passing in the `TemplateSet` object, `ts`, as an input argument to the method. This allows the cog to include its templates to the application's template set, so that the subsequent template bundle that is to be produced will include templates associated with the cog.

We call the `BundleStaticAssets` method to generate the static assets (CSS and JavaScript source files) that are required for each cog. Two files will be produced. The first file is `cogimports.css`, which will contain the CSS source code needed for all the cogs, and the second file is `cogimports.js`, which will contain the JavaScript source code needed for all of the cogs.

## The time ago cog

Now that we have seen how the cogs are initialized on the server-side, it's time to take a look at what goes into making a cog. We're going to start out by making a very simple cog, a time ago cog, which displays time in human understandable format.

It's time revisit the gopher bios on the about page. In the section *Custom template functions* found in [Chapter 3, Go on the Front-End with GopherJS](#), we learned how to use a custom template function to display the gopher's start date-time in Ruby format.

We are going to go a step further and display the start date-time in human understandable format by implementing a time ago cog. *Figure 9.5* is an illustration showing the start date for Molly in the default Go format, in Ruby format, and in human understandable format:

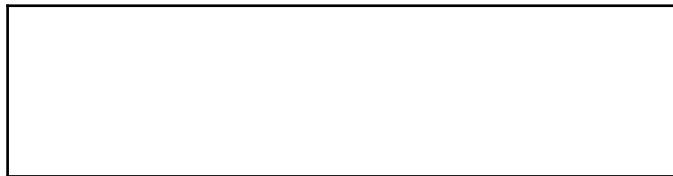


Figure 9.5: Illustration depicting the time ago cog, which is the last line showing the time in human readable format

Molly joined the IGWEB team on May 24, 2017, which in human readable format is 6 months ago (at the present time of writing).

In the `about_content tmpl` template source file, we introduce a `div` container for the time ago cog:

```
<h1>About</h1>

<div id="gopherTeamContainer">
  {{range .Gophers}}
    <div class="gopherContainer">
      <div class="gopherImageContainer">
        
      </div>

      <div class="gopherDetailsContainer">
        <div class="gopherName"><h3><b>{{.Name}}</b></h3></div>
        <div class="gopherTitle"><span>{{.Title}}</span></div>
        <div class="gopherBiodata"><p>{{.Biodata}}</p></div>
        <div class="gopherStartTime">
          <p class="standardStartTime">{{.Name}} joined the IGWEB team on
          <span class="starttime">{{.StartTime}}.</p>
          <p class="rubyStartTime">That's <span
          class="starttime">{{.StartTime | rubyformat}}</span> in Ruby date
          format.</p>
          <div class="humanReadableGopherTime">That's
          <div id="Gopher-{{.Name}}" data-starttimeunix="{{.StartTime |
          unixformat}}" data-component="cog" class="humanReadableDate
          starttime"></div>
          in Human readable format.
        </div>
      </div>
    </div>
  {{end}}
</div>
```

```
        </div>
    </div>
</div>

{{end}}
</div>
```

Notice that we've assigned the attribute, named `data-component`, with a value of `cog`. This is to indicate that this `div` container will serve as a mount point, housing the rendered content of a `cog`. We set the `id` attribute of the container to the first name of the Gopher with a prefix of "Gopher-".

Later you will see that when we instantiate a `cog`, we must supply a `cog`'s `div` container with an ID, so that the `cog` instance knows where its mount point is the place that the `cog` should render its output to. We define another custom data attribute, `starttimeunix`, and we set it to the Unix timestamp value of when the Gopher started working for IGWEB.

Recall that the value is obtained from calling the template action, which places the value obtained by pipelining the `StartTime` property to the custom template function, `unixformat`.

The `unixformat` custom template function is an alias to the `UnixTime` function defined in the `shared/templatefuncs/funcs.go` source file:

```
func UnixTime(t time.Time) string {
    return strconv.FormatInt(t.Unix(), 10)
}
```

This function will return the time as a `string` value, in Unix format, for a given `Time` instance.

Returning back to the `about_content tmpl` source file, take note of the `humanReadableDate` CSS `className` provided to the `div` container. We'll be using this CSS `className` later to fetch all the `timeago` cog `div` containers on the **About** page.

Now that we've seen how to declare the `cog`'s `div` container on the **About page**, let's take a look at how to implement the time ago `cog`.

The time ago `cog` is a pure Go `cog`. That means it's implemented using only Go. The Go package, `go-humanize`, provides us the functionality we need to display the time in human readable format. We are going to utilize this package to implement the time ago `cog`. Here's the URL to the GitHub page for the `go-humanize` package: <https://github.com/dustin/go-humanize>.

Let's examine the `shared/cogs/timeago/timeago.go` source file. We start out by declaring the package name as `timeago`:

```
package timeago
```

In our import grouping, we include `github.com/uxtoolkit/cog`, the package that provides us with functionality to implement a `cog` (shown in bold). We include the `go-humanize` grouping in our import grouping and alias it with the name "`humanize`" (shown in bold):

```
import (
    "errors"
    "reflect"
    "time"

humanize "github.com/dustin/go-humanize"
    "github.com/uxtoolkit/cog"
)
```

All cogs must declare an unexported variable called `cogType`, which is of type `reflect.Type`:

```
var cogType reflect.Type
```

Inside the `init` function, we assign the `cogType` variable with the value returned by calling the `reflect.TypeOf` function on a newly created `TimeAgo` instance:

```
func init() {
    cogType = reflect.TypeOf(TimeAgo{})
}
```

Initializing the `cogType` variable is also required for every `cog` that we implement. Properly setting the `cogType` allows the static assets bundling system to account for the `cog`'s static assets dependencies in the web application. The `cogType` will be utilized to gather all of the templates and static assets that are required to make the `cog` function.

Here's the struct we use to define the `TimeAgo` `cog`:

```
type TimeAgo struct {
    cog.UXCog
    timeInstance time.Time
}
```

Take note that we have embedded `ux.UXCog` in our struct definition. As noted earlier, the `cog.UXCog` type will provide us the necessary functionality to allow us to render the `cog`. Besides embedding the `ux.UXCog`, we have declared an unexported field, called `timeInstance`, of type `time.Time`. This will contain the `time.Time` instance that we will be converting to human readable format.

We create a constructor function called `NewTimeAgo` that returns a new `TimeAgo` cog instance:

```
func NewTimeAgo() *TimeAgo {
    t := &TimeAgo{}
    t.SetCogType(cogType)
    return t
}
```

The constructor function that we have here, follows the same pattern as any other constructor function implemented in Go. Notice that we pass the `cogType` to the `SetCogType` method of the newly created `TimeAgo` instance. This is required so that the the `cog`'s static assets are included in the static assets bundle that is produced by isokit's static asset bundling system.

We create a setter method for the `TimeAgo` struct's `timeInstance` field called `SetTime`:

```
func (t *TimeAgo) SetTime(timeInstance time.Time) {
    t.timeInstance = timeInstance
}
```

This setter method will be used by the client-side application to set the time for the `TimeAgo` cog. We will use the `SetTime` method to set the start date of when the gopher joined the IGWEB team.

In order to implement the `Cog` interface, a `cog` must define a `Start` method. The `Start` method is where the action in a `cog` happens. You should be able to get a general idea of what a `cog` does by reading its `Start` method. Here's the `Start` method for the `TimeAgo` cog:

```
func (t *TimeAgo) Start() error {

    if t.timeInstance.IsZero() == true {
        return errors.New("The time instance value has not been set!")
    }

    t.SetProp("timeAgoValue", humanize.Time(t.timeInstance))

    err := t.Render()
}
```

```
if err != nil {  
    return err  
}  
  
return nil  
}
```

The `Start` method returns an error object to inform the caller whether or not the `cog` started up properly. Prior to performing any activity, a check is made to see if a `timeInstance` value has been set. We use an `if` conditional statement to check if the `timeInstance` value is at its zero value, indicating that it has not been set. If this condition occurs, the method returns a newly created `error` object indicating that the time value has not been set. If the `timeInstance` value has been set, we continue forward.

We call the `cog`'s `SetProp` method to set the `timeAgoValue` property with the human understandable time value. We get the human understandable time value by calling the `Time` function from the `go-humanize` package (aliased as `humanize`) and passing it the `cog`'s `timeInstance` value.

We call the `cog`'s `Render` method to render the `cog`. If an error occurred while attempting to render the `cog`, the `Start` method will return the `error` object. Otherwise, a value of `nil` will be returned to indicate that there was no error starting the `cog`.

At this point, we have implemented the Go portion of the `timeago` cog. In order to make the human readable time appear on the web page, we have to implement the `cog`'s template.

The `timeago.tpl` file (found in the `shared/cogs/timeago/templates` directory) is a simple, one-liner template. We declare the following `span` element, and we have a template action to render the `timeAgoValue` property:

```
<span class="timeagoSpan">{{.timeAgoValue}}</span>
```

By convention, we must name the primary template of a `cog` found in the `cog` package's `templates` folder with the same name of the `cog`'s package. For example, for the `timeago` package, the primary template of the `cog` will be `timeago.tpl`. You are free to define and use any custom template function that has been registered with the application's template set along with the `cog` templates. You are also free to create any number of sub-templates that will be called by the `cog`'s primary template.

Now that we have the template for the `TimeAgo` cog in place, we have everything we need to instantiate the `cog` on the **About** page.

Let's examine the `InitializeAboutPage` function in the `client/handlers/about.go` source file:

```
func InitializeAboutPage(env *common.Env) {
    humanReadableDivs :=
env.Document.GetElementsByClassName("humanReadableDate")
    for _, div := range humanReadableDivs {
        unixTimestamp, err := strconv.ParseInt(div.GetAttribute("data-
starttimeunix"), 10, 64)
        if err != nil {
            log.Println("Encountered error when attempting to parse int64 from
string:", err)
        }
        t := time.Unix(unixTimestamp, 0)
        humanTime := timeago.NewTimeAgo()
        humanTime.CogInit(env.TemplateSet)
        humanTime.SetID(div.ID())
        humanTime.SetTime(t)
        err = humanTime.Start()
        if err != nil {
            println("Encountered the following error when attempting to start the
timeago cog: ", err)
        }
    }
}
```

Since there are three gophers listed on the **About** page, there will be a total of three `TimeAgo` cog instances running on the page. We gather the `div` containers for the cogs using the `GetElementByClassName` method on the `env.Document` object, supplying a class name of `humanReadableDate`. We then loop through each `div` element, and this is where all the action to instantiate the `cog` takes place.

First, we extract the Unix timestamp value from the custom `data` attribute contained in the `div` container. Recall that we had populated the `starttimeunix` custom `data` attribute with the Unix timestamp of the Gopher's start time, using the custom template function, `unixformat`.

We then create a new `time.Time` object using the `Unix` function available in the `time` package and providing the `unixTimestamp` we extracted from the custom `data` attribute of the `div` container. The code to instantiate and set up the `TimeAgo` cog is shown in bold. We first instantiate a new `TimeAgo` cog by calling the constructor function, `NewTimeAgo`, and assigning it to the `humanTime` variable.

We then call the `CogInit` method on the `humanTime` object and supply it with the `env.TemplateSet` object. We call the `SetID` method to register the `div` container's `id` attribute to associate it with the `cog` instance. We then call the `SetTime` method on the `TimeAgo cog`, passing in the `time.Time` object, `t`, that we had created using the `unixTimestamp` we extracted from the `div` container.

We have everything in place now to start up the `cog` by calling its `Start` method. We assign the `error` object returned by the `Start` method to `err`. If `err` is not equal to `nil`, it indicates that an error occurred while starting up the `cog`, and in that case we print out a meaningful message in the web console. If there were no errors, the `cog` will be rendered to the web page. *Figure 9.6* shows a screenshot of Molly's start time in human readable format.



Molly joined the IGWEB team on 2017-05-24 17:09:00 +0000 UTC.

That's Wed May 24 17:09:00 +0000 2017 in Ruby date format.

That's 6 months ago in Human readable format.

Figure 9.6: The time ago cog in action

## The live clock cog

When we called the `Start` method on the time ago `cog`, the time was rendered on the web page using the virtual DOM, instead of a replace inner HTML operation taking place. Since the time ago `cog`, only updates the time once, upon calling the `Start` method of the `cog`, it's hard to appreciate the `cog`'s virtual DOM in action.

In this example, we're going to build a live clock `cog`, which has the ability to display the current time of any place in the world. Since we'll be displaying the time to the seconds, we'll be performing a `SetProp` operation every second to re-render the live clock `cog`.

*Figure 9.7* is an illustration of the live clock:

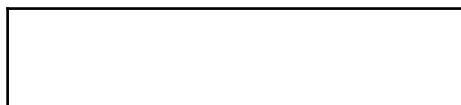


Figure 9.7: An illustration depicting the live clock cog

We'll be rendering the current time for four places: where you are presently located, Chennai, Singapore, and Hawaii. Inside the `shared/templates/index_content.tpl` template source file, we declare four `div` containers that serve as the mounting points for the four live clock cogs we'll be instantiating:

```
<div data-component="cog" id="myLiveClock" class="liveclockTime"></div>
<div data-component="cog" id="chennaiLiveClock"
class="liveclockTime"></div>
<div data-component="cog" id="singaporeLiveClock"
class="liveclockTime"></div>
<div data-component="cog" id="hawaiiLiveClock"
class="liveclockTime"></div>
```

Notice again that we have defined the mount points for the live clocks by declaring `div` containers containing the attribute, `"data-component"`, and having its value set to `"cog"`. We assign unique IDs to all four `cog` containers. The class name, `liveclockTime`, that we have declared in the `div` container is for styling purposes.

Now that we've set up the mounting points for the four live clock cogs, let's take a look at how to implement the live clock cog.

The implementation for the live clock Cog can be found in the `liveclock.go` source file in the `shared/cogs/liveclock` folder.

We declare the name `liveclock` for the cog's package name:

```
package liveclock
```

Notice, that in our import grouping we have included the `github.com/uxtoolkit/cog` package:

```
import (
    "errors"
    "reflect"
    "time"
    "github.com/uxtoolkit/cog"
)
```

We define the `cogType` unexported package variable:

```
var cogType reflect.Type
```

Inside the `init` function, we assign the `cogType` variable with the value returned by calling the `reflect.TypeOf` function on a newly created `LiveClock` instance:

```
func init() {
    cogType = reflect.TypeOf(LiveClock{})
}
```

This is a required step to implement a `cog`.

At this point, we've established that declaring and initializing the `cogType` of a `cog` are part of the baseline requirements that we must perform to implement a `cog`.

Here's what the struct for the `LiveClock` cog looks like:

```
type LiveClock struct {
    cog.UXCog
    ticker *time.Ticker
}
```

We embed the `cog.UXCog` type in the `cog`'s struct definition. We introduce a `ticker` field which is a pointer to a `time.Ticker`. We'll be using this `ticker` to tick at every second for the live clock.

Here's the `LiveClock` cog's constructor function:

```
func NewLiveClock() *LiveClock {
    liveClock := &LiveClock{}
    liveClock.SetCogType(cogType)
    liveClock.SetCleanupFunc(liveClock.Cleanup)
    return liveClock
}
```

The `NewLiveClock` function serves as the constructor function for the live clock `cog`. We declare and initialize the `liveClock` variable to a new `LiveClock` instance. We call the `SetCogType` method of the `liveClock` object and pass the `cogType`. Recall that this is a required step (shown in bold) that must be present in a `cog`'s constructor function.

We then call the `SetCleanupFunc` method of the `liveClock` object, and provide it a clean up function, `liveClock.Cleanup`. The `SetCleanupFunc` method is included in the `cog.UXCog` type. It allows us to specify a clean up function that should be called prior to the `cog` being removed from the DOM. Finally, we return the new instance of the `LiveClock` `cog`.

Let's examine the `Cleanup` function:

```
func (lc *LiveClock) Cleanup() {
    lc.ticker.Stop()
}
```

This function is really simple. We simply call the `Stop` method on the cog's `ticker` object to stop the `ticker`.

Here's the cog's `Start` method where the `ticker` will be started:

```
func (lc *LiveClock) Start() error {
```

We start out by declaring the time layout constant, `layout`, and setting it to the `RFC1123` time format. We declare a `location` variable, a pointer to a `time.Location` type:

```
const layout = time.RFC1123
var location *time.Location
```

Prior to starting up a `LiveClock` cog, the user of the cog must set two important props, the `"timezoneName"`, and the `"timezoneOffset"`:

```
if lc.Props["timezoneName"] != nil && lc.Props["timezoneOffset"] != nil {
    location = time.FixedZone(lc.Props["timezoneName"].(string),
        lc.Props["timezoneOffset"].(int))
} else {
    return errors.New("The timezoneName and timezoneOffset props need to be
set!")
}
```

These values are used to initialize the `location` variable. If either of these props were not provided, an `error` will be returned.

If both of the props are present, we proceed to assigning the `ticker` property of the live clock cog to a newly created `time.Ticker` instance, which will tick at every second:

```
lc.ticker = time.NewTicker(time.Millisecond * 1000)
```

We range on the `ticker`'s channel to iterate at every one second, as a value arrives, and we set the `currentTime` prop, providing it a formatted time value (shown in bold):

```
go func() {
    for t := range lc.ticker.C {
        lc.SetProp("currentTime", t.In(location).Format(layout))
    }
}()
```

Note that we used both the location, and the time layout to format the time. Once the cog has been rendered, calls to `SetProp` that will occur every second will automatically call the `Render` method to re-render the cog.

We make a call to the cog's `Render` method to render the cog to the web page:

```
err := lc.Render()
if err != nil {
    return err
}
```

In the last line of the method, we return a `nil` value to indicate that no errors occurred:

```
return nil
```

We've defined the template for the cog in the `liveclock tmpl` source file:

```
<p>{{.timeLabel}}: {{.currentTime}}</p>
```

We print out the time label, along with the current time. The `timeLabel` prop is used to supply the time label to the cog and will be the name of the place for which we want to know the current time of.

Now that we've seen what goes into making the live clock cog, and how it displays the time, let's go ahead and sprinkle some live clock cogs on the home page.

Here's the section of code inside the `InitializeIndexPage` function of the `index.go` source file, where we instantiate the live clock cog for the local time zone:

```
// Localtime Live Clock Cog
localZonename, localOffset := time.Now().In(time.Local).Zone()
lc := liveclock.NewLiveClock()
lc.CogInit(env.TemplateSet)
lc.SetID("myLiveClock")
lc.SetProp("timeLabel", "Local Time")
lc.SetProp("timezoneName", localZonename)
lc.SetProp("timezoneOffset", localOffset)
err = lc.Start()
if err != nil {
    println("Encountered the following error when attempting to start the
local liveclock cog: ", err)
}
```

In order to instantiate the cog for the local time, we first obtain the local zone name and the local time zone offset. We then create a new instance of a `LiveClock` cog called `lc`. We call the `CogInit` method to initialize the cog. We call the `SetID` method to register the `id` of the cog's mount point, the `div` container where the cog will render its output to. We make calls to the `SetProp` method to set the `"timeLabel"`, `"timezoneName"`, and `"timezoneOffset"` props. Finally, we call the `Start` method to start up the `LiveClock` cog. As usual, we check to see if the cog started up properly, and if it didn't, we print the `error` object in the web console.

In a similar manner, we instantiate the `LiveClock` cogs for Chennai, Singapore, and Hawaii, in much the same way as we did for the local time, except for one thing. For the other places, we explicitly provide the timezone name and the GMT timezone offset for each place:

```
// Chennai Live Clock Cog
chennai := liveclock.NewLiveClock()
chennai.CogInit(env.TemplateSet)
chennai.SetID("chennaiLiveClock")
chennai.SetProp("timeLabel", "Chennai")
chennai.SetProp("timezoneName", "IST")
chennai.SetProp("timezoneOffset", int(+5.5*3600))
err = chennai.Start()
if err != nil {
    println("Encountered the following error when attempting to start the
chennai liveclock cog: ", err)
}

// Singapore Live Clock Cog
singapore := liveclock.NewLiveClock()
singapore.CogInit(env.TemplateSet)
singapore.SetID("singaporeLiveClock")
singapore.SetProp("timeLabel", "Singapore")
singapore.SetProp("timezoneName", "SST")
singapore.SetProp("timezoneOffset", int(+8.0*3600))
err = singapore.Start()
if err != nil {
    println("Encountered the following error when attempting to start the
singapore liveclock cog: ", err)
}

// Hawaii Live Clock Cog
hawaii := liveclock.NewLiveClock()
hawaii.CogInit(env.TemplateSet)
hawaii.SetID("hawaiiLiveClock")
hawaii.SetProp("timeLabel", "Hawaii")
```

```
hawaii.SetProp("timezoneName", "HDT")
hawaii.SetProp("timezoneOffset", int(-10.0*3600))
err = hawaii.Start()
if err != nil {
    println("Encountered the following error when attempting to start the
hawaii liveclock cog: ", err)
}
```

Now, we will be able to see the live clock cogs in action. *Figure 9.8* is a screenshot of the live clogs displayed on the homepage.



Figure 9.8: The live clock cog in action

With every passing second, each live clock gets updated with the new time value. The virtual DOM kicks in and renders only the difference of what changed, efficiently re-rendering the live clock at every second.

The first two cogs that we have implemented so far, have been pure cogs that are entirely implemented in Go. What if we wanted to leverage an existing JavaScript solution to provide a specific feature? That would be a situation that calls for implementing a hybrid cog, a cog that is implemented in Go and JavaScript.

## Implementing hybrid cogs

JavaScript has been around for more than two decades. In that time span, a lot of robust, production-ready solutions have been created using the language. Isomorphic Go cannot exist on its own island, and we have to acknowledge that there are many helpful, ready-made solutions that are useful from the JavaScript ecosystem. In many scenarios, we can save considerable time and effort by creating solutions that leverage existing JavaScript solutions, rather than re-implementing the whole solution in a pure Go manner.

Hybrid cogs are implemented using Go and JavaScript. The main purpose of hybrid cogs is to leverage functionality in existing JavaScript solutions and expose that functionality as a `cog`. This means that `cog` implementors would need to know both Go and JavaScript to implement hybrid cogs. Keep in mind that users of hybrid cogs only need to know Go, since the usage of JavaScript is an internal implementation detail of the `cog`. This allows cogs to be readily usable by Go developers that may not be familiar with JavaScript.

## The date picker cog

Let's consider a scenario that warrants the implementation of a hybrid `cog`. Molly, the de-facto product manager of IGWEB, came up with a killer idea to provide better customer support. Her feature request to the tech team was to allow the website user to provide an optional priority date on the contact form, by which a user should hear back from a gopher on the IGWEB team.

Molly found a self-contained, date picker widget, implemented in vanilla JavaScript (no framework/library dependencies) called Pikaday: <https://github.com/dbushell/Pikaday>.

The Pikaday, JavaScript date picker widget, highlights the fact that was presented in the beginning of this section. JavaScript is not going away, and there are many useful solutions that have already been made using it. That means, we must have the capability to leverage existing JavaScript solutions when it makes sense to do so. The Pikaday date picker is a particular use-case where it is more beneficial to leverage this existing JavaScript date picker widget, rather than implement one as a pure cog:

The wireframe shows a 'Contact' form. It includes fields for First Name, Last Name, and E-mail Address, each with a text input box. Below these is a text area with the placeholder 'Enter your message for us here.' At the bottom, there is a section labeled 'Time Sensitivity Date' containing a calendar for August 2016. The calendar shows the days of the week (S, M, T, W, T, F, Sa) and the dates from 31 of July to 10 of September. The 1st of August is highlighted in grey. The 'Contact' button is located at the bottom of the form.

S	M	T	W	T	F	Sa
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	1	2	3
4	5	6	7	8	9	10

Figure 9.9: A wireframe design depicting the time sensitivity date input field and the calendar date picker widget

Figure 9.9 is a wireframe design depicting the contact form with a time sensitivity input field, that when clicked, will reveal a calendar date picker. Let's see what it takes to fulfill Molly's request by implementing the date picker cog, a hybrid cog, made with Go and JavaScript.

We start out by placing the JavaScript and CSS source files required by the Pikaday, date picker widget, inside the `js` and `css` folders (respectively), inside the cog's `static` folder.

Inside the `shared/templates/partials/contactform_partial.tpl` source file, we declare the mount point for the date picker cog (shown in bold):

```
<fieldset class="pure-control-group">
  <div data-component="cog" id="sensitivityDate"></div>
</fieldset>
```

The `div` container fulfills two basic requirements of all cog mounting points: we have set the attribute, `"data-component"`, with the value `"cog"`, and we have specified an `id` of `"sensitivityDate"` for the cog container.

Let's examine the implementation of the date picker cog, defined in the `shared/cogs/datepicker/datepicker.go` source file, section by section. First, we start out by declaring the package name:

```
package datepicker
```

Here's the cog's import grouping:

```
import (
  "errors"
  "reflect"
  "time"

  "github.com/gopherjs/gopherjs/js"
  "github.com/uxtoolkit/cog"
)
```

Notice that we include the `gopherjs` package in the import grouping (shown in bold). We will need functionality from `gopherjs` to query the DOM.

Right after we declare the `cogType`, we initialize the `JS` variable to `js.Global`:

```
var cogType reflect.Type
var JS = js.Global
```

As you may recall, this saves us a little bit of typing. We can directly refer to `js.Global` as `JS`.

From the Pikaday project web page, <https://github.com/dbushell/Pikaday>, we can learn all of the input parameters the date picker widget accepts. The input parameters are provided as a single JavaScript object. The date picker `cog` will expose a subset of these input parameters, just enough to fulfill Molly's feature request. We create a struct called `DatePickerParams` that serves as the input parameters to the date picker widget:

```
type DatePickerParams struct {
    *js.Object
    Field *js.Object `js:"field"`
    FirstDay int `js:"firstDay"`
    MinDate *js.Object `js:"minDate"`
    MaxDate *js.Object `js:"maxDate"`
    YearRange []int `js:"yearRange"`
}
```

We embed the `*js.Object` to indicate that this is a JavaScript object. We then declare the respective Go fields for the struct for the respective properties of the JavaScript input object. For example, the field named `Field` is for the `field` property. The `"js"` struct tags that we have provided for each field allows GopherJS to convert the struct and its field, from its designated Go name to its equivalent JavaScript name. Just as we declared the field named `Field`, we have also declared fields for `FirstDay` (`firstDay`), `MinDate` (`minDate`), `MaxDate` (`maxDate`), and `YearRange` (`yearRange`).

Reading the Pikaday documentation, <https://github.com/dbushell/Pikaday>, we can learn what purpose each of these input parameters serve:

- `Field` - Used to bind the date picker to a form field.
- `FirstDay` - Used to specify the first day of the week. (0 for Sunday, 1 for Monday, etc).
- `MinDate` - The earliest date that can be selected in the date picker widget.
- `MaxDate` - The latest date that can be selected in the date picker widget.
- `YearRange` - The range of years to display.

Now that we've defined the date picker's input parameters struct, `DatePickerParams`, it's time to implement the date picker cog. We start out by declaring the `DatePicker` struct:

```
type DatePicker struct {
    cog.UXCog
    picker *js.Object
}
```

As usual, we embed the `cog.UXCog` to bring along all the `UXCog` functionality we need. We also declare a field, `picker`, that's a pointer to a `js.Object`. The `picker` property will be used to refer to the `Pikaday` date picker JavaScript object.

We then implement a constructor function for the date picker cog called `NewDatePicker`:

```
func NewDatePicker() *DatePicker {
    d := &DatePicker{}
    d.SetCogType(cogType)
    return d
}
```

By now, the cog constructor should look familiar to you. Its duties are to return a new instance of a `DatePicker` and to set the cog's `cogType`.

Now that our constructor function is in place, it's time to examine the date picker cog's `Start` method:

```
func (d *DatePicker) Start() error {

    if d.Props["datepickerInputID"] == nil {
        return errors.New("Warning: The datePickerInputID prop need to be
set!")
    }

    err := d.Render()
    if err != nil {
        return err
    }
}
```

We start out by checking to see if the "datepickerInputID" prop has been set. This is the id of the input field element, which will be used as the Field value in the DatePickerParams struct. It is a hard requirement, that this prop must be set by the caller, before starting the cog. Failure to set this prop will result in an error.

If the "datepickerInputID" prop has been set, we call the cog's Render method to render the cog. This will render the HTML markup for the input field that the date picker JavaScript widget object will rely on.

We then go on to declare and instantiate, params, the input parameters JavaScript object that will be shipped to the date picker JavaScript widget:

```
params := &DatePickerParams{Object: js.Global.Get("Object").New()}
```

The date picker input parameters object, params, is a JavaScript object. The Pikaday JavaScript object will use the params object for initial configuration.

We use the cog's Props property to range through the cog's properties. For each iteration we fetch the property's name (propName) and the property's value (propValue):

```
for propName, propValue := range d.Props {
```

The switch block we have declared is important for readability:

```
switch propName {  
  
case "datepickerInputID":  
    inputFieldID := propValue.(string)  
    dateInputField := JS.Get("document").Call("getElementById",  
    inputFieldID)  
    params.Field = dateInputField  
  
case "datepickerLabel":  
    // Do nothing  
  
case "datepickerMinDate":  
    datepickerMinDate := propValue.(time.Time)  
    minDateUnix := datepickerMinDate.Unix()  
    params.MinDate = JS.Get("Date").New(minDateUnix * 1000)  
  
case "datepickerMaxDate":  
    datepickerMaxDate := propValue.(time.Time)  
    maxDateUnix := datepickerMaxDate.Unix()  
    params.MaxDate = JS.Get("Date").New(maxDateUnix * 1000)  
  
case "datepickerYearRange":
```

```
yearRange := propValue.([]int)
params.YearRange = yearRange

default:
    println("Warning: Unknown prop name provided: ", propName)
}
```

Each case statement inside the `switch` block, tells us all the properties the date picker cog accepts as input parameters that will be ferried over to the Pikaday JavaScript widget. If a prop name is not recognized, we print out a warning in the web console that the prop is unknown.

The first case, handles the `"datepickerInputID"` prop. It will be used to specify the `id` of the input element that activates the Pikaday widget. Inside this `case`, we get the input element field by calling the `getElementById` method on the `document` object and passing the `inputFieldID` to the method. We set the `input` `params` property, `Field` to the input field element that was obtained from the `getElementById` method call.

The second case handles the `"datepickerLabel"` prop. The value for the `"datepickerLabel"` prop will be used in the cog's template source file. Therefore there's no work needed to handle this particular case.

The third case handles the `"datepickerMinDate"` prop. It will be used to get the minimum date that should be shown by the Pikaday widget. We convert the `"datepickerMinDate"` value of type `time.Time` provided by the caller to its Unix timestamp representation. We then create a new JavaScript `date` object using the Unix timestamp, which is suitable for use for the `minDate` input parameter.

The fourth case handles the `"datepickerMaxDate"` prop. It will be used to get the maximum date that should be shown by the date picker widget. We follow the same strategy here, that we did for the `minDate` argument.

The fifth case handles the `"datepickerYearRange"` prop. It will be used to specify the range of years that the displayed calendar will cover. The year range is a slice, and we populate the `YearRange` property of the `input` parameters object using the prop's value.

As stated earlier, the `default` case handles the scenario, where the caller provides a prop name that is not known. If we reach the `default` case, we print out a warning message in the web console.

Now we can instantiate the Pikaday widget and provide the input parameters object, `params`, to it like so:

```
d.picker = JS.Get("Pikaday").New(params)
```

Finally, we indicate that there were no errors starting up the cog, by returning a `nil` value:

```
return nil
```

Now that we have implemented the date picker cog, let's take a look at what the cog's primary template, defined in the

`shared/cogs/datepicker/templates/datepicker.tpl` source file, looks like this:

```
<label class="datepickerLabel"  
for="datepicker">{{.datepickerLabel}}</label>  
<input class="datepickerInput" type="text" id="{{.datepickerInputID}}"  
name="{{.datepickerInputID}}>
```

We declare a `label` element to display the label of the date picker cog using the prop `"datepickerLabel"`. We declare an `input` element that will serve as the input element field that will be used in conjunction with the Pikaday widget. We specify the `id` attribute of the input element field using the `"datepickerInputID"` prop.

Now that we have implemented the date picker cog, it's time to start using it. We instantiate the cog inside the `InitializeContactPage` function, found in the `client/handlers/contact.go` source file:

```
byDate := datepicker.NewDatePicker()  
byDate.CogInit(env.TemplateSet)  
byDate.SetID("sensitivityDate")  
byDate.SetProp("datepickerLabel", "Time Sensitivity Date:")  
byDate.SetProp("datepickerInputID", "byDateInput")  
byDate.SetProp("datepickerMinDate", time.Now())  
byDate.SetProp("datepickerMaxDate", time.Date(2027, 12, 31, 23, 59, 0, 0,  
time.UTC))  
err := byDate.Start()  
if err != nil {  
    println("Encountered the following error when attempting to start the  
datepicker cog: ", err)  
}
```

First, we create a new instance of the `DatePicker` cog. We then call the cog's `CogInit` method, to register the application's template set. We call the `SetID` method to set the cog's mount point. We make calls to the cog's `SetProp` method to set the `datePickerLabel`, `datepickerInputID`, `datepickerMinDate`, and `datepickerMaxDate` props. We call the cog's `Start` method to activate it. If there were any errors starting the cog, we print the error message out to the web console.

And that's all there is to it! We could leverage the functionality we needed from the Pikaday widget using the date picker hybrid cog. The advantage of this approach is that Go developers utilizing the date picker cog will not need to have knowledge of the inner workings (JavaScript) of the Pikaday widget in order to use it. Instead, they can use the functionality that the date picker cog exposes to them from within the confines of Go.

Figure 9.10 shows a screenshot of the date picker cog in action:

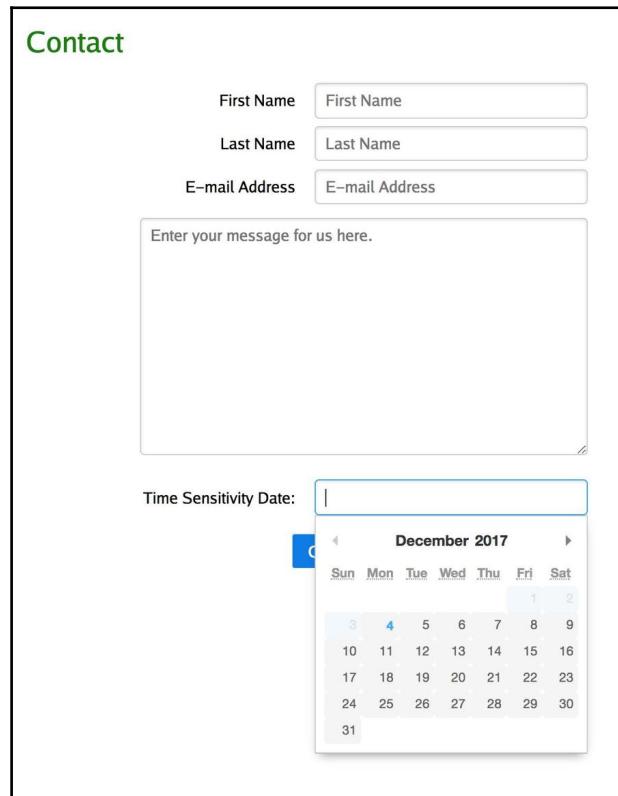


Figure 9.10: The calendar date picker widget in action

Even if the cog user didn't provide any props, other than the required `datepickerInputID`, to custom configure the date picker cog, the Pikaday widget starts up just fine. However, what if we needed to supply a default set of parameters for the cog? In the next example, we are going to build another hybrid cog, a carousel (an image slider) cog, in which we will define default parameters.

## The carousel cog

In this example, we will be creating an image carousel cog, as depicted by the wireframe design in *Figure 9.11*.

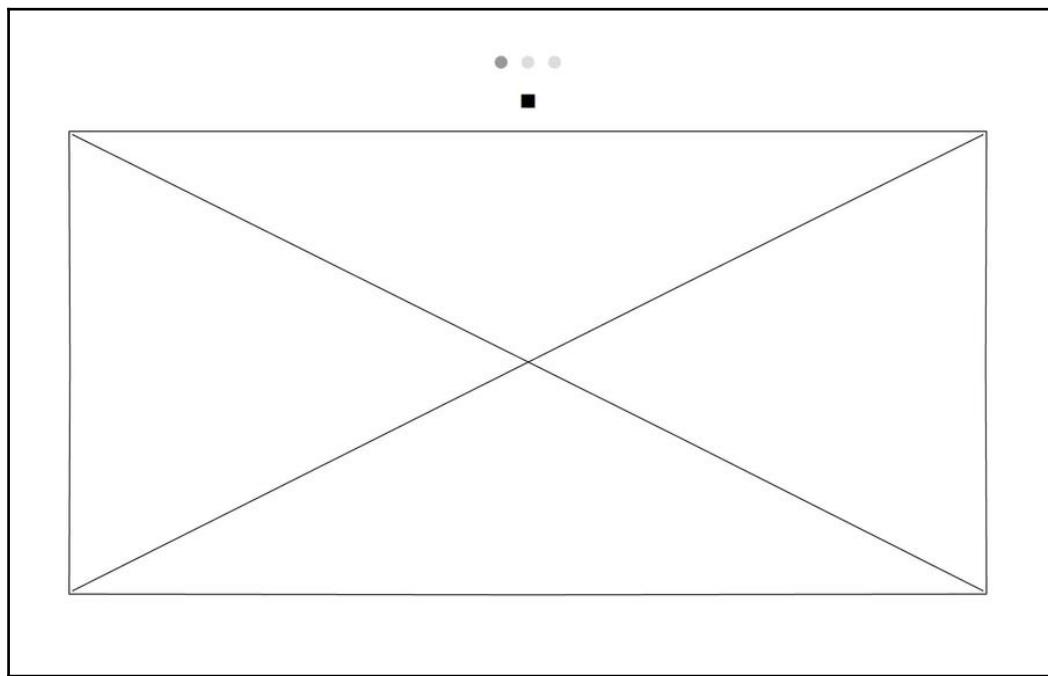


Figure 9.11: A wireframe design depicting the carousel cog

The carousel cog will be powered by the tiny-slider widget which is implemented in vanilla JavaScript. Here's the URL to the tiny-slider project: <https://github.com/ganlanyuan/tiny-slider>.

We place the JavaScript source file for the tiny-slider widget, `tiny-slider.min.js` in the cog's `static/js` folder. We place the CSS files associated with the tiny-slider widget, `tiny-slider.css`, and `styles.css` in the `static/css` folder.

The carousel cog that we will build will expose the following input parameters provided by the tiny-slider widget:

```
container Node | String Default: document.querySelector('.slider').
```

The `container` parameter represents the slider container element or selector:

```
items Integer Default: 1.
```

The `items` parameter represents the number of slides being displayed:

```
slideBy Integer | 'page' Default: 1.
```

The `slideBy` parameter represents the number of slides going on one "click":

```
autoplay Boolean Default: false.
```

The `autoplay` parameter toggles the automatic change of slides:

```
autoplayText Array (Text | Markup) Default: ['start', 'stop'].
```

The `autoplayText` parameter controls the text or markup that appears in the autoplay start/stop button.

```
controls Boolean Default: true.
```

The `controls` parameter is used to toggle the display and functionalities of controls (previous/next buttons).

The image carousel will display a set of featured products available on IGWEB. We've declared the cog's mounting point in the `shared/templates/index_content.tpl` source file:

```
<div data-component="cog" id="carousel"></div>
```

We've declared the `div` container that will serve as the carousel cog's mount point. We've declared the attribute, `"data-component"`, and assigned it a value of `"cog"`. We've also declared an `id` attribute of `"carousel"`.

The carousel cog is implemented in the `carousel.go` source file found in the `shared/cogs/carousel` folder. Here's the package declaration and import groupings:

```
package carousel

import (
    "errors"
    "reflect"

    "github.com/gopherjs/gopherjs/js"
    "github.com/uxtoolkit/cog"
)
```

The tiny-slider widget is instantiated with an input parameters JavaScript object. We'll be using the `CarouselParams` struct to model the input parameters object:

```
type CarouselParams struct {
    *js.Object
    Container string `js:"container"`
    Items int `js:"items"`
    SlideBy string `js:"slideBy"`
    Autoplay bool `js:"autoplay"`
    AutoplayText []string `js:"autoplayText"`
    Controls bool `js:"controls"`
}
```

After embedding the pointer to `js.Object`, each field that we have defined in the `struct` corresponds to its equivalent JavaScript parameter object property. For example, the `Container` field maps to the `container` property of the input parameters object.

Here's the struct that defines the `carousel` cog:

```
type Carousel struct {
    cog.UXCog
    carousel *js.Object
}
```

As usual we embed the `cog.UXCog` type to borrow functionality from the `UXCog`. The `carousel` field will be used to refer to the tiny-slider widget, which is a JavaScript object.

By now, you should be able to guess what the `carousel` cog's constructor function looks like:

```
func NewCarousel() *Carousel {
    c := &Carousel{}
    c.SetCogType(cogType)
    return c
}
```

Besides creating a new reference to a `Carousel` instance, the constructor function sets the `cog`'s `cogType`.

Now it's time to examine the lion's share of the carousel cog's implementation, which is found in the `cog`'s `Start` method:

```
func (c *Carousel) Start() error {
```

We start out by checking if the user of the `cog` has set the `contentItems` and `carouselContentID` props. The `contentItems` prop is a string slice of server relative image paths for the images that should appear in the carousel. The `carouselContentID` prop is the value of the `id` attribute of the `div` container that holds the carousel's content.

If either of these props haven't been set, we return an `error` indicating that both of these props must be set. If the two props have been set, we proceed to render the `cog`:

```
if c.Props["contentItems"] == nil || c.Props["carouselContentID"] == nil
{
    return errors.New("The contentItems and carouselContentID props need to
be set!")
}

err := c.Render()
if err != nil {
    return err
}
```

We render the `cog` at this juncture, since there is HTML markup that needs to exist on the web page in order for the `cog` to function properly. Notably, the `div` container that holds the carousel's content, whose `id` we supply using the required `carouselContentID` prop. If there was an `error` rendering the `cog`, we return the `error` to indicate that the `cog` cannot be started. If there was no `error` encountered while rendering the `cog`, we proceed to instantiate the `inputParameters` object:

```
params := &CarouselParams{Object: js.Global.Get("Object").New()}
```

This `struct` represents the input parameters that we will be feeding to the `tiny-slider` object upon its instantiation.

The next section of code is important, since this is where we define the default parameters:

```
// Set the default parameter values
params.Items = 1
params.SlideBy = "page"
params.Autoplay = true
params.AutoplayText = []string{PLAYTEXT, STOPTEXT}
params.Controls = false
```

When a cog maintainer looks at this block of code, they can readily ascertain what the default behavior of the cog is. From looking at the default parameters, one can tell that the slider will show only one item at a time. The slider is set to slide by page mode and the slider will automatically start the slideshow. We've provided a string slice for the AutoplayText property, with text symbols for the play and stop buttons using the PLAYTEXT and STOPTEXT constants respectively. We have set the Controls property to false, so that the **Next** and **Previous** buttons will not appear on the image carousel by default.

We proceed to iterate over all the properties that the user of the cog has provided, accessing each prop, consisting of a propName (string) and a propValue (interface{}):

```
for propName, propValue := range c.Props {
```

We declare a switch block on the propName:

```
switch propName {

case "carouselContentID":
    if propValue != nil {
        params.Container = "#" + c.Props["carouselContentID"].(string)
    }

case "contentItems":
    // Do nothing

case "items":
    if propValue != nil {
        params.Items = propValue.(int)
    }

case "slideBy":
    if propValue != nil {
        params.SlideBy = c.Props["slideBy"].(string)
    }

case "autoplay":
```

```
if propValue != nil {
    params.Autoplay = c.Props["autoplay"].(bool)
}

case "autoplayText":
    if propValue != nil {
        params.AutoplayText = c.Props["autoplayText"].(>[]string)
    }

case "controls":
    if propValue != nil {
        params.Controls = c.Props["controls"].(bool)
    }

default:
    println("Warning: Unknown prop name provided: ", propName)
}
```

Using the `switch` block makes it easy to see the names of all the valid props in each `case` statement that is defined. If a prop name is not known, it falls into the `default` case, where we print a warning message in the web console.

The first `case` handles the required `"carouselContentID"` prop. It is used to specify the `div` container that will contain the content items of the carousel.

The second `case` handles the required `"contentItems"` prop. This prop is a `string` slice, and it's meant to be used in the cog's template, so there's no action that we have to perform for it.

The third `case` handles the `"items"` prop. This is the prop that handles the `trs-slider` object's `items` parameter, which shows the amount of slides to display at a given time. If the prop value is not `nil`, we assign the `int` value of the prop value to the `params.Items` property.

The fourth `case` handles the `slideBy` prop. If the prop value is not `nil`, we assign the prop value (type asserted as a `string`) to the `SlideBy` property of the `params` object.

The fifth `case` handles the `"autoplay"` prop. If the prop value is not `nil`, we assign the prop value (type asserted as a `bool`) to the `Autoplay` property of the `params` object.

The sixth `case` handles the `"autoplayText"` prop. If the prop value is not `nil`, we assign the prop value (type asserted as a `[]string`) to the `AutoplayText` property of the `params` object.

The seventh `case` handles the `"controls"` prop. If the prop value is not `nil` we assign the property value (type asserted as a `bool`) to the `Controls` property of the `params` object.

If the property name does not fall under any of the seven previous cases, it will be handled by the `default case`. Recall, that if we were to reach this `case`, it indicates that the user of the `cog` has supplied a prop name that is unknown.

We can now instantiate the tiny-slider widget and assign it to the `cog`'s `carousel` property:

```
c.carousel = JS.Get("tns").New(params)
```

A value of `nil` is returned by the `Start` method, to indicate that there were no errors encountered while starting up the `cog`:

```
return nil
```

The `shared/cogs/carousel/templates/carousel tmpl` source file defines the template for the `carousel` `cog`:

```
<div id="{{.carouselContentID}}" class="carousel">
{{range .contentItems}}
<div>.

Peek inside the `shared/cogs/notify` folder, and notice that no `templates` folder exists. We have placed the static assets for Alertify's CSS, and JavaScript source files, in the `shared/cogs/notify/static/css` and `shared/cogs/notify/static/js` folders respectively.

The `notify cog` is implemented in the `notify.go` source file found in the `shared/cogs/notify` folder. Since it makes sense for the client-side web application to have only one notification system that which is provided by the `notify cog`, only one instance of the `cog` should ever be started. In order to keep track and ensure that only one `notify cog` instance can be started, we will declare the `alreadyStarted` Boolean variable:

```
var alreadyStarted bool
```

The `Notify` struct defines the fields of the notify cog:

```
type Notify struct {
    cog.UXCog
    alertify *js.Object
    successNotificationEventListener func(*js.Object)
    errorNotificationEventListener func(*js.Object)
}
```

We type embed the `cog.UXCog` in order to bring over the functionality needed to implement the `Cog` interface. The `alertify` field is used to refer to the `alertify` JavaScript object.

The notify cog that we are building is event-driven. For example, a success notification will be displayed when a custom success notification event is fired from any page on the client-side application. We have defined two fields, `successNotificationEventListener` and `errorNotificationEventListener`, which are both functions that take a JavaScript object pointer as an input variable. We have defined these fields, so that we can keep track of the custom event listener functions that we set up to listen for success and error notifications. When it comes time to remove the event listeners, it becomes easy to access them because they are properties of the notify cog instance.

The `NewNotify` function serves as a constructor function:

```
func NewNotify() *Notify {
    n := &Notify{}
    n.SetCogType(cogType)
    n.SetCleanupFunc(n.Cleanup)
    return n
}
```

Notice that we have registered a clean-up function (shown in bold), which will be called prior to the cog being destroyed.

Let's examine the cog's `Start` method:

```
func (n *Notify) Start() error {
    if alreadyStarted == true {
        return errors.New("The notification cog can be instantiated only
once.")
    }
}
```

We first check to see if a notify cog instance has already been started by checking the value of the `alreadyStarted` Boolean variable. If the value of `alreadyStarted` is `true`, it indicates that a previous notify cog instance has already been started, so we return an `error` indicating that the notify cog could not be started.

If the cog has not been started yet, we proceed to instantiate the Alertify JavaScript object:

```
n.alertify = js.Global.Get("alertify")
```

We make a call to the cog's `StartListening` method to set up the event listeners that listen for custom success and error notification message events:

```
n.StartListening()  
return nil
```

Here's the cog's `StartListening` method:

```
func (n *Notify) StartListening() {  
  
    alreadyStarted = true  
    D := dom.GetWindow()  
    n.successNotificationEventListener =  
    D.AddEventListener("displaySuccessNotification", false, func(event  
dom.Event) {  
        message := event.Underlying().Get("detail").String()  
        n.notifySuccess(message)  
    })  
  
    n.errorNotificationEventListener =  
    D.AddEventListener("displayErrorNotification", false, func(event dom.Event)  
{  
        message := event.Underlying().Get("detail").String()  
        n.notifyError(message)  
    })  
}
```

If we have reached this method, it indicates that the cog was started successfully, so we set the `alreadyStarted` Boolean variable to `true`. We set up an event listener that will listen for the `displaySuccessNotification` custom event. We keep track of the event listener function that we are creating by assigning it to the `successNotificationEventListener` property of the cog instance. We declare and instantiate the `message` variable and set it equal to the the `detail` property of the `event` object, which will contain the `string` message that should be displayed to the user on the web page. We then call the cog's `notifySuccess` method to display the success notification message on the web page.

We follow a similar procedure to set up the event listener for the `displayErrorNotification`. We assign the event listener function to the cog's `errorNotificationEventListener` property. We extract the `detail` property from the event object and assign it to the `message` variable. We call the cog's `notifyError` method to display the error notification message on the web page.

The `notifySuccess` method is responsible for displaying a success notification message on the web page:

```
func (n *Notify) notifySuccess(message string) {
    n.alertify.Call("success", message)
}
```

We call the `alertify` object's `success` method to display the success notification message.

The `notifyError` method is responsible for displaying an error notification message on the web page:

```
func (n *Notify) notifyError(message string) {
    n.alertify.Call("error", message)
}
```

We call the `alertify` object's `error` method to display the error notification message.

The cog's `CleanUp` method simply makes a call to the cog's `StopListening` method:

```
func (n *Notify) CleanUp() {
    n.StopListening()
}
```

The `StopListening` method is used to remove the event listeners prior to the cog being destroyed:

```
func (n *Notify) StopListening() {
    D := dom.GetWindow()
    if n.successNotificationEventListener != nil {
        D.RemoveEventListener("displaySuccessNotification", false,
        n.successNotificationEventListener)
    }

    if n.errorNotificationEventListener != nil {
        D.RemoveEventListener("displayErrorNotification", false,
        n.errorNotificationEventListener)
    }
}
```

We call the DOM object's `RemoveEventListener` method to remove the event listener functions that handle the `displaySuccessNotification` and `displayErrorNotification` custom events.

The `notify` package's exported `Success` function is used to broadcast a custom success event notification message:

```
func Success(message string) {
    var eventDetail = js.Global.Get("Object").New()
    eventDetail.Set("detail", message)
    customEvent :=
        js.Global.Get("window").Get("CustomEvent").New("displaySuccessNotification",
        eventDetail)
        js.Global.Get("window").Call("dispatchEvent", customEvent)
}
```

Inside the function, we create a new JavaScript object called `eventDetail`. We assign the string message that should be displayed on the web page to the `detail` property of the `eventDetail` object. We then create a new custom event object called `customEvent`. We pass in the name of the custom event, `displaySuccessNotification`, along with the `eventDetail` object as input arguments to the `CustomEvent` type's constructor function. Finally, to dispatch the event, we call the `dispatchEvent` method on the `window` object and supply the `customEvent`.

The `notify` package's exported `Error` function is used to broadcast a custom error event notification message:

```
func Error(message string) {
    var eventDetail = js.Global.Get("Object").New()
    eventDetail.Set("detail", message)
    customEvent :=
        js.Global.Get("window").Get("CustomEvent").New("displayErrorNotification",
        eventDetail)
        js.Global.Get("window").Call("dispatchEvent", customEvent)
}
```

The implementation of this function is nearly identical to the `Success` function. The only difference is that we dispatch a `displayErrorNotification` custom event.

We instantiate and start the **notify cog** (shown in bold) inside the `InitializePageLayoutControls` function found in the `client/handlers/initpagelayoutcontrols.go` source file:

```
func InitializePageLayoutControls(env *common.Env) {  
  
    n := notify.NewNotify()  
    err := n.Start()  
    if err != nil {  
        println("Error encountered when attempting to start the notify cog: ",  
err)  
    }  
  
    liveChatIcon :=  
env.Document.GetElementByID("liveChatIcon").(*dom.HTMLImageElement)  
liveChatIcon.AddEventListener("click", false, func(event dom.Event) {  
  
    chatbox := env.Document.GetElementByID("chatbox")  
    if chatbox != nil {  
        return  
    }  
    go chat.StartLiveChat(env)  
})  
  
}
```

The notification messages (success or error) for adding an item to the shopping cart are found within the `addToCart` function inside the `client/handlers/shoppingcart.go` source file:

```
func addToCart(productSKU string) {  
  
    m := make(map[string]string)  
    m["productSKU"] = productSKU  
    jsonData, _ := json.Marshal(m)  
  
    data, err := xhr.Send("PUT", "/restapi/add-item-to-cart", jsonData)  
    if err != nil {  
        println("Encountered error: ", err)  
        notify.Error("Failed to add item to cart!")  
        return  
    }  
    var products []*models.Product  
    json.NewDecoder(strings.NewReader(string(data))).Decode(&products)  
    notify.Success("Item added to cart")  
}
```

The `notify.Error` function is called (shown in bold) if the item could not be added to the shopping cart. The `notify.Success` function is called (shown in bold) if the item was successfully added to the shopping cart.

The notification messages for removing an item from the shopping cart are found in the `removeFromCart` function in the `client/handlers/shoppingcart.go` source file:

```
func removeFromCart(env *common.Env, productSKU string) {

    m := make(map[string]string)
    m["productSKU"] = productSKU
    jsonData, _ := json.Marshal(m)

    data, err := xhr.Send("DELETE", "/restapi/remove-item-from-cart",
    jsonData)
    if err != nil {
        println("Encountered error: ", err)
        notify.Error("Failed to remove item from cart!")
        return
    }
    var products []*models.Product
    json.NewDecoder(strings.NewReader(string(data))).Decode(&products)
    renderShoppingCartItems(env)
    notify.Success("Item removed from cart")
}
```

The `notify.Error` function is called (shown in bold) if the item could not be removed from the shopping cart. The `notify.Success` function is called (shown in bold) if the item was successfully removed from the shopping cart.

*Figure 9.14* is a cropped screenshot of the notification cog in action, when we add a product to the shopping cart:



Figure 9.14: The notify cog in action

## Summary

In this chapter, we were introduced to cogs—reusable components that could be implemented either exclusively in Go (pure cogs) or with Go and JavaScript (hybrid cogs). Cogs come with many benefits. We can use them in a plug and play manner, create multiple instances of them, readily maintain them due to their self-contained nature, and easily reuse them since they can exist as their own Go package along with their required static assets (template files, CSS, and JavaScript source files).

We introduced you to the UX toolkit, which provides us with the technology to implement cogs. We studied the anatomy of a cog and explored what a cog's file structure may look like with regards to the placement of Go, CSS, JavaScript, and template files. We considered how cogs utilize a virtual DOM to render their contents instead of performing an expensive replace inner HTML operation. We presented the various stages of the cog's life cycle. We showed you how to implement various cogs that we sprinkled all across IGWEB, which included both pure cogs and hybrid cogs.

In Chapter 10, *Testing an Isomorphic Go Web Application*, we will learn how to perform automated, end-to-end testing of IGWEB. This will consist of implementing tests to exercise functionality on both the server-side and the client-side.

# 10

## Testing an Isomorphic Go Web Application

With the sprinkling of reusable components (cogs) throughout the website, that was performed in the previous chapter, we had reached a project milestone – we completed the IGWEB feature set that was laid out in [Chapter 2, \*The Isomorphic Go Toolchain\*](#). However, we can't launch IGWEB just yet. Prior to launch, we must ensure the quality of the isomorphic web application by verifying that it meets a certain baseline set of functional requirements. To do this, we must implement end-to-end tests that exercise the isomorphic web application's functionality across environments (server-side and client-side).

In this chapter, you will learn how to provide end-to-end test coverage for IGWEB. We will use Go's built-in test framework, for testing the server-side functionality, and we will use CasperJS for testing the client-side functionality. By implementing a suite of end-to-end tests, not only will we have a means for automated testing, but we also have a valuable project artifact in each test that we write, since each test communicates the intent of the expected functionality found in the isomorphic web application. By the end of this chapter, we will have created an end-to-end test suite that forms the foundation of a solid test strategy, which the reader can further build upon.

In this chapter, we will cover the following topics:

- Testing the server-side functionality with Go's testing framework
- Testing the client-side functionality with CasperJS

# Testing the server-side functionality

As we learned in [Chapter 1, Isomorphic Web Applications with Go](#), the isomorphic web application architecture utilizes the classic web application architecture, meaning that the web page response will be rendered on the server-side. This means that the web client need not be JavaScript-enabled in order to consume the content received from the server response. This fact is especially important for machine users, such as search engine bots, who need to crawl through the various links found on the website and index them. Often times, search engine spiders are not JavaScript-enabled. This means that we have to ensure that server-side routing is functioning properly, and that the web page response is rendered properly also.

In addition to this, we put in good effort in [Chapter 7, The Isomorphic Web Form](#), to create an accessible, isomorphic web form that can be accessed by users who have greater accessibility needs. We need to ensure that the contact form's validation functionality is working, and that we can successfully send a valid contact form submission.

Therefore, on the server-side, the baseline set of functionality that we'll test for, includes the following items:

1. Verifying the server-side routing and template rendering
2. Verifying the contact form's validation functionality
3. Verifying a successful contact form submission

## Go's testing framework

We will use Go's built-in testing framework to write a set of tests that exercise IGWEB's server-side functionality. All server-side tests are stored in the `tests` folder.

If you are new to Go's built-in testing framework, provided through the `testing` package, you can read more about it at this link: <https://golang.org/pkg/testing/>.

Prior to running the `go test` command to execute all the tests, you must start up the Redis server instance and IGWEB (each preferably in their own dedicated Terminal window or tab).

You can start the Redis server instance with the following command:

```
$ redis-server
```

You can start the IGWEB instance with the following command, inside the `$IGWEB_APP_ROOT` folder:

```
$ go run igweb.go
```

To run all the tests in the suite, we simply need to run the `go test` command within the `tests` folder:

```
$ go test
```

## Verifying server-side routing and template rendering

We created a test to verify all the server-side routes of the IGWEB application. Each route that we test will be associated with an expected string token, which is rendered in the page response, particularly in the primary content `div` container. So, not only will we be able to verify if the server-side route is functioning properly or not, but we will also know if the server-side template rendering is functioning normally.

Here are the contents of the `routes_test.go` source file, found in the `tests` folder:

```
package tests

import (
    "io/ioutil"
    "net/http"
    "strings"
    "testing"
)

func checkRoute(t *testing.T, route string, expectedToken string) {

    testURL := testHost + route
    response, err := http.Get(testURL)
    if err != nil {
        t.Errorf("Could not connect to URL: %s. Failed with error: %s",
            testURL, err)
    } else {
        defer response.Body.Close()
        contents, err := ioutil.ReadAll(response.Body)
        if err != nil {
            t.Errorf("Could not read response body. Failed with error: %s",
                err)
        }
        if strings.Contains(string(contents), expectedToken) == false {
            t.Errorf("Could not find expected string token: \"%s\"", in
```

```
        response body for URL: %s", expectedToken, testURL)
    }
}
}

func TestServerSideRoutes(t *testing.T) {
    routesTokenMap := map[string]string{
        "": "IGWEB", "/": "IGWEB",
        "/index": "IGWEB", "/products": "Add To Cart", "/product-
        detail/swiss-army-knife": "Swiss Army Knife", "/about": "Molly",
        "/contact": "Enter your message for us here"
    }

    for route, expectedString := range routesTokenMap {
        checkRoute(t, route, expectedString)
    }
}
```

The `testHost` variable that we defined, is used to specify the hostname and port where the `IGWEB` instance is running.

The `TestServerSideRoutes` function is responsible for testing server-side routes, and verifying that the expected token string exists within the response body. Inside the function, we declare and initialize the `routesTokenMap` variable of type `map[string]string`. The keys in this `map`, represent the server-side route that we are testing, and the value for a given key, represents the expected `string` token that should exist in the web page response returned from the server. So, this test not only will tell us if the server-side route is functioning properly, but it will also give us a good idea on the health of template rendering, since the expected `string` tokens we supply are all strings that should be found in the body of the web page. We then `range` through the `routesTokenMap`, and for each iteration, we pass in the `route` and the `expectedString` to the `checkRoute` function.

The `checkRoute` function is responsible for accessing a given route, reading its response body and verifying that the `expectedString` exists within the response body. There are three conditions that can cause the test to fail:

1. When a connection to the route URL cannot be made
2. If the response body retrieved from the server could not be read
3. If the expected string token did not exist in the web page response returned from the server

If any of these three errors occur, the test will fail. Otherwise the function will return normally.

We can run this test by issuing the following `go test` command:

```
$ go test -run TestServerSideRoutes
```

Examining the output of running the test shows that the test has passed:

```
$ go test -run TestServerSideRoutes
PASS
ok github.com/EngineerKamesh/igb/igweb/tests 0.014s
```

We have now successfully verified accessing server-side routes and ensuring that the expected string in each route was rendered properly in the web page response. Now, let's start verifying the contact form functionality, starting with the form validation functionality.

## Verifying the contact form's validation functionality

The next test that we are going to implement will test the contact form's server-side form validation functionality. There are two types of validations that we will be testing:

- An error message that is displayed when the required form field has not been filled
- An error message that is displayed when an improperly formatted email address value is provided in the email field

Here are the contents of the `contactvalidation_test.go` source file, found in the `tests` folder:

```
package tests

import (
    "io/ioutil"
    "net/http"
    "net/url"
    "strconv"
    "strings"
    "testing"
)

func TestContactFormValidation(t *testing.T) {
    testURL := testHost + "/contact"
    expectedTokenMap := map[string]string{"firstName": "The first name field is required.", "/": "The last name field is required.", "email": "The e-mail address entered has an improper syntax."}
```

```
"messageBody": "The message area must be filled."}

form := url.Values{}
form.Add("firstName", "")
form.Add("lastName", "")
form.Add("email", "devnull@g@o")
form.Add("messageBody", "")

req, err := http.NewRequest("POST", testURL,
strings.NewReader(form.Encode()))

if err != nil {
    t.Errorf("Failed to create new POST request to URL: %s, with error:
    %s", testURL, err)
}

req.Header.Add("Content-Type", "application/x-www-form-urlencoded")
req.Header.Add("Content-Length", strconv.Itoa(len(form.Encode())))

hc := http.Client{}
response, err := hc.Do(req)

if err != nil {
    t.Errorf("Failed to make POST request to URL: %s, with error: %s",
    testURL, err)
}

defer response.Body.Close()
contents, err := ioutil.ReadAll(response.Body)

if err != nil {
    t.Errorf("Failed to read response body contents with error: %s",
    err)
}

for k, v := range expectedTokenMap {
    if strings.Contains(string(contents), v) == false {
        t.Errorf("Could not find expected string token: \"%s\" for field
        \"%s\"", v, k)
    }
}
```

The `TestContactFormValidation` function is responsible for testing the contact form's server-side form validation functionality. We declare and initialize the `testURL` variable, which is the URL of the contact section of IGWEB.

We declare and initialize the `expectedTokenMap` variable of the `map [string] string` type, where keys in the map are the names of the form fields and the value for each key represents the expected error message that should be returned upon submitting the form.

We create a new form and populate the form field values using the form object's `Add` method. Notice that we provided an empty `string` value for the `firstName`, `lastName`, and `messageBody` fields. We also provided an improperly formatted email address for the `email` field.

We use the `NewRequest` function found in the `http` package to submit the form using a HTTP POST request.

We create an `http.Client`, `hc`, and submit the POST request by calling its `Do` method. We fetch the contents of the response body using the `ReadAll` function found in the `ioutil` package. We range through the `expectedTokenMap`, and in each iteration, we check whether the expected error message is found in the response body.

These are the four possible conditions that can cause this test to fail:

- If the POST request could not be created
- If the POST request failed due to connectivity issues with the web server
- If the web client failed to read the response body of the web page response returned from the web server
- If the expected error message could not be found in the body of the web page

If any one of these errors are encountered, this test will fail.

We can run this test by issuing the following command:

```
$ go test -run TestContactFormValidation
```

Examining the output of running the test shows that the test has passed:

```
$ go test -run TestContactFormValidation
PASS
ok github.com/EngineerKamesh/igb/igweb/tests 0.009s
```

## Verifying a successful contact form submission

The next test that we will implement will test a successful contact form submission. This test, will be closely similar to the last test, with the exception that we will fill in all the form fields and provide a properly formatted email address in the `email` form field.

Here are the contents of the `contact_test.go` source file, found in the `tests` folder:

```
package tests

import (
    "io/ioutil"
    "net/http"
    "net/url"
    "strconv"
    "strings"
    "testing"
)

func TestContactForm(t *testing.T) {
    testURL := testHost + "/contact"
    expectedTokenString := "The contact form has been successfully completed."

    form := url.Values{}
    form.Add("firstName", "Isomorphic")
    form.Add("lastName", "Gopher")
    form.Add("email", "devnull@test.com")
    form.Add("messageBody", "This is a message sent from the automated contact form test.")

    req, err := http.NewRequest("POST", testURL,
        strings.NewReader(form.Encode()))
    if err != nil {
        t.Errorf("Failed to create new POST request to URL: %s, with error: %s", testURL, err)
    }

    req.Header.Add("Content-Type", "application/x-www-form-urlencoded")
    req.Header.Add("Content-Length", strconv.Itoa(len(form.Encode())))

    hc := http.Client{}
    response, err := hc.Do(req)
    if err != nil {
```

```
    t.Errorf("Failed to make POST request to URL: %s, with error: %s",
    testURL, err)
}

defer response.Body.Close()
contents, err := ioutil.ReadAll(response.Body)

if err != nil {
    t.Errorf("Failed to read response body contents with error: %s",
    err)
}

if strings.Contains(string(contents), expectedTokenString) == false {
    t.Errorf("Could not find expected string token: \"%s\"",
    expectedTokenString)
}
}
```

Again, the test is very similar to the previous test that we implemented, except that we are populating all the form fields and providing a properly formatted email address. We declare and initialize the `expectedTokenString` variable to the confirmation string that we expect to be printed out in the response body, after conducting a successful form submission. The last `if` conditional block of the function checks whether the response body contains the `expectedTokenString`. If it does not, then the test will fail.

These are the four possible conditions that can cause this test to fail:

- If the POST request could not be created
- If the POST request failed due to connectivity issues with the web server
- If the web client failed to read the response body of the web page response returned from the web server
- If the expected confirmation message could not be found in the body of the web page

Again, if any one of these errors are encountered, this test will fail.

We can run the test by issuing the following command:

```
$ go test - run TestContactForm
```

From examining the output after running the test, we can see that the test has passed:

```
$ go test - run TestContactForm
PASS
ok github.com/EngineerKamesh/igb/igweb/tests 0.012s
```

You can run all the tests in the test suite by simply issuing the `go test` command inside the `tests` directory:

```
$ go test
PASS
ok github.com/EngineerKamesh/igb/igweb/tests 0.011s
```

Up to this point, we have written tests to cover the baseline set of functionality to test the server-side web application. Now, it's time to focus on testing the client-side application.

## Testing the client-side functionality

As covered in [Chapter 1, Isomorphic Web Applications with Go](#), after the initial page load, subsequent navigations on the website are served using single page application architecture. This means that an XHR call is initiated to a Rest API endpoint that provides the necessary data to render content that will be displayed on the web page. In the case where the client-side handler displays the products listing page, for example, a Rest API endpoint is utilized to fetch the list of products to display. In some cases, a Rest API endpoint is not even necessary since the page content only requires rendering a template. One such example of this is when a user accesses the contact form by clicking on the **Contact** link in the navigation bar. In this situation, we simply render the contact form template and display the contents in the primary content area.

Let's take a moment and think about all the baseline functionality that we would need to test on the client-side. We need to verify that the client-side routes are functional and that the proper page is rendered for each route, similar to how we verified the server-side routes in the previous section. In addition to that, we need to confirm that the client-side form validation is working for the contact form as well as testing the scenario of a valid form submission. The functionality to add and remove items, to and from the shopping cart, is currently implemented on the client-side only. This means that we will have to write tests to verify that this functionality is working as expected. Another feature that is currently only available on the client-side, is the live chat feature. We must validate that the user can communicate with the live chat bot, that the bot replies, and that the conversation is maintained as the user navigates to different sections of the website.

Finally, we have to test our collection of cogs. We have to make sure that the time ago cog, shows a time instance in human understandable format. We have to verify that the live clock cog is functioning properly. We have to verify that the date picker cog appears when the **Time Sensitivity Date** field is clicked. We have to verify that the carousel cog appears on the home page. Finally, we have to verify that the notify cog is displaying notifications properly when adding and removing items to and from the shopping cart.

Therefore on the client-side, the baseline set of functionality that we'll test for, includes the following items:

1. Verifying client-side routing and template rendering
2. Verifying the contact form
3. Verifying the shopping cart functionality
4. Verifying the live chat feature
5. Verifying the time ago cog
6. Verifying the live clock cog
7. Verifying the date picker cog
8. Verifying the carousel cog
9. Verifying the notify cog

In order to perform automated testing on the client-side, which includes user interactions, we need a tool that has a JavaScript runtime built into it. Therefore, we cannot use `go test` when testing the client-side functionality.

We will use CasperJS to perform automated testing on the client-side.

## CasperJS

CasperJS is an automated testing tool that sits on top of PhantomJS, a headless web browser that is used for automating user interactions. CasperJS allows us to write tests using assertions, and organize the tests, so that they can all be run together sequentially. After the tests are run, we can receive a summary of how many tests passed versus how many tests failed. In addition to this, CasperJS can tap into the functionality within PhantomJS to take web page screenshots as tests are being conducted. This allows human users to visually evaluate a test run.

In order to install CasperJS, we must first install NodeJS and PhantomJS.

You can install NodeJS by downloading the NodeJS installer for your operating system from this link: <https://nodejs.org/en/download/>.

Once NodeJS is installed, you can install PhantomJS by issuing the following command:

```
$ npm install -g phantomjs
```

You may verify that `phantomjs` has been installed properly, by issuing the following command to view the version number of PhantomJS installed on your system:

```
$ phantomjs --version
2.1.1
```

Once you have verified that PhantomJS is installed on your system, you may issue the following command to install CasperJS:

```
$ npm install -g casperjs
```

To verify that `casperjs` has been installed properly, you may issue the following command to view the version number of CasperJS installed on your system:

```
$ casperjs --version
1.1.4
```

Our client-side CasperJS tests will be housed in the `client/tests` directory. Take note of the sub folders inside the `client/tests` folder:

- tests
  - go
  - js
  - screenshots

We will be writing all of our CasperJS tests in Go, and we will be placing them in the `go` folder. We will use the `build_casper_tests.sh` bash script found in the `scripts` directory to transpile the CasperJS tests implemented in Go, to their equivalent JavaScript representation. The produced JavaScript source files will be placed in the `js` folder. Many of the tests we will create, will generate screenshots of the test run in progress, and these screenshot images will be stored in the `screenshots` folder.

You should run the following command, to make the `build_casper_tests.sh` bash script executable:

```
$ chmod +x $IGWEB_APP_ROOT/scripts/build_casper_tests.sh
```

Anytime we write a CasperJS test in Go, or make changes to it, we must execute the `build_casper_tests.sh` bash script:

```
$ $IGWEB_APP_ROOT/scripts/build_casper_tests.sh
```

Before we get started writing CasperJS tests, let's take a look at the `caspertest.go` source file found in the `client/tests/go/caspertest` directory:

```
package caspertest

import "github.com/gopherjs/gopherjs/js"

type ViewportParams struct {
    *js.Object
    Width int `js:"width"`
    Height int `js:"height"`
}
```

The `ViewportParams` struct will be used to define the web browser's viewport dimensions. We will use a dimension of  $1440 \times 960$  to simulate a desktop viewing experience for all of the client-side tests. The ramifications of setting the viewport dimensions can readily be seen by viewing the screenshots generated, after running a CasperJS test that generates one or more screenshots.

Now, let's get started with writing client-side tests using CasperJS.

## Verifying client-side routing and template rendering

The CasperJS test that we implemented in Go, to test the client-side routing, can be found in the `routes_test.go` source file in the `client/tests/go` directory.

In the import grouping, notice that we include the `caspertestjs` package, where we defined the `ViewportParams` struct, and we include the `js` package:

```
package main

import (
    "strings"

    "github.com/EngineerKamesh/igb/igweb/client/tests/go/caspertest"
    "github.com/gopherjs/gopherjs/js"
)
```

We will use the functionality found in the `js` package, quite extensively, to tap into the CasperJS functionality since there are currently no GopherJS bindings available for CasperJS.

We will define a JavaScript function called `wait`, which is responsible for waiting until the primary content `div` container gets loaded in the remote DOM:

```
var wait = js.MakeFunc(func(this *js.Object, arguments []*js.Object)
interface{} {
    this.Call("waitForSelector", "#primaryContent")
    return nil
})
```

We declare and initialize the `casper` variable to the `casper` instance, a JavaScript object, which has been populated in the remote DOM, upon executing CasperJS:

```
var casper = js.Global.Get("casper")
```

We implement the client-side routing test in the `main` function. We start by declaring a `routesTokenMap` (similar to what we did on the server-side route test), having a type of `map[string]string`:

```
func main() {
    routesTokenMap := map[string]string{"/": "IGWEB", "/index": "IGWEB",
        "/products": "Add To Cart", "/product-detail/swiss-army-knife":
        "Swiss Army Knife", "/about": "Molly", "/contact": "Contact",
        "/shopping-cart": "Shopping Cart"}
```

The key represents a client-side route, and the value for a given key represents the expected string token that should be rendered on the web page, when the given client-route has been accessed.

Using the following code, we set the web browser's viewport size:

```
viewportParams := &casperTest.ViewportParams{Object:
js.Global.Get("Object").New()
viewportParams.Width = 1440
viewportParams.Height = 960
casper.Get("options").Set("viewportSize", viewportParams)
```

Note that PhantomJS uses a default viewport of  $400 \times 300$ . We have to override this value since we will be simulating a desktop viewing experience.

We will use the `tester` module of CasperJS when writing our tests. The `Tester` class provides an API, for unit and functional testing, and can be accessed by the `test` property of the `casper` instance. The full documentation for the `tester` module is available at this link: <http://docs.casperjs.org/en/latest/modules/tester.html>.

We call the `test` object's `begin` method to start a suite of planned tests:

```
casper.Get("test").Call("begin", "Client-Side Routes Test Suite", 7,
  func(test *js.Object) {
    casper.Call("start", "http://localhost:8080", wait)
  })
}
```

The first parameter supplied to the `begin` method is the description of the test suite. We have provided a description of "Client-Side Routes Test Suite".

The second parameter represents the number of tests planned. Here we have specified that there will be a total of seven tests since that is how many client-side routes we will be testing. If the number of planned tests does not match the number of actual tests performed, then CasperJS will consider it to be a *dubious* error, so it's always a good practice to make sure that the proper number of tests planned is set correctly. We will show you how to count the number of tests performed in this example.

The third parameter is a JavaScript callback function that contains the suite of tests that will be performed. Note that the callback function gets the `test` instance, a JavaScript object, as an input argument. Inside this function we call the `casper` object's `start` method. This starts up Casper and opens up the URL specified in the first input argument to the method. The second input argument to the `start` method is considered the next step, a JavaScript callback function that will run, right after accessing the URL. The next step that we specified is the `wait` function, which we had created earlier. This will have the effect of accessing the URL to the IGWEB home page, and waiting until the primary content `div` container is available in the remote DOM.

At this point, we can start our tests. We range through each route and `expectedString` pair in the `routesTokenMap`:

```
for route, expectedString := range routesTokenMap {
  func(route, expectedString string) {
```

We call the `then` method of the `casper` object to add a new navigation step to the stack:

```
casper.Call("then", func() {
  casper.Call("click", "a[href^='"+route+"']")
})
```

Inside the function that represents the navigation step, we call the `casper` object's `click` method. The `click` method will trigger a mouse click event on the element that matches the provided CSS selector. We create a CSS selector for each route, and it will match a link on the body of the web page. The CSS selector allows us to simulate the scenario of the user clicking on the navigation links.

The two routes that are not part of the navigation links are the `/` and `/product-detail/swiss-army-knife` routes. The CSS selector for the `/` route will match the link declared for the website's logo in the upper left-hand corner of the web page. When this scenario is tested, it is equivalent to the user clicking on the logo of the website. In this case of the link to the product detail page for the Swiss Army Knife, `/product-detail/swiss-army-knife`, will be found in the primary content area `div` once the content for the products page has been rendered. When this scenario is tested, it is equivalent of the user clicking on the Swiss Army Knife image on the products listing page.

In the next navigation step, we will generate a screenshot of the tests case, and check if `expectedString` is found in the body of the web page:

```
casper.Call("then", func() {
  casper.Call("wait", 1800, func() {
    routeName := strings.Replace(route, `/`, "", -1)
    screenshotName := "route_render_test_" + routeName + ".png"
    casper.Call("capture", "screenshots/" + screenshotName)
    casper.Get("test").Call("assertTextExists", expectedString,
      "Expected text \"\"+expectedString+"\", in body of web page,
      when accessing route: "+route)
  })
}) (route, expectedString)
})
```

Here, we call the `casper` object's `capture` method to supply the path of the generated screenshot image. A screenshot will be generated for each route that we test, so we will have a total of seven screenshot images generated from this test.

Notice that we call `casper`'s `wait` method to introduce a 1800 millisecond delay, and we supply a `then` callback function. In conversational English, we can interpret this call as, "Wait 1800 milliseconds, and then do this." Inside the `then` callback function that we have supplied, we make a call to the `assertTextExists` method on `casper`'s `test` object (the `tester` module). In the `assertTextExists` method call, we supply the `expectedString` that should exist in the body of the web page, and the second parameter is a message describing the test. We have added the 1800 millisecond delay, to provide sufficient time for the page content to be displayed on the web page.

Take note that whenever any type of `assert` method call, in the family of `assert` methods in the `casper` `tester` module is called, it counts as a single test. Recall that when we called the `tester` module's `begin` method, we supplied a value of 7, to indicate that there were to be 7 expected tests that would take place in this test suite. So the amount of `assert` method calls that you have in your test, must match the number of expected tests that will be conducted, otherwise you will get a dubious error when performing a run of the test suite.

We make a call to the `casper` object's `run` method to run the test suite:

```
casper.Call("run", func() {
  casper.Get("test").Call("done")
})
```

Note that we supply a callback function to the `run` method. This callback function will be called when all steps have completed running. Inside the callback function, we call the `tester` module's `done` method to signify the end of the test suite. Remember that in a CasperJS test, whenever we call the `begin` method on the `tester` module, there must be a corresponding place in the test where we call the `tester` module's `done` method. If we forget to leave the method call to `done`, the program will hang and we'll have to break out of the program (using a *Ctrl + C* keystroke).

We must transpile the test to its JavaScript equivalent, and we can do so by running the `build_casper_tests.sh` bash script:

```
$ $IGWEB_APP_ROOT/scripts/build_casper_tests.sh
```

The bash script will transpile all of the CasperJS tests written in Go residing in the `client/tests/go` directory, and place the produced JavaScript source files in the `client/tests/js` directory. We will omit this step in subsequent test runs. Just keep in mind that if you make changes to any of the tests that you need to re-run this script so that the changes will take effect, next time you run the suite of tests.

We can run the test to check the client-side routes by issuing the following commands:

```
$ cd $IGWEB_APP_ROOT/client/tests
$ casperjs test js/routes_test.js
```

Figure 10.1 shows a screenshot of running the client-side routes test suite:

```
balans0:~ kamesh$ cd $IGWEB_APP_ROOT/client/tests
balans0:tests kamesh$ casperjs test js/routes_test.js
Test file: js/routes_test.js
# Client-Side Routes Test Suite
PASS Expected text "IGWEB", in body of web page, when accessing route: /
PASS Expected text "IGWEB", in body of web page, when accessing route: /index
PASS Expected text "Add To Cart", in body of web page, when accessing route: /products
PASS Expected text "Swiss Army Knife", in body of web page, when accessing route: /product-detail/swiss-army-knife
PASS Expected text "Molly", in body of web page, when accessing route: /about
PASS Expected text "Contact", in body of web page, when accessing route: /contact
PASS Expected text "Shopping Cart", in body of web page, when accessing route: /shopping-cart
PASS Client-Side Routes Test Suite (7 tests)
PASS 7 tests executed in 15.956s, 7 passed, 0 failed, 0 dubious, 0 skipped.
balans0:tests kamesh$
```

Figure 10.1: Running the client-side routes test suite

The produced screenshot images from the test can be found in the `client/tests/screenshots` folder. The screenshots come in handy because they allow a human user to visually look over the test results.

Figure 10.2 shows a screenshot image of testing the / route:

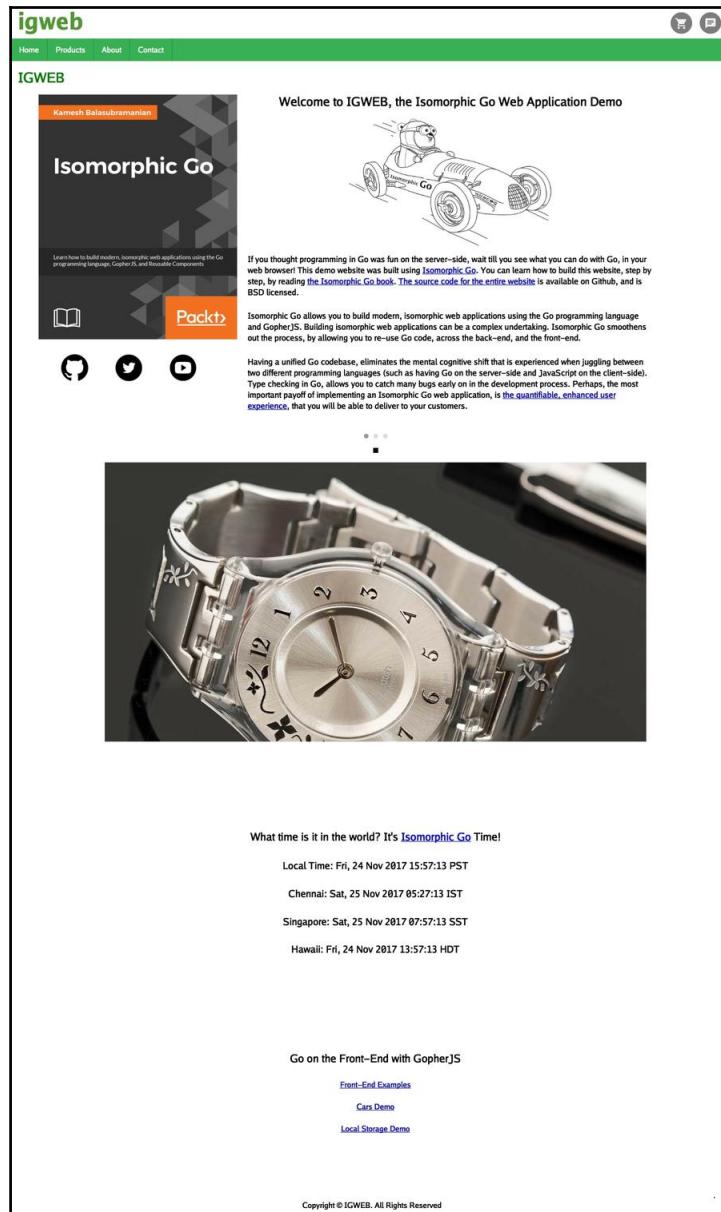


Figure 10.2: Testing the / route

Figure 10.3 shows a screenshot image of testing the /index route. Note that the page rendering is identical to Figure 10.2 as it should be:

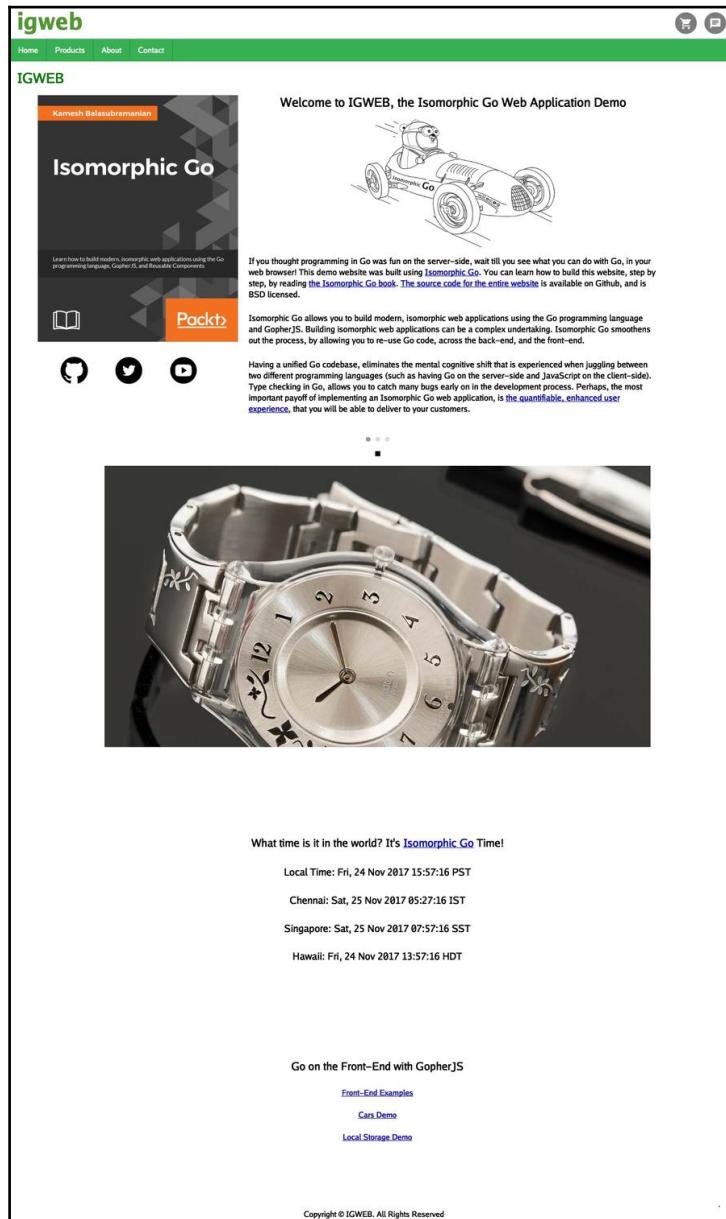


Figure 10.3: Testing the /index route

Notice that by supplying the delay time of 1800 milliseconds, we gave enough time for the carousel cog and the live clock cogs to load. Later in this chapter, you'll learn how to test these cogs.

Figure 10.4 shows a screenshot image of testing the /products route:

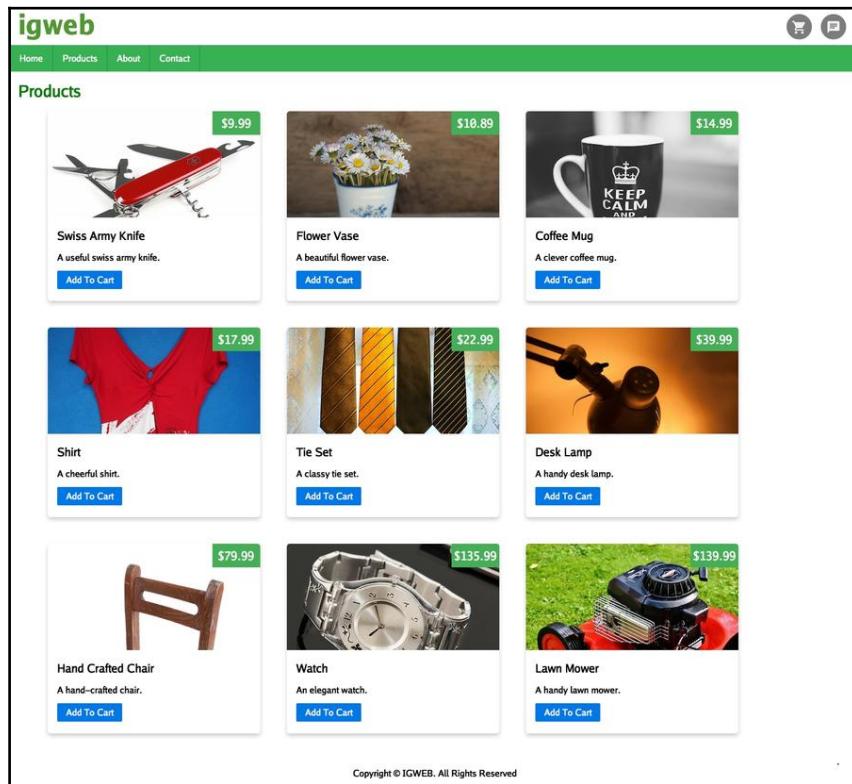


Figure 10.4: Testing the /products route

From this test, we can visually confirm that the products listing page has loaded fine. The next test step will click on the image of the Swiss Army Knife to navigate to its product detail page.

Figure 10.5 shows a screenshot image of testing the /product-detail/swiss-army-knife route:



Figure 10.5: Testing the /product-detail route

Figure 10.6 shows a screenshot image of testing the /about route:

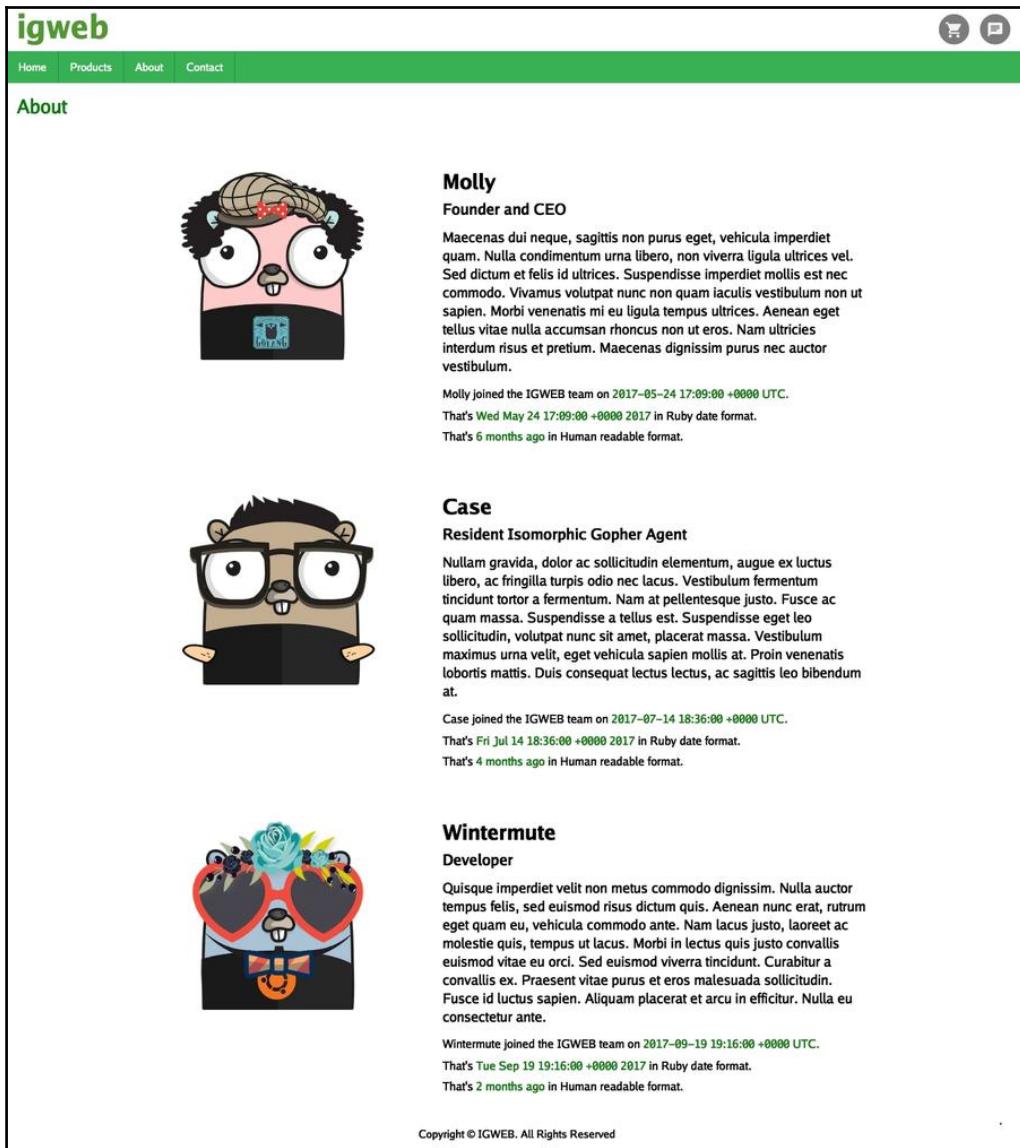
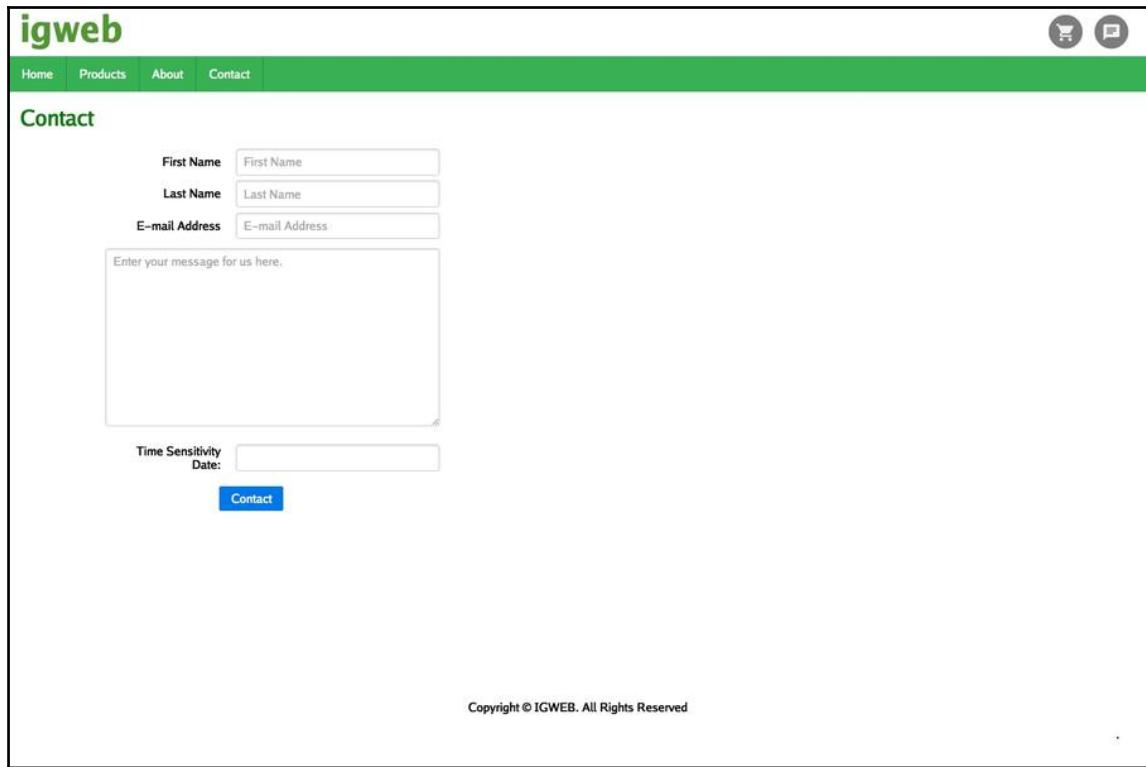


Figure 10.6: Testing the /about route

Notice that the time ago cog has rendered properly for all three gophers.

Figure 10.7 shows a screenshot image of testing the /contact route:



The screenshot shows the 'Contact' page of a web application named 'igweb'. The page has a green header with the 'igweb' logo and navigation links for Home, Products, About, and Contact. The Contact page displays a form with fields for First Name, Last Name, E-mail Address, and a large text area for a message. Below the message area is a field for Time Sensitivity Date. A blue 'Contact' button is at the bottom. The footer contains the copyright notice 'Copyright © IGWEB. All Rights Reserved'.

First Name

Last Name

E-mail Address

Enter your message for us here.

Time Sensitivity Date:

Contact

Copyright © IGWEB. All Rights Reserved

Figure 10.7: Testing the /contact route

Figure 10.8 shows a screenshot image of testing /shopping-cart route.

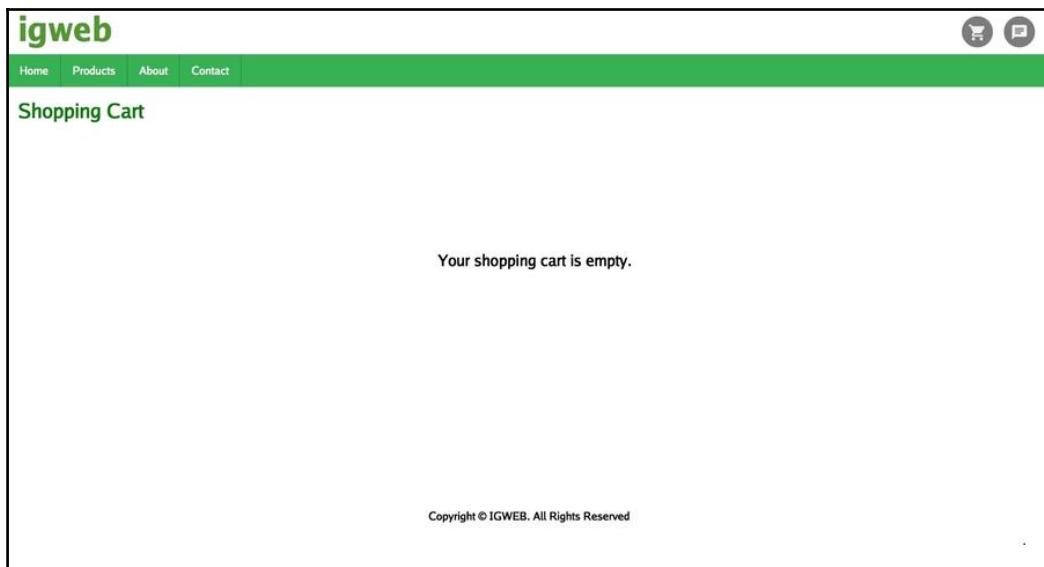


Figure 10.8: Testing the /shopping-cart route

From the visual confirmation provided by the screenshots, we can now be confident that the client-side routing is working as expected. In addition to that, the generated screenshots help us visually confirm that template rendering is functioning properly. Let's take a look at verifying the contact form functionality now.

## Verifying the contact form

The test that we implemented to verify the contact form functionality, can be found in the `contactform_test.go` source file in the `client/tests/go` directory.

In this test, we defined the `FormParams` struct, which represents the form parameters that the contact form should be filled with, when conducting our test steps:

```
type FormParams struct {
    *js.Object
    FirstName string `js:"firstName"`
    LastName string `js:"lastName"`
    Email string `js:"email"`
    MessageBody string `js:"messageBody"`
}
```

We've created a JavaScript `wait` function to ensure that the test runner will wait for the primary content `div` container to have loaded, prior to running other steps:

```
var wait = js.MakeFunc(func(this *js.Object, arguments []*js.Object) interface{} {
    this.Call("waitForSelector", "#primaryContent")
    return nil
})
```

We will introduce the following three JavaScript functions to populate the contact form's fields, depending on the type of test we are conducting:

- `fillOutContactFormWithPoorlyFormattedEmailAddress`
- `fillOutContactFormPartially`
- `filloutContactFormCompletely`

The `fillOutContactFormWithPoorlyFormattedEmailAddress` function, as the name implies, will supply an invalid email address to the `email` field:

```
var fillOutContactFormWithPoorlyFormattedEmailAddress =
js.MakeFunc(func(this *js.Object, arguments []*js.Object) interface{} {
    params := &FormParams{Object: js.Global.Get("Object").New()}
    params.FirstName = "Isomorphic"
    params.LastName = "Gopher"
    params.Email = "dev@null@test@test.com"
    params.MessageBody = "Sending a contact form submission using CasperJS
and PhantomJS"
    this.Call("fill", "#contactForm", params, true)
    return nil
})
```

Notice that we created a new `FormParams` instance, and populated the `FirstName`, `LastName`, `Email`, and `MessageBody` fields. Take particular note that we have supplied an invalid email address for the `Email` field.

In the context of this function, the `this` variable represents the `tester` module. We make a call to the `tester` module's `fill` method, supplying the CSS selector to the contact form, the `params` object, and a Boolean value of `true` to indicate that the form should be submitted.

After filling in and submitting the form, we expect the client-side form validation to present us with an error message indicating that we have provided an invalid email address.

The `fillOutContactFormPartially` function will fill out the contact form partially, leaving some required fields unfilled, resulting in an incomplete form:

```
var fillOutContactFormPartially = js.MakeFunc(func(this *js.Object,
  arguments []*js.Object) interface{} {
  params := &FormParams{Object: js.Global.Get("Object").New()}
  params.FirstName = "Isomorphic"
  params.LastName = ""
  params.Email = "devnull@test.com"
  params.MessageBody = ""
  this.Call("fill", "#contactForm", params, true)
  return nil
})
```

Here, we create a new `FormParams` instance, and take note that we have provided an empty string value for the `LastName` and `MessageBody` fields.

After filling in and submitting the form, we expect the client-side form validation to present us with an error message indicating that we have not filled in these two required fields.

The `fillOutContactFormCompletely` function will fill out all fields of the contact form and will include a properly formatted email address:

```
var fillOutContactFormCompletely = js.MakeFunc(func(this *js.Object,
  arguments []*js.Object) interface{} {
  params := &FormParams{Object: js.Global.Get("Object").New()}
  params.FirstName = "Isomorphic"
  params.LastName = "Gopher"
  params.Email = "devnull@test.com"
  params.MessageBody = "Sending a contact form submission using CasperJS
  and PhantomJS"
  this.Call("fill", "#contactForm", params, true)
  return nil
})
```

Here we create a new `FormParams` instance, and populate all the fields of the contact form. In the case of the `Email` field, we ensure to supply a properly formatted email address.

After filling in and submitting the form, we expect the client-side form validation to green-light the form, which behind the scenes, will initiate an XHR call to the rest API endpoint to verify that the contact form has been properly filled using the server-side form validation. We expect the server-side validation to also green-light the form field values, resulting in a confirmation message. Our test will pass, if we can successfully verify that we have obtained the confirmation message.

As with the previous example, we start by declaring the viewport parameters and setting the viewport size of the web browser:

```
func main() {  
  
    viewportParams := &casperTest.ViewportParams{Object:  
        js.Global.Get("Object").New()  
    }  
    viewportParams.Width = 1440  
    viewportParams.Height = 960  
    casper.Get("options").Set("viewportSize", viewportParams)
```

Notice that we call the `tester` module's `begin` method to start the tests in the **Contact Form Test Suite**:

```
casper.Get("test").Call("begin", "Contact Form Test Suite", 4,  
    func(test *js.Object) {  
        casper.Call("start", "http://localhost:8080/contact", wait)  
    })
```

We supply the `begin` method with the description of the test, "Contact Form Test Suite". We then supply the number of expected tests in this suite, which is 4. Remember that this value corresponds to the number of tests we conduct. The number of tests conducted can be ascertained by the number of calls we make to a method belonging to the `tester` module's `assert` family of methods. We provide the `then` callback function, where we call the `start` method on the `casper` object, providing the URL to the **Contact** page and supplying the `wait` function to indicate that we should wait for the primary content `div` container to load before conducting any test steps.

The first scenario that we test is to check the client-side validation when a poorly formatted email address is provided:

```
casper.Call("then",  
    fillOutContactFormWithPoorlyFormattedEmailAddress)  
casper.Call("wait", 450, func() {  
    casper.Call("capture",  
        "screenshots/contactform_test_invalid_email_error_message.png")  
    casper.Get("test").Call("assertSelectorHasText", "#emailError",  
        "The e-mail address entered has an improper syntax", "Display e-  
        mail address syntax error when poorly formatted e-mail entered.")  
})
```

We call the `casper` object's `then` method, providing the `fillOutContactFormWithPoorlyFormattedEmailAddress` JavaScript function as the `then` callback function. We wait for 450 milliseconds for a result, capture a screenshot of the test run (shown in *Figure 10.10*), and then call the `assertSelectorHasText` method on the `tester` module, providing a CSS selector of the element containing the error message, along with the expected text the error message should have displayed, followed with a description of the test we are conducting.

The second scenario that we test, is to check the client-side validation when an incomplete form has been submitted:

```
casper.Call("then", fillOutContactFormPartially)
casper.Call("wait", 450, func() {
  casper.Call("capture",
  "screenshots/contactform_test_partially_filled_form_errors.png")
  casper.Get("test").Call("assertSelectorHasText", "#lastNameError",
  "The last name field is required.", "Display error message when the
  last name field has not been filled out.")
  casper.Get("test").Call("assertSelectorHasText",
  "#messageBodyError", "The message area must be filled.", "Display
  error message when the message body text area has not been filled
  out.")
})
```

We call the `casper` object's `then` method, providing the `fillOutContactFormPartially` JavaScript function, as the `then` callback function. We wait for 450 milliseconds for a result, capture a screenshot of the test run (shown in *Figure 10.11*), and conduct two tests within this scenario.

In the first test, we call the `assertSelectorHasText` method on the `tester` module, providing a CSS selector of the element containing the error message for the last name field along with the expected text, the error message should have, followed with a description of the test. In the second test, we call the `assertSelectorHasText` method on the `tester` module, providing a CSS selector of the element containing the error message for the message body text area, the expected text the error message should have, followed with a description of the test.

The third scenario that we test is to check that a confirmation message has been displayed upon the submission of a properly filled out contact form:

```
casper.Call("then", fillOutContactFormCompletely)
casper.Call("wait", 450, func() {
  casper.Call("capture",
  "screenshots/contactform_confirmation_message.png")
  casper.Get("test").Call("assertSelectorHasText", "#primaryContent
  h1", "Confirmation", "Display confirmation message after submitting
  contact form.")
})
```

We call the `casper` object's `then` method, providing the `fillOutContactFormCompletely` JavaScript function, as the `then` callback function. We wait for 450 milliseconds for a result, capture a screenshot of the test run (shown in *Figure 10.12*), and call the `casper` object's `assertSelectorHasText` method. We provide the CSS selector `#primaryContent h1`, since the confirmation message will be inside the `<h1>` tag. We supply the expected text that the confirmation message should contain, being `"Confirmation"`. Finally, we provide a description of the test for the last parameter of the `assertSelectorHasText` method.

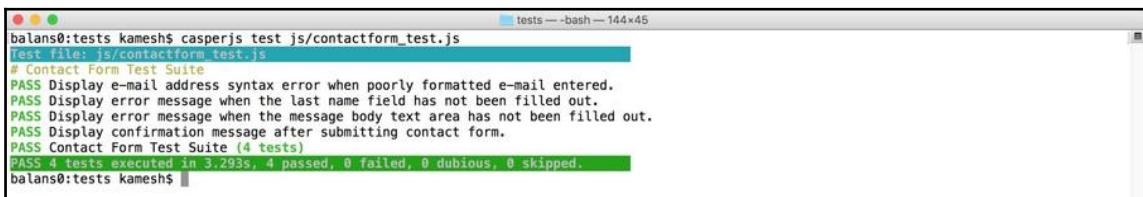
To signify the end of the test suite, we call the `casper` object's `run` method, and inside the `then` callback function, we call the tester module's `done` method:

```
casper.Call("run", func() {
  casper.Get("test").Call("done")
})
```

Assuming that you are inside the `client/tests` folder, you can issue the following command to run the contact form test suite:

```
$ casperjs test js/contactform_test.js
```

*Figure 10.9* shows a screenshot image of running the contact form test suite:



```
balans0:tests kamesh$ casperjs test js/contactform_test.js
Test file: js/contactform_test.js
# Contact Form Test Suite
PASS Display e-mail address syntax error when poorly formatted e-mail entered.
PASS Display error message when the last name field has not been filled out.
PASS Display error message when the message body text area has not been filled out.
PASS Display confirmation message after submitting contact form.
PASS Contact Form Test Suite (4 tests)
PASS 4 tests executed in 3.293s, 4 passed, 0 failed, 0 dubious, 0 skipped.
balans0:tests kamesh$
```

Figure 10.9: Running the contact form test suite

Figure 10.10 shows the generated screenshot image of running the first test which checks that the client-side form validation properly detects an improperly formatted email address:

The screenshot shows the 'Contact' page of the igweb application. The page has a green header with 'igweb' and navigation links for Home, Products, About, and Contact. The main content area is titled 'Contact' and contains a form with the following fields:

- First Name: Isomorphic
- Last Name: Gopher
- E-mail Address: dev@null@test@test.com

A red validation message next to the email field states: "The e-mail address entered has an improper syntax." Below the form, a note says: "Sending a contact form submission using CasperJS and PhantomJS". At the bottom of the page, there is a "Time Sensitivity" section with a dropdown menu, a "Contact" button, and a copyright notice: "Copyright © IGWEB. All Rights Reserved".

Figure 10.10: Testing the email validation syntax

Figure 10.11 shows the generated screenshot image of running the second test and third test, which checks that the client-side form validation properly detects that the last name field and the message body text area has not been filled in:

The screenshot shows the 'Contact' page of the igweb application. The page has a green header with the logo 'igweb' and navigation links for Home, Products, About, and Contact. The Contact page displays a form with the following fields and validation errors:

- First Name:** Isomorphic (Valid)
- Last Name:** (Empty) The last name field is required.
- E-mail Address:** devnull@test.com (Valid)
- Message Area:** (Empty) The message area must be filled.
- Time Sensitivity Date:** (Empty)

At the bottom of the form is a blue 'Contact' button. The footer of the page contains the text 'Copyright © IGWEB. All Rights Reserved'.

Figure 10.11: Test to verify if the form validation detects required fields that have not been filled out

Figure 10.12 shows the generated screenshot image of running the fourth test, which checks that the confirmation message has been displayed upon successfully filling out and submitting the contact form:

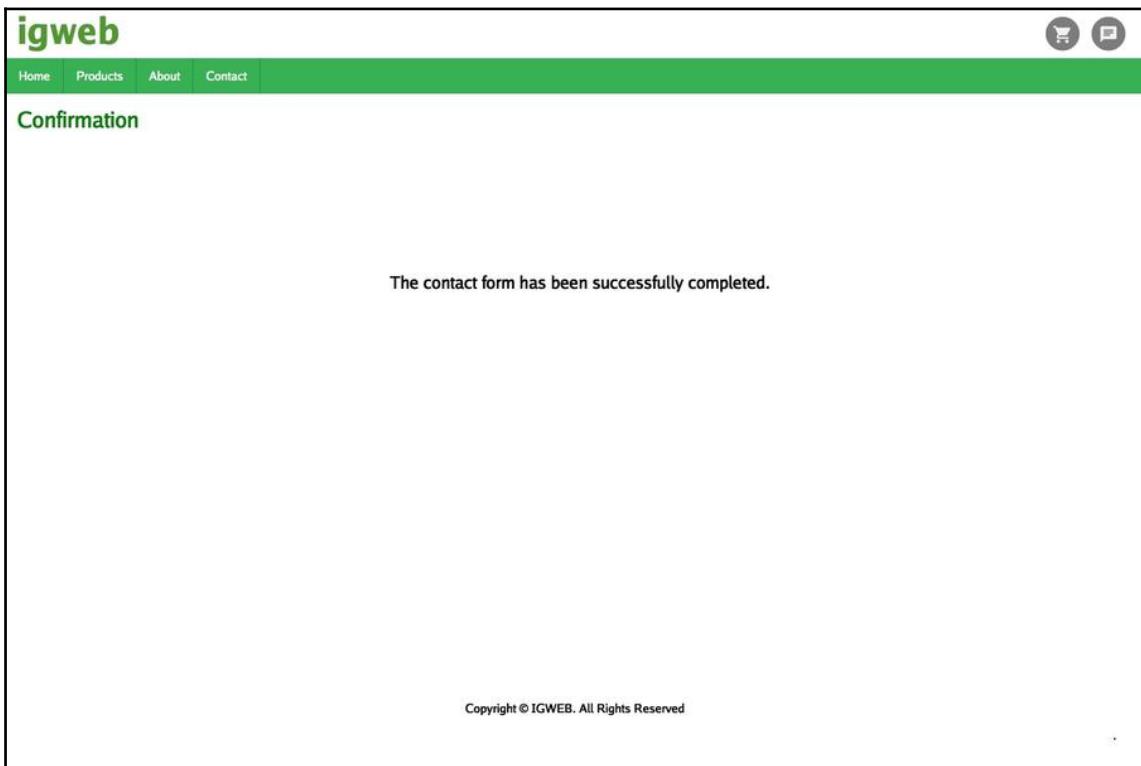


Figure 10.12: Test to verify the confirmation message

Now that we've verified the client-side validation functionality for the contact form, let's look into implementing a CasperJS test suite for the shopping cart functionality.

## Verifying the shopping cart functionality

In order to verify the shopping cart functionality, we must be able to add a product multiple times to the shopping cart, check that the product exists in the shopping cart with the proper quantity of the product displayed, and have the ability to remove the product from the shopping cart. Therefore, there are 3 expected tests that we will need in the shopping cart test suite.

The `main` function in the `shoppingcart_test.go` source file, located in the `client/tests/go` directory, implements the shopping cart test suite:

```
func main() {  
  
    viewportParams := &caspertest.ViewportParams{Object:  
        js.Global.Get("Object").New()  
    }  
    viewportParams.Width = 1440  
    viewportParams.Height = 960  
    casper.Get("options").Set("viewportSize", viewportParams)  
  
    casper.Get("test").Call("begin", "Shopping Cart Test Suite", 3,  
        func(test *js.Object) {  
            casper.Call("start", "http://localhost:8080/products", wait)  
        })  
}
```

Inside the `main` function, we set the web browser's viewport size. We start a new test suite by calling the `begin` method on the `casper` object. Note that we have indicated that there are 3 expected tests in this test suite. Inside the `then` callback function that constitutes the last argument to the `begin` method, we call the `start` method of the `casper` object, providing the URL to the products listing page, and providing the JavaScript `wait` function, as the `then` callback function. This will cause the program to wait until the primary content `div` container has loaded in the DOM, before conducting any tests.

With the following code, we add three Swiss Army Knives to the shopping cart:

```
for i := 0; i < 3; i++ {  
    casper.Call("then", func() {  
        casper.Call("click", ".addToCartButton:first-child")  
    })  
}
```

Notice that we have passed the `click` method of the `casper` object with the CSS selector `".addToCartButton:first-child"`, to ensure that the Swiss Army Knife product is clicked on, since it is the first product displayed on the products listing page.

In order to verify that the Swiss Army Knives were properly placed in the shopping cart, we need to navigate to the shopping cart page:

```
casper.Call("then", func() {
    casper.Call("click", "a[href='/shopping-cart']")
})
```

Our first test consists of verifying that the correct product type exists in the shopping cart:

```
casper.Call("wait", 207, func() {
    casper.Get("test").Call("assertTextExists", "Swiss Army Knife",
    "Display correct product in shopping cart.")
})
```

We do this by checking that the "Swiss Army Knife" text exists on the shopping cart page by calling the `assertTextExists` method on the `tester` module object and providing an expected text value of "Swiss Army Knife".

Our second test consists of verifying that the correct product quantity exists on the **Shopping Cart** page:

```
casper.Call("wait", 93, func() {
    casper.Get("test").Call("assertTextExists", "Quantity: 3", "Display
    correct product quantity in shopping cart.")
})
```

Again, we call the `tester` module object's `assertTextExists` method, passing in the expected text, "Quantity: 3".

We generate a screenshot of the shopping cart, and this screenshot (shown in *Figure 10.14*) should display the Swiss Army Knife with a quantity value of 3:

```
casper.Call("wait", 450, func() {
    casper.Call("capture", "screenshots/shoppingcart_test_add_item.png")
})
```

Our last test consists of removing an item from the shopping cart. We remove the product from the shopping cart with the following code:

```
casper.Call("then", func() {
    casper.Call("click", ".removeFromCartButton:first-child")
})
```

In order to verify that the product was successfully removed from the shopping cart, we need to check if the message indicating that the shopping cart is empty, exists on the **Shopping Cart** page:

```
casper.Call("wait", 5004, func() {
  casper.Call("capture", "screenshots/shoppingcart_test_empty.png")
  casper.Get("test").Call("assertTextExists", "Your shopping cart is
empty.", "Empty the shopping cart.")
})
```

Notice that in our call to the `tester` module object's `assertTextExists` method, we check to see whether the "Your shopping cart is empty." text exists on the web page. Prior to this, we also generate a screenshot image (shown in *Figure 10.15*), which will show us the shopping cart in the empty state.

Finally, we will signify the end of the shopping cart test suite with the following code:

```
casper.Call("run", func() {
  casper.Get("test").Call("done")
})
```

We can run the CasperJS tests for the shopping cart test suite by issuing the following command:

```
$ casperjs test js/shoppingcart_test.js
```

*Figure 10.13* shows a screenshot of the result of running the shopping cart test suite:

```
balans0:tests kamesh$ casperjs test js/shoppingcart_test.js
test file: js/shoppingcart_test.js
# Shopping Cart Test Suite
PASS Display correct product in shopping cart.
PASS Display correct product quantity in shopping cart.
PASS Empty the shopping cart.
PASS Shopping Cart Test Suite (3 tests)
PASS 3 tests executed in 6.88s, 3 passed, 0 failed, 0 dubious, 0 skipped.
balans0:tests kamesh$
```

Figure 10.13: Running the shopping cart test suite

Figure 10.14 shows the generated screenshot showing the test case, where 3 Swiss Army Knives have been successfully added to the shopping cart:

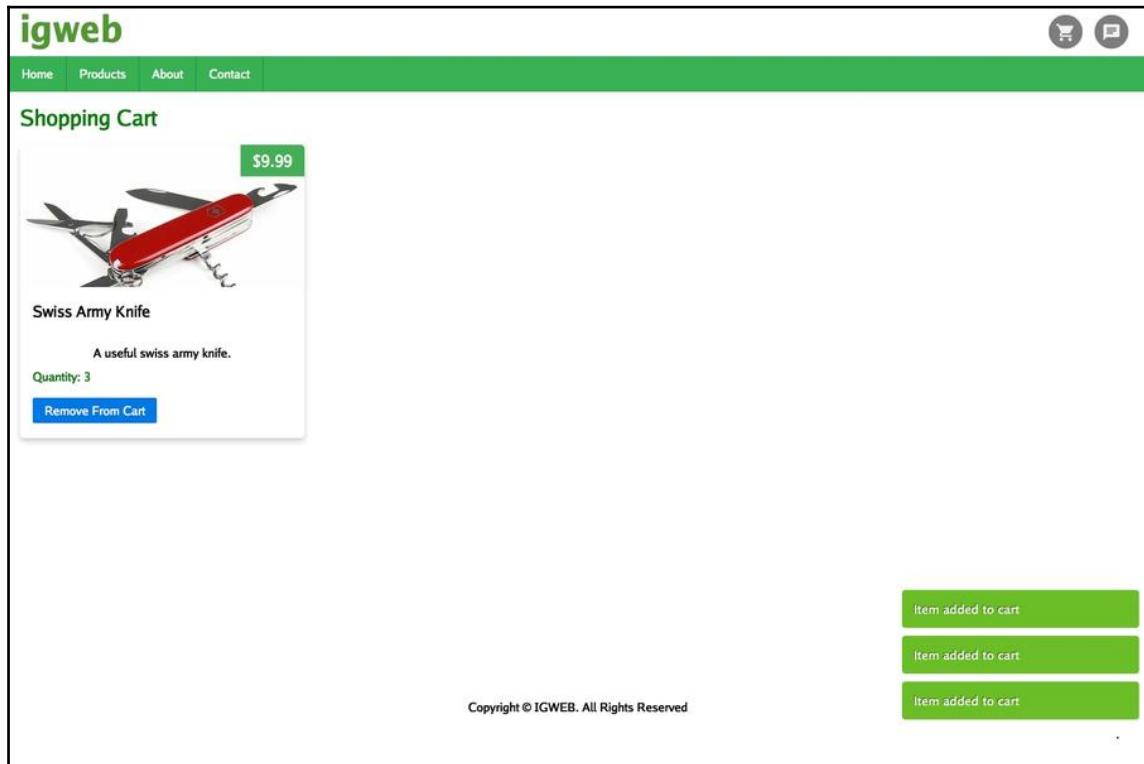


Figure 10.14: Test case to add a product multiple times to the shopping cart

Figure 10.15 shows the generated screenshot showing the test case, where the Swiss Army Knife product has been removed, thus emptying the shopping cart:

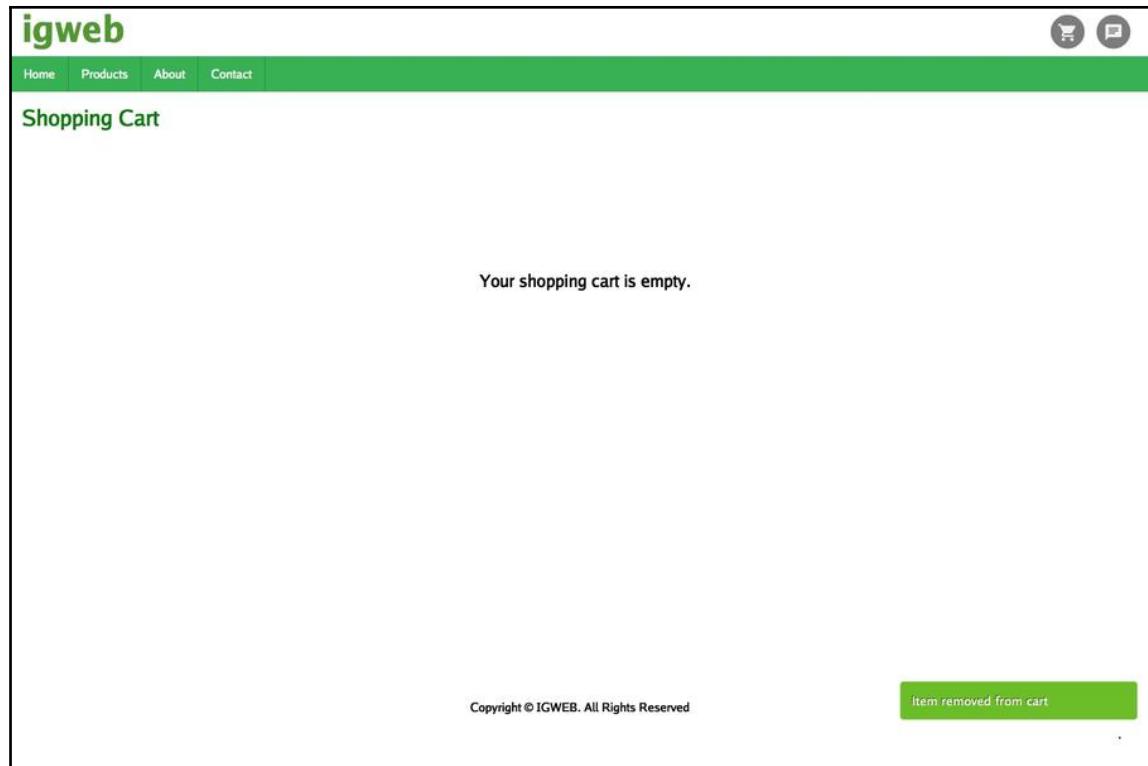


Figure 10.15: Test to verify emptying the shopping cart

Now that we've verified the functionality for the shopping cart, let's look into testing the live chat feature.

## Verifying the live chat feature

The live chat test suite consists of three tests. First, we must ensure that the chat box opens up when the live chat icon is clicked on the top bar. Second, we must ensure that the chat bot responds to us when we ask it a question. Third, we must ensure that the conversation is retained when we navigate to another section of the website.

The live chat test suite is implemented in the `livechat_test.go` source file found in the `client/tests/go` directory.

The `waitChat` JavaScript function will be used to wait for the chat box to open:

```
var waitChat = js.MakeFunc(func(this *js.Object, arguments []*js.Object)
interface{} {
    this.Call("waitForSelector", "#chatbox")
    return nil
})
```

The `askQuestion` JavaScript function will be used to send a question to the chat bot:

```
var askQuestion = js.MakeFunc(func(this *js.Object, arguments []*js.Object)
interface{} {
    this.Call("sendKeys", "input#chatboxInputField", "What is Isomorphic
Go?")
    this.Call("sendKeys", "input#chatboxInputField",
casper.Get("page").Get("event").Get("key").Get("Enter"))
    return nil
})
```

Note that we use the `sendKeys` method of the `tester` module object (the `this` variable is bound to the `tester` module object) to type the "What is Isomorphic Go" question, and we call the `sendKeys` method again to send the `enter` key (the equivalent of pressing the `enter` key on the keyboard).

Inside the `main` function, we set the web browser's viewport size and begin the test suite:

```
func main() {
    viewportParams := &casperTest.ViewportParams{Object:
js.Global.Get("Object").New()}
    viewportParams.Width = 1440
    viewportParams.Height = 960
    casper.Get("options").Set("viewportSize", viewportParams)

    casper.Get("test").Call("begin", "Live Chat Test Suite", 3, func(test
*js.Object) {
        casper.Call("start", "http://localhost:8080/index", wait)
    })
}
```

The following code will activate the live chat feature, by simulating a user click on the live chat icon on the top bar:

```
casper.Call("then", func() {
  casper.Call("click", "#livechatContainer img")
})
```

The following code will wait until the chat box has opened up, before proceeding:

```
casper.Call("then", waitChat)
```

Once the chat box has opened up, we can verify that the chat box is visible with the following code:

```
casper.Call("wait", 1800, func() {
  casper.Call("capture",
  "screenshots/livechat_test_chatbox_open.png")
  casper.Get("test").Call("assertSelectorHasText", "#chatboxTitle
  span", "Chat with", "Display chatbox.")
})
```

Note that we call the `tester` module object's `assertSelectorHasText` method, providing a CSS selector of `#chatboxTitle span` to target the chat box's title `span` element. We then check to see whether the "Chat with" text exists inside the `span` element to verify that the chat box is visible.

Notice that we have generated a screenshot image, which should show us the chat box opened up, with the chat bot providing its greeting message (shown in *Figure 10.17*).

The following code is used to verify that the chat bot provides an answer to us when we ask it a question:

```
casper.Call("then", askQuestion)
casper.Call("wait", 450, func() {
  casper.Call("capture",
  "screenshots/livechat_test_answer_question.png")
  casper.Get("test").Call("assertSelectorHasText",
  "#chatboxConversationContainer", "Isomorphic Go is the methodology
  to create isomorphic web applications", "Display the answer to
  \"What is Isomorphic Go?\"")
})
```

We call the `askQuestion` function to simulate the user typing in the "What is Isomorphic Go" question and pressing the `enter` key. We wait for 450 milliseconds, then generate a screenshot, which should show the live chat bot answering our question (shown in *Figure 10.18*). We verify that the chat bot has provided an answer by calling the `tester` module object's `assertSelectorHasText` method and providing it the CSS selector to access the `div` container, which contains the conversation and a substring of the expected answer.

Currently, we are on the home page. To test that the conversation is retained while navigating to a different section of the website, we use the following code:

```
casper.Call("then", func() {
  casper.Call("click", "a[href='/about']")
})

casper.Call("then", wait)
```

Here, we specified to navigate to the **About** page, and then wait until the primary content `div` container has loaded.

We wait for 450 milliseconds, take a screenshot image (shown in *Figure 10.19*), and then conduct the last test in our test suite:

```
casper.Call("wait", 450, func() {
  casper.Call("capture",
  "screenshots/livechat_test_conversation_retained.png")
  casper.Get("test").Call("assertSelectorHasText",
  "#chatboxConversationContainer", "Isomorphic Go is the methodology
  to create isomorphic web applications", "Verify that the
  conversation is retained when navigating to another page in the
  website.")
})
```

The last test here is a repeat of the previous test we conducted. Since we are testing that the conversation has been retained, we expect that the answer the chat bot gave us, after the last test, is preserved within the `div` container containing the conversation.

We will close the chat box by simulating a user click to the close control (the **X** found in the upper right hand corner of the chat box), so that the websocket connection is closed normally:

```
casper.Call("then", func() {
  casper.Call("click", "#chatboxCloseControl")
})
```

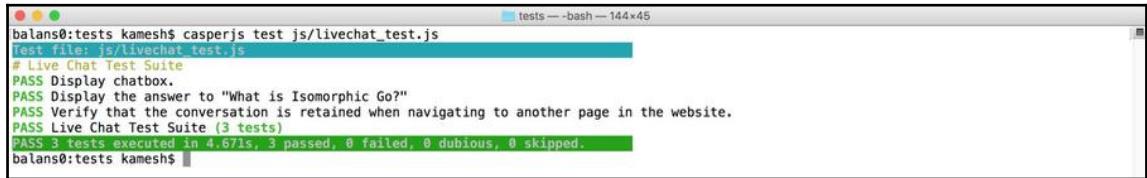
Finally, we will signify the end of the live chat test suite with the following code:

```
casper.Call("run", func() {
  casper.Get("test").Call("done")
})
```

We can run the CasperJS tests for the live chat test suite by issuing the following command:

```
$ casperjs test js/livechat_test.js
```

Figure 10.16 shows a screenshot of the result of running the live chat test suite:



```
balans0:tests kamesh$ casperjs test js/livechat_test.js
Test file: js/livechat_test.js
# Live Chat Test Suite
PASS Display chatbox.
PASS Display the answer to "What is Isomorphic Go?"
PASS Verify that the conversation is retained when navigating to another page in the website.
PASS Live Chat Test Suite (3 tests)
PASS 3 tests executed in 4.671s, 3 passed, 0 failed, 0 dubious, 0 skipped.
balans0:tests kamesh$
```

Figure 10.16: Running the live chat test suite

Figure 10.17 shows the generated screenshot showing the test case, where we check to see whether the chat box has opened:



Figure 10.17: Test to verify that the chat box appears

Figure 10.18 shows the generated screenshot showing the test case, where we check to see whether the chat bot responds to the given question:

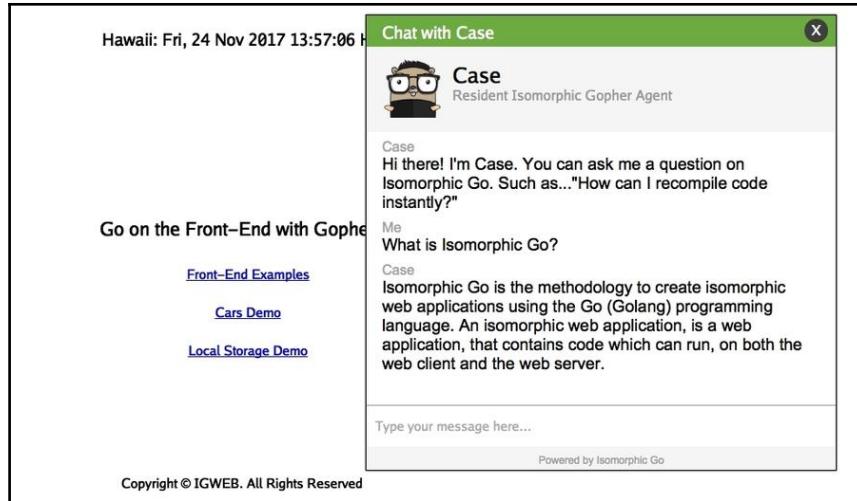


Figure 10.18: Test to verify that the chat bot responds to a question

Figure 10.19 shows the generated screenshot showing the test case, where we check to see whether the chat conversation is retained after navigating to a different page on the website:

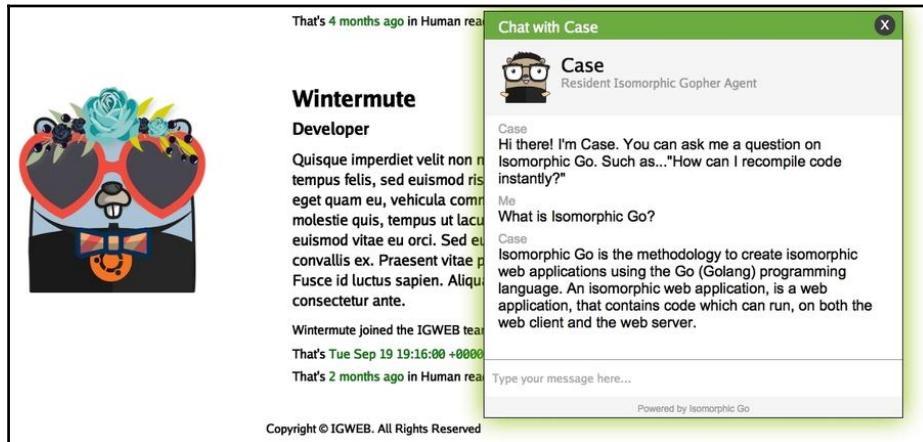


Figure 10.19: Test to see if chat conversation is retained after navigation to a different section of the website

Now that we've verified the functionality for the live chat feature, let's look into testing the cogs, starting with the time ago cog.



For the sake of brevity, the generated screenshot images shown in Figures 10.17, 10.18, 10.19, 10.21, 10.23, 10.25, 10.27, and 10.29 have been cropped.

## Verifying the time ago cog

Testing the time ago cog consists of establishing a gopher's known date of joining the IGWEB team. We will establish May 24, 2017, as Molly's start date, and use this as the basis to test the human understandable time that is displayed under Molly's bio data on the [About](#) page.

Here is the test suite for the time ago cog, which is implemented in the `humantimecog_test.go` source file found in the `client/tests/go` directory:

```
package main

import (
    "time"

    "github.com/EngineerKamesh/igb/igweb/client/tests/go/caspertest"
    humanize "github.com/dustin/go-humanize"
    "github.com/gopherjs/gopherjs/js"
)

var wait = js.MakeFunc(func(this *js.Object, arguments []*js.Object) interface{} {
    this.Call("waitForSelector", "#primaryContent")
    return nil
})

var casper = js.Global.Get("casper")

func main() {

    viewportParams := &caspertest.ViewportParams{Object:
        js.Global.Get("Object").New()
    }
    viewportParams.Width = 1440
    viewportParams.Height = 960
    casper.Get("options").Set("viewportSize", viewportParams)

    casper.Get("test").Call("begin", "Time Ago Cog Test Suite", 1,
```

```
func(test *js.Object) {
    casper.Call("start", "http://localhost:8080/about", wait)
}

// Verify the human time representation of Molly's start date
casper.Call("then", func() {
    mollysStartDate := time.Date(2017, 5, 24, 17, 9, 0, 0, time.UTC)
    mollysStartDateInHumanTime := humanize.Time(mollysStartDate)
    casper.Call("capture", "screenshots/timeago_cog_test.png")
    casper.Get("test").Call("assertSelectorHasText", "#Gopher-Molly
.timeagoSpan", mollysStartDateInHumanTime, "Verify human time of
Molly's start date produced by the Time Ago Cog.")
})

casper.Call("run", func() {
    casper.Get("test").Call("done")
})
}
```

Inside the main function, after we set the viewport size, and begin the test suite, we create a new `time` instance, called `mollysStartDate`, which represents the time when Molly joined the IGWEB team. We then pass `mollysStartDate`, to the `Time` function of the `go-humanize` package (note that we have aliased this package as "humanize") and store the human understandable value of the start date in the `mollysStartDateHumanTime` variable.

We generate a screenshot of the test run (shown in *Figure 10.21*). We then call the `tester` module object's `assertSelectorHasText` method, passing in the CSS selector to the `div` container containing Molly's start date in human readable format. We also pass in the `mollysStartDateInHumanTime` variable, since this is the expected text that should exist within the selector.

We will signify the end of the time ago cog test suite by calling the `done` method on the `tester` module object.

We can run the CasperJS test for the time ago cog test suite by issuing the following command:

```
$ casperjs test js/humantimecog_test.js
```

Figure 10.20 shows a screenshot of the result of running the time ago cog test suite:



```
balans0:tests kamesh$ casperjs test js/humantimecog_test.js
Test file: js/humantimecog_test.js
# Time Ago Cog Test Suite
PASS Verify human time of Molly's start date produced by the Time Ago Cog.
PASS Time Ago Cog Test Suite (1 test)
PASS 1 test executed in 1.089s, 1 passed, 0 failed, 0 dubious, 0 skipped.
balans0:tests kamesh$
```

Figure 10.20: Running the time ago cog test suite

Figure 10.21 shows the generated screenshot showing the **About** page, with Molly's start date printed out in human readable time format:

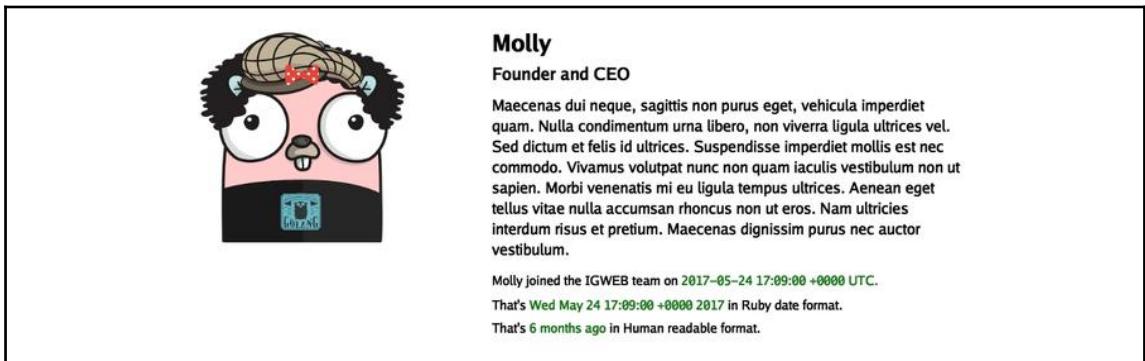


Figure 10.21: Test to verify the time ago cog

Now that we've verified the functionality for the time ago cog, let's look into testing the functionality of the live clock cog.

## Verifying the live clock cog

Verifying the functionality for the live clock cog for the user's local time, consists of creating a new `time` instance, of the current time formatted based on the local zone name and the local timezone offset, and comparing it to the value that is in the `myLiveClock` div container displayed on the home page.

Here is the test suite for the live clock cog, which is implemented in the `liveclockcog_test.go` source file found in the `client/tests/go` directory:

```
package main

import (
    "time"

    "github.com/EngineerKamesh/igb/igweb/client/tests/go/caspertest"
    "github.com/gopherjs/gopherjs/js"
)

var wait = js.MakeFunc(func(this *js.Object, arguments []*js.Object) interface{} {
    this.Call("waitForSelector", "#myLiveClock div")
    return nil
})

var casper = js.Global.Get("casper")

func main() {

    viewportParams := &caspertest.ViewportParams{Object:
        js.Global.Get("Object").New()}
    viewportParams.Width = 1440
    viewportParams.Height = 960
    casper.Get("options").Set("viewportSize", viewportParams)

    casper.Get("test").Call("begin", "Live Clock Cog Test Suite", 1,
        func(test *js.Object) {
            casper.Call("start", "http://localhost:8080/index", wait)
        })

    // Verify that the live clock shows the current time for the local
    // time zone
    casper.Call("then", func() {
        casper.Call("wait", 900, func() {

            localZonename, localOffset := time.Now().In(time.Local).Zone()
            const layout = time.RFC1123
            var location *time.Location
            location = time.FixedZone(localZonename, localOffset)
            casper.Call("wait", 10, func() {
                t := time.Now()
                currentTime := t.In(location).Format(layout)
                casper.Get("test").Call("assertSelectorHasText", "#myLiveClock
div", currentTime, "Display live clock for local timezone.")
            })
        })
    })
}
```

```

        })
    })
})

casper.Call("then", func() {
    casper.Call("capture", "screenshots/liveclock_cog_test.png")
})

casper.Call("run", func() {
    casper.Get("test").Call("done")
})

}

```

After setting the web browser's viewport size and starting the test suite by accessing the home page, we wait for 900ms, and then we gather the user's local time zone name and the local time zone offset. We will be formatting the time according to the RFC1123 layout. This happens to be the same layout that is used by the live clock cog for displaying the time.

We call the `FixedZone` function from the `time` package passing in `localZoneName` and `localOffset` to get the location. We create a new time zone instance, and format it using the `location` and the `RFC1123` layout. We use the `tester` module object's `assertSelectorHasText` method to see whether the current time formatted, using the `RFC1123` layout and using the user's present `location`, exists within the selector specified to the `assertSelectorHasText` method.

We generate a screenshot of the test run (shown in *Figure 10.23*), and then call the `done` method on the `tester` module object to signify the end of the test suite.

We can run the CasperJS test for the live clock cog test suite by issuing the following command:

```
$ casperjs test js/liveclockcog_test.js
```

*Figure 10.22* shows a screenshot of the result of running the live clock cog test suite:

```

balans0:tests kamesh$ casperjs test js/liveclockcog_test.js
test file: js/liveclockcog_test.js
# Live Clock Cog Test Suite
PASS Display live clock for local timezone.
PASS Live Clock Cog Test Suite (1 test)
PASS 1 test executed in 1.722s, 1 passed, 0 failed, 0 dubious, 0 skipped.
balans0:tests kamesh$ 
```

Figure 10.22: Running the live clock cog test suite

Figure 10.23 shows the generated screenshot showing the live clock cog on the **Home** page:

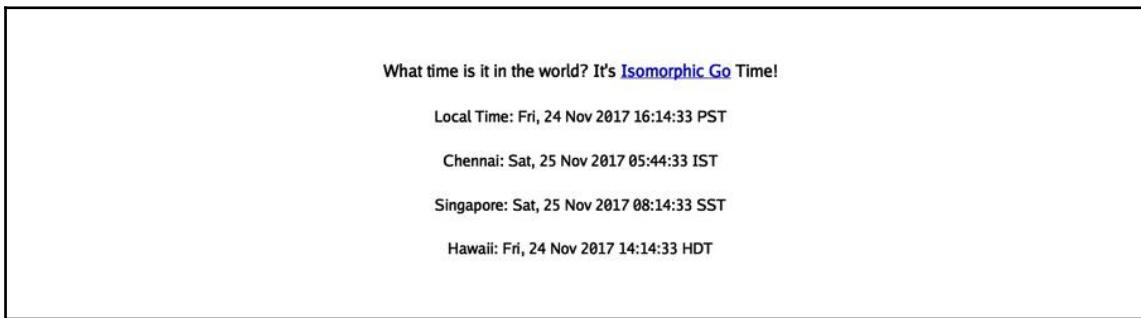


Figure 10.23: Testing the live clock cog on the home page

Now that we've verified the functionality for the live clock cog, let's look into testing the functionality of the date picker cog.

## Verifying the date picker cog

Verifying the functionality for the date picker cog consists of navigating to the **Contact** page, and clicking on the **Time Sensitivity Date** input field. This should trigger the display of the calendar widget.

Here is the test suite for the date picker cog, which is implemented in the `datepickercog_test.go` source file, located in the `client/tests/go` directory:

```
package main

import (
    "github.com/EngineerKamesh/igb/igweb/client/tests/go/caspertest"
    "github.com/gopherjs/gopherjs/js"
)

var wait = js.MakeFunc(func(this *js.Object, arguments []*js.Object) interface{} {
    this.Call("waitForSelector", "#primaryContent")
    return nil
})

var casper = js.Global.Get("casper")

func main() {
```

```
viewportParams := &caspertest.ViewportParams{Object:
js.Global.Get("Object").New()
viewportParams.Width = 1440
viewportParams.Height = 960
casper.Get("options").Set("viewportSize", viewportParams)

casper.Get("test").Call("begin", "Date Picker Cog Test Suite", 1,
func(test *js.Object) {
    casper.Call("start", "http://localhost:8080/contact", wait)
})

// Verify that the date picker is activated upon clicking the date
input field
casper.Call("then", func() {
    casper.Call("click", "#byDateInput")
    casper.Call("capture", "screenshots/datepicker_cog_test.png")
    casper.Get("test").Call("assertVisible", ".pika-single", "Display
    Datepicker Cog.")
})

casper.Call("run", func() {
    casper.Get("test").Call("done")
})
}
```

Inside the main function, we set the web browser's viewport size and start the test suite by navigating to the **Contact** page.

We then call the `casper` object's `click` method and provide the CSS selector `"#byDateInput"`, which will send a mouse click event to the **Time Sensitivity Date** input field, which should reveal the calendar widget.

We take a screenshot of the test run (shown in *Figure 10.25*), and then call the `tester` module object's `assertVisible` method, providing the `".pika-single"` selector and the name of the test, as input arguments to the method. The `assertVisible` method will assert that at least one element matching the provided selector expression is visible.

Finally, we call the `done` method on the `tester` module object to signify the end of the test suite.

We can run the CasperJS test for the date picker cog test suite by issuing the following command:

```
$ casperjs test js/datepickercog_test.js
```

Figure 10.24 shows a screenshot of the result of running the date picker cog test suite:

```
balans0:tests kamesh$ casperjs test js/datepickercog_test.js
Test file: js/datepickercog_test.js
# Date Picker Cog Test Suite
PASS Display Datepicker Cog.
PASS Date Picker Cog Test Suite (1 test)
PASS 1 test executed in 0.862s, 1 passed, 0 failed, 0 dubious, 0 skipped.
balans0:tests kamesh$
```

Figure 10.24: Running the date picker cog test suite

Figure 10.25 shows the generated screenshot showing the calendar widget after the **Time Sensitivity Date** input field has been clicked:

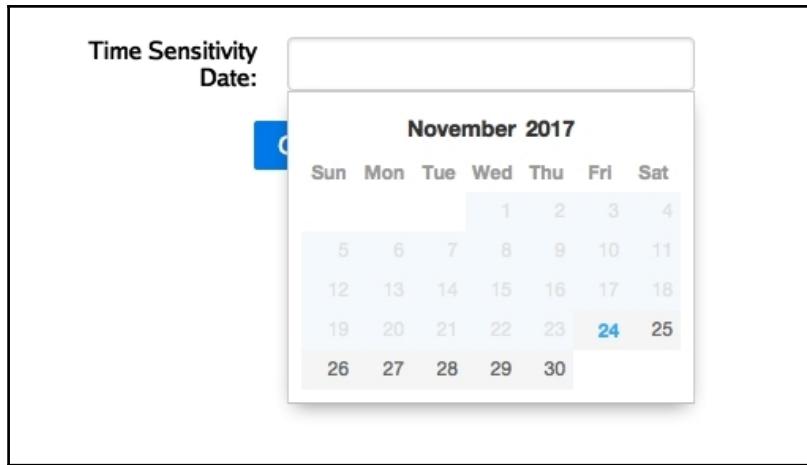


Figure 10.25: Test verifying that the date picker appears

Now that we've verified the functionality for the date picker cog, let's look into testing the functionality of the carousel cog.

## Verifying the carousel cog

Verifying the functionality for the carousel cog consists of providing sufficient time for the images of the carousel to load, and for the first image, the `watch.jpg` image file to appear on the web page.

Here is the test suite for the carousel cog, which is implemented in the `carouselcog_test.go` source file, located in the `client/tests/go` directory:

```
package main

import (
    "github.com/EngineerKamesh/igb/igweb/client/tests/go/caspertest"
    "github.com/gopherjs/gopherjs/js"
)

var wait = js.MakeFunc(func(this *js.Object, arguments []*js.Object) interface{} {
    this.Call("waitForSelector", "#carousel")
    return nil
})

var casper = js.Global.Get("casper")

func main() {

    viewportParams := &caspertest.ViewportParams{Object:
        js.Global.Get("Object").New()}
    viewportParams.Width = 1440
    viewportParams.Height = 960
    casper.Get("options").Set("viewportSize", viewportParams)

    casper.Get("test").Call("begin", "Carousel Cog Test Suite", 1,
        func(test *js.Object) {
            casper.Call("start", "http://localhost:8080/index", wait)
        })

    // Verify that the carousel cog has been loaded.
    casper.Call("wait", 1800, func() {
        casper.Get("test").Call("assertResourceExists", "watch.jpg",
            "Display carousel cog.")
    })

    casper.Call("then", func() {
        casper.Call("capture", "screenshots/carousel_cog_test.png")
    })

    casper.Call("run", func() {
        casper.Get("test").Call("done")
    })
}
```

After setting the web browser's viewport size and starting the test suite, by navigating to the **Home** page, we wait 1800 milliseconds and then we call the `assetResourceExists` method on the `tester` module object, supplying the name of the resource to check, which happens to be the "watch.jpg" image file, and a description of the test. The `assertResourceExists` function checks to see whether the "watch.jpg" image file exists in the set of assets that were loaded on the web page.

We take a screenshot of the test run (shown in *Figure 10.27*), and then call the `done` method on the `casper` object to signify the end of the test suite.

We can run the CasperJS test for the carousel cog test suite by issuing the following command:

```
$ casperjs test js/carouselcog_test.js
```

*Figure 10.26* shows a screenshot of the result of running the carousel cog test suite:

```
balans@tests kamesh$ casperjs test js/carouselcog_test.js
tests -- bash -- 144x45
Test file: js/carouselcog_test.js
# Carousel Cog Test Suite
PASS Display carousel cog.
PASS Carousel Cog Test Suite (1 test)
PASS 1 test executed in 2.559s, 1 passed, 0 failed, 0 dubious, 0 skipped.
balans@tests kamesh$
```

Figure 10.26: Running the carousel cog test suite

*Figure 10.27* shows the generated screenshot showing the carousel cog:



Figure 10.27: Test to verify that the carousel cog appears

Now that we've verified the functionality for the carousel cog, let's look into testing the functionality of the notify cog.

## Verifying the notify cog

Verifying the functionality for the notify cog consists of navigating to the products listing page, adding an item to the shopping cart by clicking the **Add to Cart** button on a listed product, and then verifying that the notification appeared on the web page.

Here is the test suite for the notify cog, which is implemented in the `notifycog_test.go` source file, located in the `client/test/go` directory:

```
package main

import (
    "github.com/EngineerKamesh/igb/igweb/client/tests/go/caspertest"
    "github.com/gopherjs/gopherjs/js"
)

var wait = js.MakeFunc(func(this *js.Object, arguments []*js.Object)
interface{} {
    this.Call("waitForSelector", "#primaryContent")
    return nil
})

var casper = js.Global.Get("casper")

func main() {

    viewportParams := &caspertest.ViewportParams{Object:
        js.Global.Get("Object").New()}
    viewportParams.Width = 1440
    viewportParams.Height = 960
    casper.Get("options").Set("viewportSize", viewportParams)

    casper.Get("test").Call("begin", "Notify Cog Test Suite", 1,
        func(test *js.Object) {
            casper.Call("start", "http://localhost:8080/products", wait)
        })

    // Add an item to the shopping cart
    casper.Call("then", func() {
        casper.Call("click", ".addToCartButton:nth-child(1)")
    })

    // Verify that the notification has been displayed
    casper.Call("wait", 450, func() {
        casper.Get("test").Call("assertSelectorHasText", "#alertify-logs
            .alertify-log-success", "Item added to cart", "Display Notify Cog
            when item added to shopping cart.")
    })
}
```

```
    })

    casper.Call("wait", 450, func() {
        casper.Call("capture", "screenshots/notify_cog_test.png")
    })

    // Navigate to Shopping Cart page
    casper.Call("then", func() {
        casper.Call("click", "a[href='/shopping-cart']")
    })

    // Remove product from shopping cart
    casper.Call("wait", 450, func() {
        casper.Call("click", ".removeFromCartButton:first-child")
    })

    casper.Call("run", func() {
        casper.Get("test").Call("done")
    })
}
```

After setting the web browser's viewport and starting the test suite by navigating to the products listing page, we call the `casper` object's `click` method, providing the `".addToCartButton:nth-child(1)"` selector. This sends a mouse click event to the first **Add to Cart** button on the web page.

We wait for 450 milliseconds and then call the `tester` module's `assertSelectorHasText` method providing the CSS selector, the text that should exist within the element returned from the selector, and the description of the test as input parameters.

We take a screenshot of the test run (shown in *Figure 10.29*). We then navigate to the shopping cart page, and remove the item from the shopping cart.

Finally, we call the `done` method on the `tester` module object to signify the end of the test suite.

We can run the CasperJS test for the notify cog test suite by issuing the following command:

```
$ casperjs test js/notifycog_test.js
```

Figure 10.28 shows a screenshot of the result of running the notify cog test suite:



```
balans0:tests kamesh$ casperjs test js/notifycog_test.js
Test file: js/notifycog_test.js
# Notify Cog Test Suite
PASS Display Notify Cog when item added to shopping cart.
PASS Notify Cog Test Suite (1 test)
PASS 1 test executed in 1.164s, 1 passed, 0 failed, 0 dubious, 0 skipped.
balans0:tests kamesh$
```

Figure 10.28: Running the notify cog test suite

Figure 10.29 shows the generated screenshot showing the notification message displayed on the lower right hand corner of the web page as expected:

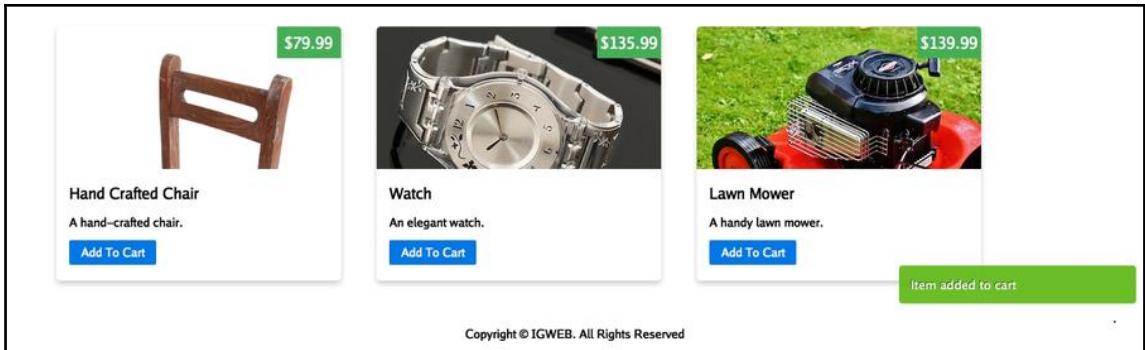
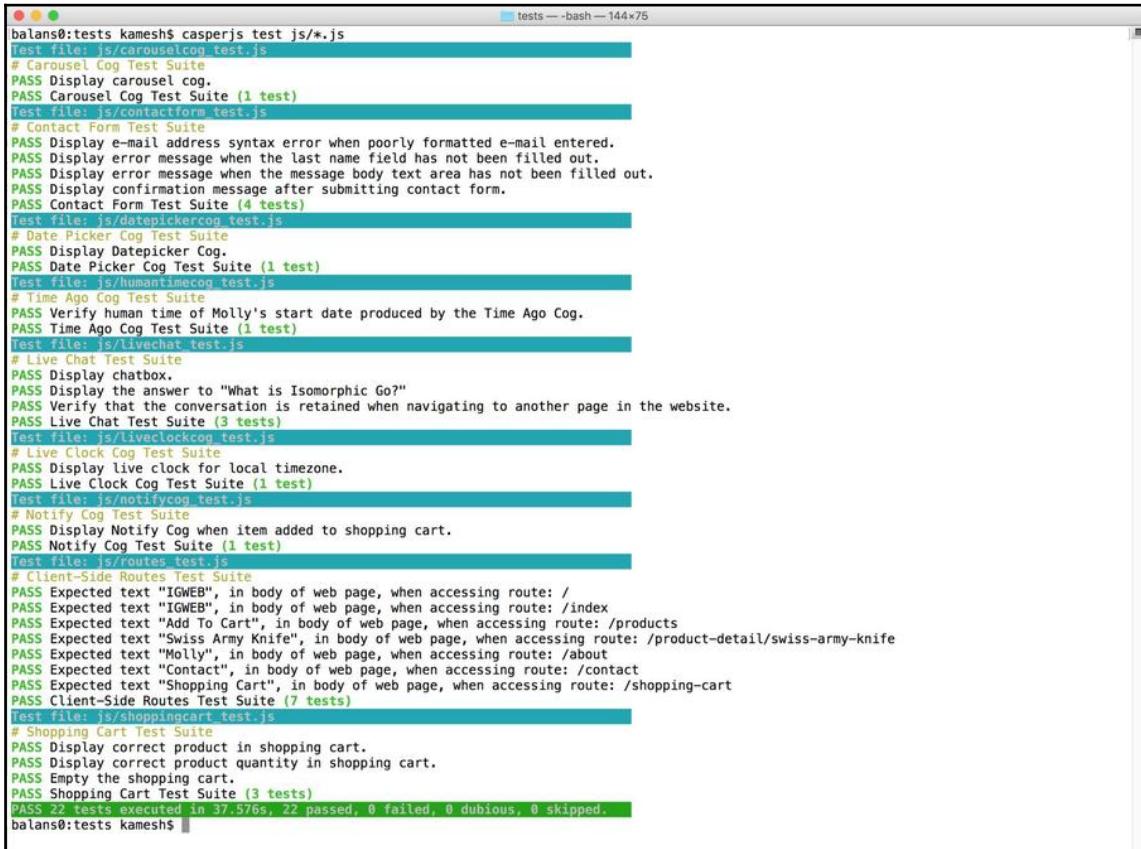


Figure 10.29: Running the test to verify that the notification message was displayed

We have now verified that the notify cog is functioning as expected and this wraps up our testing of IGWEB's client-side functionality.

Figure 10.30 shows a screenshot of running the whole collection of test suites by running the following command:

```
$ casperjs test js/*.js
```



```
balans0:tests kamesh$ casperjs test js/*.js
Test file: js/carouselcog test.js
# Carousel Cog Test Suite
PASS Display carousel cog.
PASS Carousel Cog Test Suite (1 test)

Test file: js/contactform test.js
# Contact Form Test Suite
PASS Display e-mail address syntax error when poorly formatted e-mail entered.
PASS Display error message when the last name field has not been filled out.
PASS Display error message when the message body text area has not been filled out.
PASS Display confirmation message after submitting contact form.
PASS Contact Form Test Suite (4 tests)

Test file: js/daterangepickercog test.js
# Date Picker Cog Test Suite
PASS Display Datepicker Cog.
PASS Date Picker Cog Test Suite (1 test)

Test file: js/humantimecog test.js
# Time Ago Cog Test Suite
PASS Verify human time of Molly's start date produced by the Time Ago Cog.
PASS Time Ago Cog Test Suite (1 test)

Test file: js/livechat test.js
# Live Chat Test Suite
PASS Display chatbox.
PASS Display the answer to "What is Isomorphic Go?"
PASS Verify that the conversation is retained when navigating to another page in the website.
PASS Live Chat Test Suite (3 tests)

Test file: js/livelockclockcog test.js
# Live Clock Cog Test Suite
PASS Display live clock for local timezone.
PASS Live Clock Cog Test Suite (1 test)

Test file: js/notifycog test.js
# Notify Cog Test Suite
PASS Display Notify Cog when item added to shopping cart.
PASS Notify Cog Test Suite (1 test)

Test file: js/routes test.js
# Client-Side Routes Test Suite
PASS Expected text "IGWEB", in body of web page, when accessing route: /
PASS Expected text "IGWEB", in body of web page, when accessing route: /index
PASS Expected text "Add To Cart", in body of web page, when accessing route: /products
PASS Expected text "Swiss Army Knife", in body of web page, when accessing route: /product-detail/swiss-army-knife
PASS Expected text "Molly", in body of web page, when accessing route: /about
PASS Expected text "Contact", in body of web page, when accessing route: /contact
PASS Expected text "Shopping Cart", in body of web page, when accessing route: /shopping-cart
PASS Client-Side Routes Test Suite (7 tests)

Test file: js/shoppingcart test.js
# Shopping Cart Test Suite
PASS Display correct product in shopping cart.
PASS Display correct product quantity in shopping cart.
PASS Empty the shopping cart.
PASS Shopping Cart Test Suite (3 tests)
PASS 22 tests executed in 37.576s, 22 passed, 0 failed, 0 dubious, 0 skipped.

balans0:tests kamesh$
```

Figure 10.30: Running the whole collection of CasperJS test suites

# Summary

In this chapter, you learned how to perform end-to-end testing to verify the functionality of an Isomorphic Go web application. To ensure the quality of IGWEB, prior to website launch, we started out by gathering the set of baseline functionality to test for.

To verify server-side functionality, we implemented tests using Go's testing package from the standard library. We implemented tests that verified server-side routing/template rendering, the contact form's validation functionality, and the successful contact form submission scenario.

To verify client-side functionality, we implemented tests using CasperJS that verified multiple user interaction scenarios. We were able to perform automated user interaction testing using CasperJS, since it sits on top of PhantomJS, a headless web browser equipped with a JavaScript runtime. We implemented CasperJS tests to verify the client-side routing/template rendering, the contact form's client-side validation functionality, the successful contact form submission scenario on the client-side, the functionality of the shopping cart, and the functionality of the live chat feature. We also implemented CasperJS tests, to verify the functionality of the collection of cogs that we implemented in [Chapter 9, Cogs – Reusable Components](#).

In [Chapter 11, Deploying an Isomorphic Go Web Application](#), you will learn how to deploy IGWEB to the cloud. We will first explore the procedure to release the website to a standalone server. After that, you'll learn how to utilize Docker, to release the website as a multi-container Docker application.

# 11

## Deploying an Isomorphic Go Web Application

With the automated end-to-end tests that we implemented in the last chapter, the IGWEB demo website now meets a baseline set of expected functionality. The time has come to free our Isomorphic Go web application out into the web. It's time to focus on deploying IGWEB to production.

Our exploration of Isomorphic Go production deployments will include deploying IGWEB as a static binary executable, along with static assets, to a standalone server (real or virtual) and deploying IGWEB as a multi-docker container application.

Deploying web applications is a vast subject, an ocean, worthy of the many books that are dedicated to covering this topic alone. Real-world web application deployments may include continuous integration, configuration management, automated testing, deployment automation tools, and agile team management. These deployments may also include multiple team members, fulfilling various roles in the deployment process.

The focus in this chapter will solely be on deploying an Isomorphic Go web application by a single individual. For the purpose of illustration, the deployment procedure will be performed manually.

There are certain considerations that need to be made to successfully prepare an Isomorphic Go web application for production use, such as minifying the GopherJS produced JavaScript source file and ensuring that static assets are transferred to the web client with GZIP compression enabled. By keeping the emphasis of the material presented in this chapter on Isomorphic Go, it is left to the reader to adapt the concepts and techniques presented in this chapter for their own particular deployment needs.

In this chapter, we will cover the following topics:

- How IGWEB operates in production mode
- Deploying an Isomorphic Go web application to a standalone server
- Deploying an Isomorphic Go web application using Docker

## How IGWEB operates in production mode

Before proceeding to production deployment, we need to understand how the server-side web application, `igweb`, operates when it's placed into production mode. Production mode can be turned on by setting the `IGWEB_MODE` environment variable with the value "production" before starting the `igweb` server-side application, like so:

```
$ export IWEB_MODE=production
```

There are three important behaviors that will take place when IGWEB runs in production mode:

1. The JavaScript external `<script>` tag that includes the client-side application, inside the header partial template, will request the minified JavaScript source file located at `$IGWEB_APP_ROOT/static/js/client.min.js`.
2. Static assets for cogs (`cogimport.css` and `cogimport.js`) will not be generated automatically when the web server instance starts. Instead, the minified source files containing the bundled static assets for the CSS and JavaScript will be located at `$IGWEB_APP_ROOT/static/css/cogimports.min.css` and `$IGWEB_APP_ROOT/static/js/cogimports.min.js`, respectively.
3. Rather than relying on the templates found in the `$IGWEB_APP_ROOT/shared/templates` folder, templates will be read from a single, gob encoded, template bundle file persisted on the disk.

We are going to consider how the server-side web application responds to each one of these behaviors.

## GopherJS-produced JavaScript source file

Inside the `funcs.go` source file where our template functions are defined, we introduce a new function called `IsProduction`:

```
func IsProduction() bool {
    if isokit.OperatingEnvironment() == isokit.ServerEnvironment {
        return os.Getenv("IGWEB_MODE") == "production"
    } else {
        return false
    }
}
```

This function, meant to be used on the server-side, will return a value of `true` if the current operating mode is production, and `false` if it isn't. We can use this custom function within a template to determine from where the client-side JavaScript application should be obtained.

When operating in non-production mode, the `client.js` source file will be obtained from the server relative path of `/js/client.js`. In production mode, the `minified` JavaScript source file will be obtained from the server relative path of `/static/js/client.min.js`.

Inside the header partial template, we call the `productionmode` custom function to determine from which path to serve the client-side JavaScript source file, like so:

```
<head>
    <meta name="viewport" content="initial-scale=1.0, maximum-scale=1.0,
user-scalable=no">
    <title>{{.PageTitle}}</title>
    <link rel="icon" type="image/png"
href="/static/images/isomorphic_go_icon.png">
    <link rel="stylesheet" href="/static/css/pure.min.css">
    {{if productionmode}}
        <link rel="stylesheet" type="text/css"
href="/static/css/cogimports.min.css">
        <link rel="stylesheet" type="text/css" href="/static/css/igweb.min.css">
        <script type="text/javascript" src="/static/js/client.min.js"
async></script>
        <script src="/static/js/cogimports.min.js" type="text/javascript"
async></script>
    {{else}}
        <link rel="stylesheet" type="text/css" href="/static/css/cogimports.css">
        <link rel="stylesheet" type="text/css" href="/static/css/igweb.css">
        <script src="/static/js/cogimports.js" type="text/javascript"
async></script>
        <script type="text/javascript" src="/js/client.js" async></script>
```

```
  {{end}}
</head>
```

You might be wondering why we include different JavaScript source files (`client.js` versus `client.min.js`) between non-production mode and production-mode. Recall that in a development environment with `kick` running, the `client.js` and `client.js.map` source files get produced in the `$IGWEB_APP_ROOT/client` folder. Inside `igweb.go`, we registered handlers functions for routes that will map the `/js/client.js` path and the `/js/client.js.map` path to the respective source files in the `$IGWEB_APP_ROOT/client` folder:

```
// Register Handlers for Client-Side JavaScript Application
if WebAppMode != "production" {
    r.Handle("/js/client.js",
    isokit.GopherjsScriptHandler(WebAppRoot)).Methods("GET")
    r.Handle("/js/client.js.map",
    isokit.GopherjsScriptMapHandler(WebAppRoot)).Methods("GET")
}
```

This provided us with the convenience that we could have `kick` automatically transpile the JavaScript code for us, the instant that we made a change to the application code. In non-production mode, we prefer not to minify the JavaScript source file so that we can get more detailed debug information through the web console, such as the panic stack trace (covered in Appendix, *Debugging Isomorphic Go*).

In production mode, there is no need to use `kick`. If you inspect the file size of the `client.js` source file, you will notice that it's approximately 8.1 MB big! That sure is some serious sticker shock! In the next section, we'll learn how to cut that unwieldy file footprint down to size.

## Taming the GopherJS-produced JavaScript file size

During the production deployment process, we must issue the `gopherjs build` command, specifying the option to minify the produced JavaScript source file and to save the output of the JavaScript source file to a specified target location.

We must minify the produced JavaScript code to reduce its file size. As mentioned previously, the un-minified, JavaScript source file is 8.1 MB! We can further reduce the size of the source file to 2.9 MB by minifying it, running the `gopherjs build` command with the `-m` option and by specifying the `--tags` option with the value `clientonly` like so:

```
$ gopherjs build -m --verbose --tags clientonly -o  
$IGWEB_APP_ROOT/static/js/client.min.js
```

The `clientonly` tag, tells isokit to avoid transpiling source files that are not used by the client-side application. The `-o` option will place the produced output JavaScript source file in the specified target location.



Prior to running the `gopherjs build` command, it's always a good idea to execute `clear_gopherjs_cache.sh` bash script found in the `$IGWEB_APP_ROOT/scripts` directory. It will clear project artifacts that have been cached from previous `gopherjs build` runs.

Serving a JavaScript source file that is nearly 3 MB large is still an untenable proposition for production needs. We can further cut down the size of transferring the file by enabling GZIP compression. Once the source file is sent using GZIP compression, the transfer file size will be approximately 510 KB. We will learn how to enable GZIP compression on the web server in the *Enabling GZIP compression* section.

## Generating static assets

When deploying server-side Go web applications, it is commonplace to not only push out the binary executable file for the web server instance but also static asset files (CSS, JavaScript, template files, images, fonts, and so on) and template files. In traditional Go web applications, we would have to push out the individual template files to the production system, since traditional Go web applications would be dependent on having each individual file available in order to render the given template on the server-side.

Since we are utilizing the concept of a template set persisted in-memory through the running application, there is no need to bring along the individual template files to the production environment. This is due to the fact that all we need to generate the in-memory template set is a single `gob` encoded template bundle file, which is persisted on disk in the `$IGWEB_APP_ROOT/static/templates` folder.

By setting the exported `StaticTemplateBundleFilePath` variable in the `isokit` package, we instruct `isokit` to generate the static template bundle file at the file path that we provide. Here's the line in the `initializeTemplateSet` function in the `igweb.go` source file where we set the variable:

```
isokit.StaticTemplateBundleFilePath = StaticAssetsPath +  
"/templates/igweb.tmplbundle"
```

In Chapter 9, *Cogs – Reusable Components*, we learned that `isokit` bundles all of the JavaScript source files from all of the cogs into a single `cogimports.js` source file when the `igweb` application is first started. In a similar manner, all of the CSS stylesheets from all of the cogs are bundled into a single `cogimports.css` source file. When running `IGWEB` in non-production mode, the static assets are bundled automatically by calling the `isokit.BundleStaticAssets` function (shown in bold) in the `initailizeCogs` function found in the `igweb.go` source file:

```
func initializeCogs(ts *isokit.TemplateSet) {  
    timeago.NewTimeAgo().CogInit(ts)  
    liveclock.NewLiveClock().CogInit(ts)  
    datepicker.NewDatePicker().CogInit(ts)  
    carousel.NewCarousel().CogInit(ts)  
    notify.NewNotify().CogInit(ts)  
    isokit.BundleStaticAssets()  
}
```

The automatic static assets bundling should not be utilized in a production environment, because the dynamic functionality that bundles the JavaScript and CSS is dependent on the server having an installed Go distribution with a configured Go workspace, and access to the source files for the cogs must be present in that Go workspace.

This immediately removes one of the advantages that Go comes with out of the box. Since Go produces statically linked binary executable files, we don't need to have a Go runtime installed on our production server in order to deploy our application.

When we run `IGWEB` in production mode, we can prevent the automatic static assets bundling by introducing the following code in the `initializeTemplateSet` function found in the `igweb.go` source file:

```
if WebAppMode == "production" && oneTimeStaticAssetsGeneration == false {  
    isokit.UseStaticTemplateBundleFile = true  
    isokit.ShouldBundleStaticAssets = false  
}
```

We instruct isokit to use a static template bundle file, and we instruct isokit not to automatically bundle static assets.

In order to generate the static assets (CSS, JavaScript, and the template bundle) that our Isomorphic Go web application requires, we can run `igweb` with the `--generate-static-assets` flag on a non-production system:

```
$ igweb --generate-static-assets
```

This command will produce the necessary static assets, and then it will exit the `igweb` program. The implementation for this functionality can be found in the `generateStaticAssetsAndExit` function defined in the `igweb.go` source file:

```
func generateStaticAssetsAndExit(env *common.Env) {
    fmt.Println("Generating static assets...")
    isokit.ShouldMinifyStaticAssets = true
    isokit.ShouldBundleStaticAssets = true
    initializeTemplateSet(env, true)
    initializeCogs(env.TemplateSet)
    fmt.Println("Done")
    os.Exit(0)
}
```

Three files will be created upon instructing `igweb` to generate the static assets:

- `$IGWEB_APP_ROOT/static/templates/igweb tmplbundle` (template bundle)
- `$IGWEB_APP_ROOT/static/css/cogimports.min.css` (minified CSS bundle)
- `$IGWEB_APP_ROOT/static/js/cogimports.min.js` (minified JavaScript bundle)

Upon performing a production deployment, the entire `$IGWEB_APP_ROOT/static` folder can be copied over to the production system, ensuring that the three aforementioned static assets will be made available on the production system.

At this point, we have established how `IGWEB` will operate in production mode. Now, it's time to perform the most simplest of deployments—deploying an Isomorphic Go web application to a standalone server.

# Deploying an Isomorphic Go web application to a standalone server

For demonstrating a standalone Isomorphic Go deployment, we will be using a virtual private server (VPS) hosted on Linode (<http://www.linode.com>). The procedure presented herein holds good for any other cloud provider, as well as the scenario where the standalone server happens to be a real server residing in your server room. The standalone deployment procedure that we will outline is performed manually to illustrate each step of the process.

## Provisioning the server

The server in this demonstration, and the servers mentioned in subsequent demonstrations in this chapter will be running Ubuntu Linux version 16.04 LTS on Linode, a provider of **virtual private server (VPS)** instances. We will be running Linode's default stock image of Ubuntu 16.04 without making any kernel modifications.



When we issue any commands in this chapter prefaced with `sudo`, we assume that your user account is part of the sudoers group. If you are using the server's root account, you need not preface the command with `sudo`.

We will create a less-privileged user called `igweb` by issuing the following command:

```
$ sudo adduser igweb
```

After running the `adduser` command, you will be prompted to enter additional information for the `igweb` user and the password. If you are not prompted to enter the password for the user, you may set a password by issuing the following command:

```
$ sudo passwd igweb
```

The `igweb` application depends on two components to function properly. First, we need to install the Redis database. Second, we need to install `nginx`. We will be using `nginx` as a reverse proxy server, which will allow us to enable GZIP compression when serving static assets to the web client. As you will see, this makes a huge difference when it comes to the file size of the GopherJS-produced JavaScript source file (510 KB versus 3MB). *Figure 11.1* depicts the Linode VPS instance with the three key components, `igweb`, `nginx`, and `redis-server`:

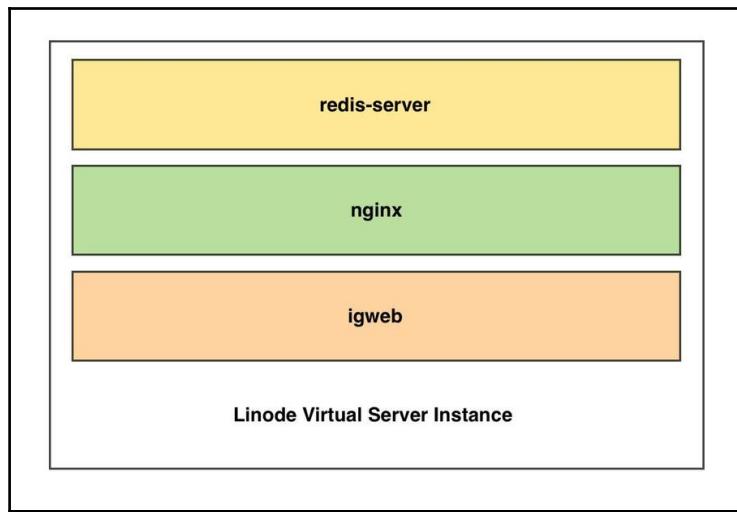


Figure 11.1: The Linode VPS instance running igweb, nginx, and redis-server

## Setting up the Redis database instance

You can follow the same procedure that was demonstrated in [Chapter 2, The Isomorphic Go Toolchain](#), to install the Redis database. Before doing so, you should issue the following command to install the essential build tools:

```
$ sudo apt-get install build-essential tcl
```

Once you have installed the Redis database, you should launch the Redis server by issuing the following command:

```
$ sudo redis-server --daemonize yes
```

The `--daemonize` command-line argument allows us to run the Redis server in the background. The server will continue to run even after our session has ended.



You should secure the Redis installation by adding sufficient firewall rules to prevent external traffic from accessing port 6379, the default port of the Redis server instance.

## Setting up the NGINX reverse proxy

Although the `igweb` web server instance, a Go application, can single-handedly fulfill the major needs to serve IGWEB, it is more advantageous to have the `igweb`, web server instance, sit behind a reverse proxy.

A reverse proxy server is a type of proxy server that will service a client request by dispatching the request to a designated destination server (in this case, `igweb`), get the response from the `igweb` server instance, and send the response back to the client.

Reverse proxies come in handy for several reasons. The most important reason for the immediate benefit of releasing IGWEB is that we can enable GZIP compression on the outbound static assets. In addition to that, reverse proxies also allow us to easily add redirect rules to control the flow of traffic as the need arises.

NGINX is a popular high performance web server. We will be using `nginx` as the reverse proxy that sits in front of the `igweb` web server instance. *Figure 11.2* depicts a typical reverse proxy configuration, where a web client will issue a HTTP request over port 80 and `nginx` will service the request by sending the HTTP request to the `igweb` server instance over port 8080, retrieving the response from the `igweb` server and sending the response back to the web client over port 80:

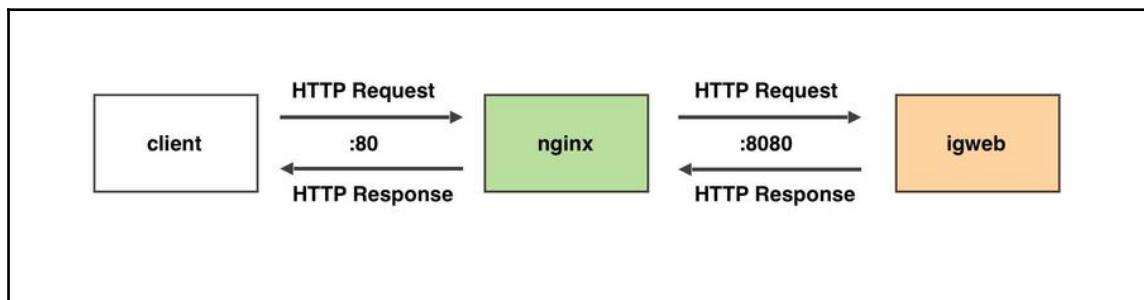


Figure 11.2: The reverse proxy configuration

Here is a listing of the `nginx` configuration file, `nginx.conf`, that we will use to run `nginx` as a reverse proxy:

```
user igweb;
worker_processes 1;

error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;
```

```
events {
    worker_connections 1024;
}

http {
    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for"';

    sendfile on;
    keepalive_timeout 65;

    gzip on;
    gzip_min_length 1100;
    gzip_buffers 16 8k;
    gzip_types text/plain application/javascript text/css;
    gzip_vary on;
    gzip_comp_level 9;

    server_tokens off;

    server {
        listen 80;
        access_log /var/log/nginx/access.log main;
        location / {
            proxy_pass http://192.168.1.207:8080/;
            proxy_set_header X-Forwarded-For $remote_addr;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
            proxy_set_header Host $host;
        }
    }
}
```

There are two sections of settings that are of particular interest to us, the section to enable GZIP compression and the section for proxy settings.

## Enabling GZIP compression

Let's examine the `nginx` configuration settings related to enable GZIP compression.

We set the `gzip` directive to `on` to enable the gzipping of server responses.

The `gzip_min_length` directive allows us to specify the minimum length of a response that will be gzipped.

The `gzip_buffers` directive sets the number and size of buffers that are used to compress the response. We have specified that we will be using 16 buffers with a memory page size of 8K.

The `gzip_types` directive allows us to specify the MIME types that we should enable GZIP compression on in addition to `text/HTML`. We have specified the MIME types for plain text files, JavaScript source files, and CSS source files.

The `gzip_vary` directive is used to either enable or disable the `Vary: Accept-Encoding` response header. The `Vary: Accept-Encoding` response header instructs the cache to store a different version of the web page if there is a variation in the header. This setting is particularly important for web browsers that do not support GZIP encoding to receive the uncompressed version of the file properly.

The `gzip_comp_level` directive specifies the level of GZIP compression that will be used. We have specified a value of 9, which is the maximum level of GZIP compression.

## Proxy settings

The second section in the `nginx` configuration settings that is important is the reverse proxy settings.

We include the `proxy_pass` directive inside the `location` block with the value of the address and port of the web server. This specifies that all requests should be sent to the specified proxy server (`igweb`) located at `http://192.168.1.207:8080`.

Remember to replace the IP address 192.168.1.207 shown in this example with the IP address of the machine that is running your `igweb` instance.



The reverse proxy will fetch the response from the `igweb` server instance and send it back to the web client.

The `proxy_set_header` directive allows us to redefine (or append) fields to the request header that are passed on to the proxy server. We have included the `X-Forwarded-For` header so that the proxy server can identify the originating IP address of the web client that initiated the request.

To support the proper functioning of websockets (which the live chat feature depends on), we include the following proxy settings. First, we specify using the `proxy_http_version` directive that the server will be using HTTP version 1.1. The "Upgrade" and "Connection" headers are not passed to a proxied server by default. Due to this, we must send these headers to the proxy server using the `proxy_set_header` directive.

We can install `nginx` by issuing the following command:

```
$ sudo apt-get install nginx
```

Upon installing `nginx`, the web server usually starts up by default. However if it doesn't, we can start up `nginx` by issuing the following command:

```
$ sudo systemctl start nginx
```

The `nginx.conf` file found in the `$IGWEB_APP_ROOT/deployments-  
config/standalone-setup` folder can be placed in the production server's `/etc/nginx` folder.

*Figure 11.3* depicts the **502 Bad Gateway** error encountered when we attempt to access the `igweb.kamesh.com` URL:



Figure 11.3: The 502 Bad Gateway Error

We get this server error because we haven't started `igweb` yet. To get `igweb` up and running, we first need to set up a place on the server where the `igweb` binary executable and the static assets will reside.

## Setting up the IGWEB root folder

The IGWEB root folder is where the `igweb` executable and the static assets will reside on the production server. We use the following command to become the `igweb` user on the production server:

```
$ su - igweb
```

We create an `igweb` folder in the home directory of the `igweb` user like so:

```
mkdir ~/igweb
```

This is the directory that will contain the binary executable file for the `igweb` web server instance and the static assets that are required by the IGWEB demo website. Take note that the static assets will reside in the `~/igweb/static` folder.

## Cross compiling IGWEB

Using the `go build` command, we can actually build binaries for different target operating systems, a technique known as **cross compiling**. For example, on my macOS machine, I can build a 64-bit Linux binary that we can push out to the standalone production server running Ubuntu Linux. Prior to building our binary, we specify the target operating system we want to build to by setting the `GOOS` environment variable:

```
$ export GOOS=linux
```

By setting the `GOOS` environment variable to `linux`, we have specified that we wish to generate a binary file for Linux.

In order to specify that we want the binary to be a 64-bit binary, we set the `GOARCH` environment variable to specify the target architecture:

```
$ export GOARCH=amd64
```

By setting the `GOARCH` variable to `amd64`, we have specified that we want a 64-bit binary.

Let's create a `builds` directory within our `igweb` folder by issuing the `mkdir` command:

```
$ mkdir $IGWEB/builds
```

This directory will serve as the depot containing `igweb` binary executables for various operating systems. For the purpose of this chapter, we'll only consider building a 64-bit Linux binary, but in the future we can accommodate builds for other operating systems, such as Windows, in this directory.

We issue the `go build` command and provide the `-o` argument to specify where the produced binary file should reside:

```
$ go build -o $IGWEB_APP_ROOT/builds/igweb-linux64
```

We have instructed that the produced 64-bit Linux binary should be created in the `$IGWEB_APP_ROOT/builds` folder and that the name of the executable will be `igweb-linux64`.

You can verify that the produced binary is a Linux binary by issuing the `file` command:

```
$ file builds/igweb-linux64
builds/igweb-linux64: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
  statically linked, not stripped
```

From the result, we can see that the `go build` command has produced a 64-bit LSB (Linux Standard Base) executable.



If you are interested in building Go binaries for other operating systems besides Linux, this link will provide you with a full list of all possible `GOOS` and `GOARCH` values: <https://golang.org/doc/install/source#environment>.

## Preparing the deployment bundle

Besides shipping out the `igweb` executable file, we also need to ship the contents of the `static` folder, that holds all of IGWEB's static assets.

Preparing the static assets for the deployment bundle consists of the following steps:

1. Transpiling the client-side application
2. Generating the static assets bundles (template bundle, CSS, and JavaScript)
3. Minifying the IGWEB CSS stylesheet

First, we transpile the client-side application:

```
$ cd $IGWEB_APP_ROOT/client
$ $IGWEB_APP_ROOT/scripts/clear_gopherjs_cache.sh
$ gopherjs build --verbose -m --tags clientonly -o
$IGWEB_APP_ROOT/static/js/client.min.js
```

Second, we have to generate the static assets bundle:

```
$ $IGWEB_APP_ROOT/igweb --generate-static-assets
Generating static assets...Done
```

The third, and final step, of preparing the deployment bundle consists of minifying the CSS stylesheet.

First, we need to install the Go-based minifier by issuing the following commands:

```
$ go get -u github.com/tdewolff/minify/cmd/minify
$ go install github.com/tdewolff/minify
```

Now, we can minify IGWEB's CSS stylesheet:

```
$ minify --mime="text/css" $IGWEB_APP_ROOT/static/css/igweb.css >
$IGWEB_APP_ROOT/static/css/igweb.min.css
```

With these items in place, we are now ready to create a deployment bundle, a tarball, which includes the `igweb` Linux binary along with the `static` folder. We create the tarball by issuing the following commands:

```
$ cd $IGWEB_APP_ROOT
$ tar zcvf /tmp/bundle.tgz builds/igweb-linux64 static
```

We'll use the `scp` command to ship the bundle off to the remote server:

```
$ scp /tmp/bundle.tgz igweb@targetserver:/tmp/.
```

The `scp` command will copy the tarball, `bundle.tgz`, to the `/tmp` directory on the server having the hostname `targetserver`. With the deployment bundle now placed on the server, it's time to get `igweb` up and running.

## Deploying the bundle and starting IGWEB

We move the template bundle that we had secure copied to the `/tmp` folder into the `~/igweb` folder and extract the contents of the tarball:

```
$ cd ~/igweb
$ mv /tmp/bundle.tgz .
$ tar zxvf bundle.tgz
```

After we are done extracting the contents of the `bundle.tgz` tarball, we remove the tarball file by issuing the `rm` command.

```
$ rm bundle.tgz
```

We can rename the binary file back to `igweb` using the `mv` command:

```
$ mv igweb-linux64 igweb
```

We had tacked on the `-linux64` to the name of the binary file in our local machine so that we could distinguish it from builds for other operating system/architecture combinations.

At this point we have deployed the bundle to the production server. It's now time to run `igweb`.

## Running IGWEB

Prior to running the `igweb` executable, we must set the `$IGWEB_APP_ROOT` and `$IGWEB_MODE` environment variables on the production server:

```
$ export IGWEB_APP_ROOT=/home/igweb/igweb
$ export IGWEB_MODE=production
```

Setting the `$IGWEB_APP_ROOT` environment variable allows the `igweb` application to know the designated `igweb` directory that will contain dependent resources, such as static assets.

Setting the `$IGWEB_MODE` environment variable to `production` allows us to run the `igweb` application in production mode.

You should add entries for these two environment variables in the `igweb` user's `.bashrc` configuration file:

```
export IGWEB_APP_ROOT=/home/igweb/igweb
export IGWEB_MODE=production
```



You can log out and log back in on the production server for the changes made to the `.bashrc` to take effect.

## Running IGWEB in the foreground

Let's start up the `igweb` web server instance:

```
$ cd $IGWEB_APP_ROOT
$ ./igweb
```

Figure 11.4 shows a screenshot of IGWEB running on a standalone server instance at the address <http://igweb.kamesh.com>:

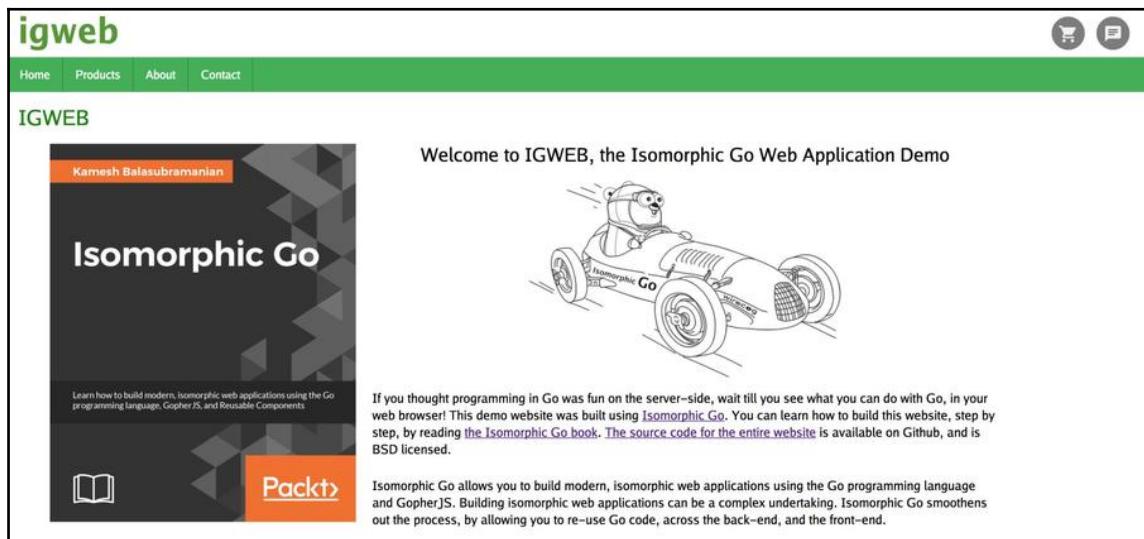


Figure 11.4: IGWEB running on a standalone server instance

When we hit the `Ctrl + C` key combination to exit the `igweb` program, our web server instance comes to a grinding halt since it's been running in the foreground. NGINX will return the **502 Bad Gateway** server error for any client request. We need a way to daemonize `igweb` so that it runs in the background.

## Running IGWEB in the background

The `igweb` web server instance can run in the background using the `nohup` command:

```
$ nohup ./igweb 2>&1 &
```

The `nohup` command is used to continue running the `igweb` program even after the current session has been terminated. On a Unix-like system, the `2>&1` construct means redirecting standard error (`stderr`) to the same place as the standard output (`stdout`). Log messages from the `igweb` program will be available for view by tailing the `/var/log/syslog` file. Finally, the last ampersand, `&`, in the command indicates running the program in the background.

We can stop the `igweb` process by first getting the **PID** (process ID):

```
$ ps -ef | grep igweb | grep -v grep
```

In the output returned from running this command, the PID value will be right next to the name of the executable, `igweb`. Once we determine the PID of the process, we can stop the `igweb` process by killing it using the `kill` command and specifying the PID's value:

```
$ kill PID
```



Note that we have placed the name `PID` in the preceding `kill` command for illustration purposes only. You will have to provide the `kill` command with the numeric value of the PID returned from running the `ps` command.

## Running IGWEB with systemd

This approach to running `igweb` works for the time being, but what if the server is rebooted? We need a means for the `igweb` program to be more resilient. It has to be able to start up again once the server comes back online, and `nohup` is not the suitable choice to accomplish this goal.

What we really need is a way to turn `igweb` into a system service. We can do exactly that with `systemd`, an init system, that is available with Ubuntu 16.04 LTS. With `systemd`, we can initialize, manage, and track system services. It can be used while the system starts up or while it is running.



You will need to run the following commands as the `root` user, since you need to be `root` in order to add a new system service.

In order to turn `igweb` into a service, we create a unit file called `igweb.service` and place it in the `/etc/systemd/system` directory. Here's the contents of the unit file:

```
[Unit]
Description=IGWEB

[Service]
USER=igweb
GROUP=igweb
Environment=IGWEB_APP_ROOT=/home/igweb/igweb
Environment=IGWEB_MODE=production
WorkingDirectory=/home/igweb/igweb
ExecStart=/home/igweb/igweb/igweb
Restart=always

[Install]
WantedBy=multi-user.target
```

Specifying the file extension of `.service` indicates that we are creating a service unit that describes how to manage an application on the server. This includes performing actions such as starting or stopping the service, and if the service should be started on system startup.

The unit file is organized into multiple sections, where the start of each section is denoted with a pair of square brackets `[` and `]` with the name of the section enclosed between the brackets.



Section names in unit files are case sensitive!

The first section is the `[Unit]` section. This is used to define the metadata for the unit and how this unit relates to other units. Inside the `[Unit]` section we have specified a value for the `Description`, which is used to describe the name of the unit. For example, we run the following command:

```
$ systemctl status nginx
```

When we run it, the description we see for `nginx` is the description that was specified using the `Description` directive.

The `[Service]` section is used to specify the configuration of the service. The `USER` and `GROUP` directive specify what user and group the command should run as. We use the `Environment` directive to set the `$IGWEB_APP_ROOT` environment variable, and we use it again to set the `$IGWEB_MODE` environment variable.

The `WorkingDirectory` directive sets the working directory for the executed command. The `ExecStart` directive specifies the full path to the command that is to be executed; in this case, we have provided the full path to the `igweb` executable file.

The `Restart` directive is used to specify the circumstances that `systemd` will attempt to restart the service. By providing a value of *always*, we have specified that the service should always be running, and if for some reason it stops, it should be started up again.

The last section we've defined is the `[Install]` section. This section allows us to specify the behavior of a unit when it is enabled or disabled.

The `WantedBy` directive that is declared in this section tells `systemd` how a unit should be enabled, that is, at what system runlevel should the service run in when it is enabled. By setting the value for this directive to `multi-user.target`, we specify that this service has a system runlevel of 3 (multi-user mode).

Anytime we introduce a new `systemd` service script or make changes to an existing one, we must reload the `systemd` daemon. We can do so by issuing the following command:

```
$ systemctl daemon-reload
```

We can specify, that we want the `igweb` service to startup automatically on boot by issuing the following command:

```
$ systemctl enable igweb
```

If we don't want the `igweb` service to startup automatically on boot, we can issue the following command:

```
$ systemctl disable igweb
```

We can start the `igweb` service by issuing the following command:

```
$ systemctl start igweb
```

We can stop the `igweb` service by issuing the following command:

```
$ systemctl stop igweb
```

We have now completed the standalone deployment of `igweb`. It's amazing that we can run the `igweb` application without having to install Go on the target production system.

However, this approach is rather opaque to the DevOps crew that is tasked to keep IGWEB up and running. What I mean by *opaque* is that there's not much a DevOps engineer can ascertain by examining a static binary executable file and a bunch of static assets.

What we need is a more streamlined way to deploy IGWEB, a procedure that shows us all of the dependencies needed to launch an `igweb` instance from scratch. To achieve this goal, we need to dockerize IGWEB.

## Deploying an Isomorphic Go web application using Docker

This section provides an overview of deploying `igweb` as a multi-container Docker application on the Linode cloud. Docker is a technology and a platform that allows us to run and manage multiple Docker containers on a single machine. You can think of a Docker container as a modular, lightweight virtual machine. We can make an application, such as `igweb`, instantly portable by packaging it as a Docker container. The application is guaranteed to run the same way inside the container, regardless of which environment it is run on.



You can learn more about Docker at the following link: <https://www.docker.com>.

Most cloud providers offer support for Docker making it a very handy tool for cloud-based deployments. As you will see later in this chapter, deploying a multi-container Docker application on the Linode cloud is relatively easy.

## Installing Docker

Before installing Docker on the production system, we first need to install some prerequisites:

```
$ sudo apt-get install dmsetup && dmsetup mknodes
```

Now, we can issue the following command to install Docker:

```
$ sudo apt-get install docker-ce
```

To verify that Docker has been installed properly on the production system, you may issue the following command:

```
$ docker --version
Docker version 17.09.0-ce, build afdb6d4
```

You should see the version of Docker installed after running the command.

## Dockerizing IGWEB

The process to dockerize `igweb` first involves creating a `Dockerfile`, a file that specifies instructions on how to create a Docker image. The Docker image will then be used to create a Docker container.

After having created the `Dockerfile`, we will be using the `docker-compose` tool to define and run multiple containers needed to power the IGWEB website.

Deploying `igweb` as a multi-container Docker application is a three-step process:

1. Create a `Dockerfile` from which an IGWEB docker image can be created
2. Define the services that make up IGWEB in a `docker-compose.yml` file
3. Run `docker-compose up` to start up the multi-container application

## The Dockerfile

The `Dockerfile` describes what an `igweb` docker image should be made of. The file is located in the `deployments-config/docker-single-setup` folder. Let's examine the `Dockerfile` to examine how it works.

The `FROM` instruction specifies the base parent image from which the current image is derived:

```
FROM golang
```

Here, we have specified that we will be using the base `golang` docker image.



More information about the `golang` docker image can be found at [https://hub.docker.com/\\_/golang/](https://hub.docker.com/_/golang/).

The `MAINTAINER` instruction specifies the name of the maintainer of the `Dockerfile` along with their email address:

```
MAINTAINER Kamesh Balasubramanian kamesh@kamesh.com
```

We have specified a group of `ENV` instructions which allow us to define and set all the required environment variables:

```
ENV IGWEB_APP_ROOT=/go/src/github.com/EngineerKamesh/igb/igweb
ENV IGWEB_DB_CONNECTION_STRING="database: 6379"
ENV IGWEB_MODE=production
ENV GOPATH=/go
```

For the proper operation of the `igweb` application, we set the `$IGWEB_APP_ROOT`, the `$IGWEB_DB_CONNECTION`, `$IGWEB_MODE`, and the `$GOPATH` environment variables.

In this block, we use `RUN` instructions to get the Go packages that are required by the `igweb` application:

```
RUN go get -u github.com/gopherjs/gopherjs
RUN go get -u honnef.co/go/js/dom
RUN go get -u -d -tags=js github.com/gopherjs/jsbuiltin
RUN go get -u honnef.co/go/js/xhr
RUN go get -u github.com/gopherjs/websocket
RUN go get -u github.com/tdewolff/minify/cmd/minify
RUN go get -u github.com/isomorphicgo/isokit
RUN go get -u github.com/uxtoolkit/cog
RUN go get -u github.com/EngineerKamesh/igb
```

This is basically the list of Go packages that are needed to get `igweb` up and running.

The following `RUN` command installs a Go-based CSS/JavaScript minifier:

```
RUN go install github.com/tdewolff/minify
```

We use another `RUN` instruction to transpile the client-side Go program:

```
RUN cd $IGWEB_APP_ROOT/client; go get ./...; /go/bin/gopherjs build -m --verbose --tags clientonly -o $IGWEB_APP_ROOT/static/js/client.min.js
```

This command is actually a combination of three sequential commands, where each command is separated using a semicolon.

The first command changes directory into the `$IGWEB_APP_ROOT/client` directory. In the second command, we fetch any remaining required Go packages in the current directory and all sub-directories. The third command transpiles the Go code to a minified JavaScript source file, `client.min.js`, which is placed in the `$IGWEB_APP_ROOT/static/js` directory.

The next `RUN` instruction builds and installs the server-side Go program:

```
>RUN go install github.com/EngineerKamesh/igb/igweb
```

Take note that the `go install` command will not only produce the `igweb` binary executable file by performing a build operation, but it will also move the produced executable file to `$GOPATH/bin`.

We issue the following `RUN` instruction to generate the static assets:

```
RUN /go/bin/igweb --generate-static-assets
```

This `RUN` instruction minifies IGWEB's CSS stylesheet:

```
RUN /go/bin/minify --mime="text/css" $IGWEB_APP_ROOT/static/css/igweb.css > $IGWEB_APP_ROOT/static/css/igweb.min.css
```

The `ENTRYPOINT` instruction allows us to set the main command of the container:

```
# Specify the entrypoint
ENTRYPOINT /go/bin/igweb
```

This provides us the ability to run the image as if it were a command. We have set the `ENTRYPOINT` to the path of the `igweb` executable file: `/go/bin/igweb`.

We use the `EXPOSE` instruction to inform Docker the network port the container should listen in on at runtime:

```
EXPOSE 8080
```

We have exposed port 8080 of the container.

Besides being able to build a docker image using a `Dockerfile`, one of the most important benefits of this file is that it conveys meaning and intent. It can be treated as a first-class project configuration artifact to understand exactly what goes into building the isomorphic web application that is comprised of the server-side `igweb` application and the client-side application `client.min.js`. From looking over the `Dockerfile`, a DevOps engineer can readily ascertain the procedure to successfully build the entire isomorphic web application from scratch.

## The Dockerfile for a closed source project

The `Dockerfile` that we presented works great for an open source project, but what would you do if your particular Isomorphic Go project is closed source? How could you still take advantage of running Docker in the cloud and keep your source code secure from view at the same time? We would need to make slight modifications to the `Dockerfile` to account for a closed-source project.

Let's consider a scenario where the `igweb` code distribution is closed source. Let's presume that we could not obtain it using the `go get` command.

Let's also assume that you have created a tarball bundle of the closed source `igweb` project, including a closed-source friendly `Dockerfile` at the root of the project directory. You have securely copied the tarball from your local machine to the target machine, and you have extracted the tarball.

Here are the changes that we would need to make to the `Dockerfile`. First, we comment out the respective `RUN` instruction that gets the `igb` distribution using the `go get` command:

```
# Get the required Go packages
RUN go get -u github.com/gopherjs/gopherjs
RUN go get -u honnef.co/go/js/dom
RUN go get -u -d -tags=js github.com/gopherjs/jsbuiltin
RUN go get -u honnef.co/go/js/xhr
RUN go get -u github.com/gopherjs/websocket
RUN go get -u github.com/tdewolff/minify/cmd/minify
```

```
RUN go get -u github.com/isomorphicgo/isokit
RUN go get -u github.com/uxtoolkit/cog
# RUN go get -u github.com/EngineerKamesh/igb
```

Right after the set of RUN instructions, we immediately introduce a COPY instruction:

```
COPY . $IGWEB_APP_ROOT/.
```

This COPY instruction will recursively copy all files and folders within the current directory to the destination specified by \$IGWEB\_APP\_ROOT/.. That's all there is to it.

Now that we have taken an in-depth look at the anatomy of IGWEB's Dockerfile, we have to acknowledge the fact that the igweb web server instance cannot serve the IGWEB website by itself. It has certain service dependencies that we must account for, such as the Redis database for its data persistence needs and the NGINX reverse proxy for serving hefty static assets in a sensible gzipped manner.

What we need is a Docker container for Redis and another Docker container for NGINX. igweb is turning out to be a multi-container Docker application. It's time to turn our focus to docker-compose, the handy tool for defining and running multi-container applications.

## Docker compose

The docker-compose tool allows us to define a multi-container Docker application and run it using a single command, docker-compose up.

docker-compose works by reading a docker-compose.yml file that contains specific instructions that not only describe the containers in the application, but also their individual dependencies. Let's examine each section of the docker-compose.yml file for the multi-container igweb application.

In the first line of the file, we indicate that we will be using version 2 of the Docker Compose configuration file format:

```
version: '2'
```

We declare the application's services inside the services section. Each service (shown in bold) is given a name to indicate its role in the multi-container application:

```
services:
  database:
    image: "redis"
  webapp:
```

```
depends_on:
  - database
build: .
ports:
  - "8080:8080"
reverseproxy:
  depends_on:
    - webapp
  image: "nginx"
  volumes:
    - ./deployments-config/docker-single
      setup/nginx.conf:/etc/nginx/nginx.conf
  ports:
    - "80:80"
```

We have defined a service with the name `database`, which will be the container for the Redis database instance. We set the `image` option to `redis` to tell `docker-compose` to run a container based on the Redis image.

Right after that, we define a service with the name `webapp`, which will be the container for the `igweb` application. We use the `depends_on` option to explicitly state that the `webapp` service needs the `database` service to function. Without the `database` service up, the `webapp` service cannot be brought up.

We specify the `build` option to tell `docker-compose` to build an image based on the `Dockerfile` in the path specified. By specifying the relative path of `..`, we indicate that the `Dockerfile` that exists in the current directory should be used to build the base image.

We specify a value of `8080:8080` (HOST:CONTAINER) for the `ports` section to indicate that we want to open up port `8080` on the host and forward connections to port `8080` of the Docker container.

We have defined the service with the name `reverseproxy`, which will be the container for the `nginx` reverse proxy server. We set the `depends_on` option to `webapp` to indicate that the `reverseproxy` service cannot be brought up without the `webapp` service being up. We've set the `image` option to `nginx` to tell `docker-compose` to run a container based on the `nginx` image.

In the `volumes` section, we can define our mount paths, in the form of HOST:CONTAINER. We have defined a single mount path where we have mounted the `nginx` configuration file, `nginx.conf`, located in the `./deployments-config/docker-single-setup` directory to the `/etc/nginx/nginx.conf` path inside the container.

Since the reverseproxy service will be servicing HTTP client requests, we specify a value of 80:80 for the ports section to indicate that we want to open up port 80 (the default HTTP port) on the host and forward connections to port 80 of the Docker container.

Now that we've gone through the Docker Compose configuration file, it's time to start up igweb as a multi-container Docker application using the `docker-compose up` command.

## Running Docker compose

We issue the following command to build the services:

```
$ docker-compose build
```

Here's the output of running the `docker-compose build` command (some part of the output has been omitted for brevity):

```
database uses an image, skipping
Building webapp
Step 1/22 : FROM golang
--> 99e596fc807e
Step 2/22 : MAINTAINER Kamesh Balasubramanian kamesh@kamesh.com
--> Running in 107a99d5c4ee
--> 6facac83509e
Removing intermediate container 107a99d5c4ee
Step 3/22 : ENV IGWEB_APP_ROOT /go/src/github.com/EngineerKamesh/igb/igweb
--> Running in f009d8391fc4
--> ec1b1d15c6c3
Removing intermediate container f009d8391fc4
Step 4/22 : ENV IGWEB_DB_CONNECTION_STRING "database:6379"
--> Running in 2af5e98c71e2
--> 6748f0f5bc4d
Removing intermediate container 2af5e98c71e2
Step 5/22 : ENV IGWEB_MODE production
--> Running in 1a87b871f761
--> 9871fc511e80
Removing intermediate container 1a87b871f761
Step 6/22 : ENV GOPATH /go
--> Running in c6c2eff0ded2
--> 4dc456357dc9
Removing intermediate container c6c2eff0ded2
Step 7/22 : RUN go get -u github.com/gopherjs/gopherjs
--> Running in c8996108bd96
--> 6ae68fb84178
Removing intermediate container c8996108bd96
Step 8/22 : RUN go get -u honnef.co/go/js/dom
```

```
----> Running in a1ad103c4c10
----> abd1f7f3b8b7
Removing intermediate container a1ad103c4c10
Step 9/22 : RUN go get -u -d --tags=js github.com/gopherjs/jsbuiltin
----> Running in d7dc4ec21ee1
----> cd5829fb609f
Removing intermediate container d7dc4ec21ee1
Step 10/22 : RUN go get -u honnef.co/go/js/xhr
----> Running in b4e88d0233fb
----> 3fe4d470799e
Removing intermediate container b4e88d0233fb
Step 11/22 : RUN go get -u github.com/gopherjs/websocket
----> Running in 9cebc021cb34
----> 20cd1c09d6cd
Removing intermediate container 9cebc021cb34
Step 12/22 : RUN go get -u github.com/tdewolff/minify/cmd/minify
----> Running in 9875889cc267
----> 3c60c2de51b0
Removing intermediate container 9875889cc267
Step 13/22 : RUN go get -u github.com/isomorphicgo/isokit
----> Running in eb839d91588e
----> e952d6e6cbe2
Removing intermediate container eb839d91588e
Step 14/22 : RUN go get -u github.com/uxtoolkit/cog
----> Running in 3e6853ff7196
----> 3b00f78e5acf
Removing intermediate container 3e6853ff7196
Step 15/22 : RUN go get -u github.com/EngineerKamesh/igb
----> Running in f5082861ca8a
----> 93506a92526c
Removing intermediate container f5082861ca8a
Step 16/22 : RUN go install github.com/tdewolff/minify
----> Running in b0a72d9e9807
----> e3e49d9c2898
Removing intermediate container b0a72d9e9807
Step 17/22 : RUN cd $IGWEB_APP_ROOT/client; go get ./...; /go/bin/gopherjs
build -m --verbose --tags clientonly -o
$IGWEB_APP_ROOT/static/js/client.min.js
----> Running in 6f6684209cfcd
Step 18/22 : RUN go install github.com/EngineerKamesh/igb/igweb
----> Running in 17ed6a871db7
----> 103f12e38c04
Removing intermediate container 17ed6a871db7
Step 19/22 : RUN /go/bin/igweb --generate-static-assets
----> Running in d6fb5ff48a08
Generating static assets...Done
----> cc7434fbb94d
Removing intermediate container d6fb5ff48a08
```

```

Step 20/22 : RUN /go/bin/minify --mime="text/css"
$IGWEB_APP_ROOT/static/css/igweb.css >
$IGWEB_APP_ROOT/static/css/igweb.min.css
    --> Running in e1920eb49cc2
    --> adb78450b9c
Removing intermediate container e1920eb49cc2
Step 21/22 : ENTRYPOINT /go/bin/igweb
    --> Running in 20246e214462
    --> a5f1d978060d
Removing intermediate container 20246e214462
Step 22/22 : EXPOSE 8080
    --> Running in 6e12e970dfe2
    --> 4c7f474b2704
Removing intermediate container 6e12e970dfe2
Successfully built 4c7f474b2704
reverseproxy uses an image, skipping

```

After the build is complete, we can go ahead and run the multi-container `igweb` application by issuing the following command:

```
$ docker-compose up
```

Figure 11.5 is a screenshot of IGWEB running as a multi-container application:

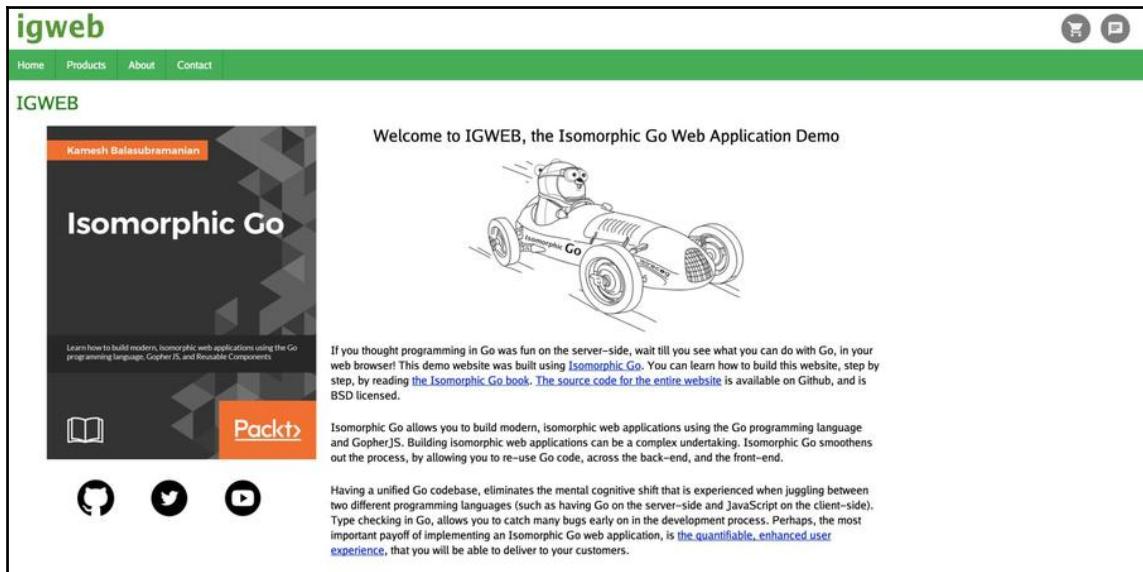


Figure 11.5: IGWEB running as a multi-container application

When we run the `docker-compose up` command, the command provides us live output of activity across all of the running containers. To exit the program, you can use the *Ctrl + C* key combination. Note that this will terminate the `docker-compose` program, which will shut down the running containers in a graceful manner.

Alternatively, when starting the multi-container `igweb` application, you may specify the `-d` option to run in detached mode, which will run the containers in the background, like so:

```
$ docker-compose up -d
```

If you wish to bring the multi-container application down, you can issue the following command:

```
$ docker-compose down
```

If you make further changes to the `Dockerfile` or the `docker-compose.yml` file, you must run the `docker-compose build` command again to rebuild the services:

```
$ docker-compose build
```

It's convenient to have the facility of `docker-compose up -d` to run the containers in the background, but by now, we know that it would be best to turn our multi-container Docker application into a `systemd` service.

## Setting up the dockerized IGWEB service

Setting up the `systemd` service for the dockerized `igweb` is pretty straightforward. Here are the contents of the `igweb-docker.service` file, which should be placed in the production system's `/etc/systemd/system` directory:

```
[Unit]
Description=Dockerized IGWEB
After=docker.service
Requires=docker.service

[Service]
Type=oneshot
RemainAfterExit=yes
WorkingDirectory=/opt/igb/igweb
ExecStart=/usr/bin/docker-compose -f /opt/igb/igweb/docker-compose.yml up -d
ExecStop=/usr/bin/docker-compose -f /opt/igb/igweb/docker-compose.yml down

[Install]
```

```
WantedBy=multi-user.target
```

In the `[Unit]` section, we have set the `After` directive with the value `docker.service`. This indicates that the `docker` unit must be started before the `igweb-docker` unit. The `Requires` directive has also been set with the value `docker.service`. This indicates that the `igweb-docker` unit is dependent on the `docker` unit to successfully run. Failure to start the `docker` unit will result in a failure to start the `igweb-docker` unit.

In the `[Service]` section, we have set the `Type` directive to `oneshot`. This indicates that the executable we are launching is short-lived. It makes sense to use it because we will be running `docker-compose` up with the `-d` flag specified (detached mode) so that the containers run in the background.

We have specified the `RemainAfterExit` directive in conjunction with the `Type` directive. By setting the `RemainAfterExit` directive to `yes`, we indicate that the `igweb-docker` service should be considered active even after the `docker-compose` process exits.

Using the `ExecStart` directive, we start the `docker-compose` process in detached mode. We have specified the `ExecStop` directive to indicate the command that is needed to stop the service.

In the `[Install]` section by setting the value for the `WantedBy` directive to `multi-user.target`, we specify that this service has a system runlevel of 3 (multi-user mode).

Recall that after placing the `igweb-docker.service` file in the `/etc/systemd/system` directory, we must reload the `systemd` daemon like so:

```
$ systemctl daemon-reload
```

Now, we can start the dockerized `igweb` application:

```
$ systemctl start igweb-docker
```

You may use the `systemctl enable` command to specify that `igweb-docker` should be started on system startup.

We can take down the service by issuing the following command:

```
$ systemctl stop igweb-docker
```

At this point, we've demonstrated how to run the `igweb` application as a multi-container Docker application hosted on the Linode cloud. Again, although we are using Linode, the procedure that we have demonstrated can be replicated on your preferred cloud provider of choice.

## Summary

In this chapter, we learned how to deploy an isomorphic web application to the cloud. We presented how the `igweb` server-side application operates in production mode, showing you how external CSS and JavaScript source files were included by the application. We also showed you how to tame the file size of the JavaScript program produced by GopherJS. We showed you how to generate static assets for the application's template bundle along with the JavaScript and CSS that were to be used by the deployed cogs.

We first considered the deployment of an isomorphic web application to a standalone server. This consisted of adding an `igweb` user to the server, setting up the `redis-server` instance, setting up `nginx` as a reverse proxy with GZIP compression enabled, and setting up the `igweb` root folder. We also showed you how to cross compile Go code from the development system (64-bit macOS) to the operating system running on the production system (64-bit Linux). We guided you through the process of preparing a deployment bundle, and then we deployed the bundle to the production system. Finally, we showed you how to set up `igweb` as a `systemd` service so that it could easily be started, stopped, restarted, and automatically started on system startup.

We then focused our attention to the deployment of an isomorphic web application as a multi-container Docker application. We showed you how to install Docker on the production system. We walked you through the process of dockerizing `igweb`, which consisted of creating a `Dockerfile`, defining the services that make up IGWEB in a `docker-compose.yml` file, and running the `docker-compose up` command to start up IGWEB as a multi-container Docker application. Finally, we showed you how to set up the `igweb-docker` `systemd` script to manage `igweb` as a system service.

# Debugging Isomorphic Go

Debugging an Isomorphic Go web application consists of the following items:

- Identifying compiler/transpiler errors
- Examining panic stack traces
- Tracing code to pinpoint the source of issues

## Identifying compiler/transpiler errors

We can think of programming as a conversation between you (the programmer) and the machine (the compiler/transpiler). Because Go is a typed language, we can find many errors at compile/transpile time itself. This is a clear advantage to writing vanilla JavaScript where problems (caused by the lack of type checking) can lay hidden and surface at the most inopportune times. Compiler errors are the means by which the machine communicates to us that something is fundamentally wrong with our program, whether it's a mere syntactical issue or the inappropriate use of a type.

`kick` comes in very handy for displaying compiler errors, since it will show you errors from both the Go compiler and the GopherJS transpiler. The moment you introduce an error (that the compiler/transpiler can identify) and save your source file, you will see the error displayed in the terminal window where you have `kick` running.

For example, let's open up the `client/client.go` source file. In the `run` function, let's comment out the line where we set the `ts` variable to the `TemplateSet` object that we receive over the `templateSetChannel`:

```
//ts := <-templateSetChannel
```

We know that the `ts` variable will be used later to populate the `TemplateSet` field of the `env` object. Let's set the `ts` variable to the Boolean value of `false` by introducing the following code:

```
ts := false
```

The moment we save the `client.go` source file, `kick` will give us a *kick* (pun intended), about the error that we just introduced as shown in *Figure A1*:



The screenshot shows a terminal window with the following text:

```
balans0:~ kamesh$ kick --appPath=$IGWEB_APP_ROOT --gopherjsAppPath=$IGWEB_APP_ROOT/client --mainSourceFile=igweb.go
Instant KickStart Applied! (Recompiling and restarting project.)
client.go:92:20: cannot use ts (variable of type bool) as *github.com/isomorphicgo/isokit.TemplateSet value in assignment
```

Figure A1: The `kick` command immediately shows us the transpiler error upon saving the Go source file

The compiler error received shows us the exact line where the problem occurred, from which we can diagnose and rectify the issue. The lesson to be learned from this example is that it comes in very handy to have a terminal window running `kick` in the background while you are developing your Isomorphic Go web application. By doing so, you will be able to see compiler/transpiler errors the moment that you make them.

## Examining panic stack traces

For runtime errors that cannot be found by the transpiler at *transpile time*, there usually is a helpful **panic stack trace**, which is displayed in the web browser's console, and provides us with valuable information to diagnose issues. The JavaScript source map file that GopherJS produces helps the web browser map the JavaScript instructions to their respective lines in the Go source files.

Let's introduce a runtime error whereby our client-side program is syntactically correct (it will pass the transpiler checks); however, the code will issue a problem at runtime.

Going back to the `run` function found in the `client/client.go` source file, notice the following code changes we've made with regards to the `ts` variable:

```
func run() {
    println("IGWEB Client Application")

    // Fetch the template set
    templateSetChannel := make(chan *isokit.TemplateSet)
    funcMap := template.FuncMap{"rubyformat": templatefuncs.RubyDate,
        "unixformat": templatefuncs.UnixTime, "productionmode":
        templatefuncs.IsProduction}
    go isokit.FetchTemplateBundleWithSuppliedFunctionMap(templateSetChannel,
        funcMap)
    // ts := <-templateSetChannel

    env := common.Env{}
    // env.TemplateSet = ts
    env.TemplateSet = nil
    env.Window = dom.GetWindow()
    env.Document = dom.GetWindow().Document()
    env.PrimaryContent = env.Document.GetElementByID("primaryContent")
    env.Location = env.Window.Location()

    registerRoutes(&env)
    initializePage(&env)
}
```

We have commented out the declaration and initialization of the `ts` variable, and we have also commented out the assignment of the `ts` variable to the `TemplateSet` field of the `env` object. We have introduced a line of code to assign the `nil` value to the `TemplateSet` field of the `env` object. By taking this action, we have essentially disabled the client-side template set, which will prevent us from being able to render any template on the client-side. This also prevents any cog from being rendered, since cogs are dependent on the template set to properly function.

After loading the IGWEB home page, a panic stack trace is generated and is visible in the web browser's console as shown in *Figure A2*:

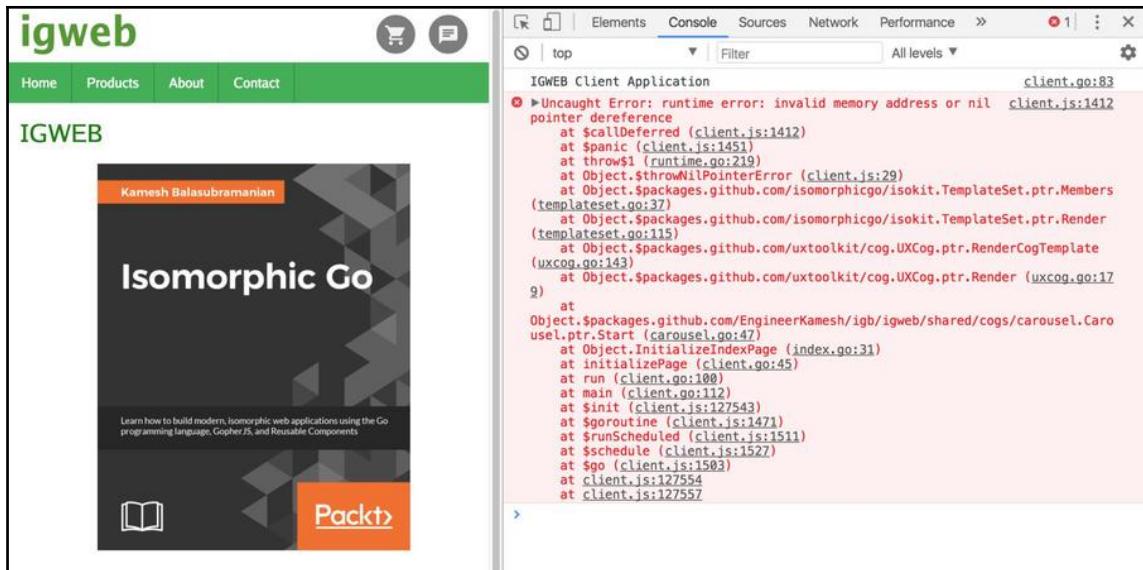


Figure A2: The panic stack trace shown in the web browser's console

In your front-end debugging travels, you will come across the following error message quite often:

```
client.js:1412 Uncaught Error: runtime error: invalid memory address or nil
pointer dereference
```

runtime error: invalid memory address or nil pointer dereference usually means that we are attempting to perform operations on a value (such as accessing or mutating a property) that is equal to a JavaScript `null` value.

Examining the produced panic stack trace helps us zero in on the issue:

```
Uncaught Error: runtime error: invalid memory address or nil pointer
dereference
at $callDeferred (client.js:1412)
at $panic (client.js:1451)
at throw$1 (runtime.go:219)
at Object.$throwNilPointerException (client.js:29)
at Object.$packages.github.com/isomorphicgo/isokit.TemplateSet.ptr.Members
(templateset.go:37)
at Object.$packages.github.com/isomorphicgo/isokit.TemplateSet.ptr.Render
```

```
(templateSet.go:115)
at Object.$packages.github.com/uxtoolkit/cog.UXCog.ptr.RenderCogTemplate
(uxcog.go:143)
at Object.$packages.github.com/uxtoolkit/cog.UXCog.ptr.Render
(uxcog.go:179)
at
Object.$packages.github.com/EngineerKamesh/igb/igweb/shared/cogs/carousel.C
arousel.ptr.Start (carousel.go:47)
at Object.InitializeIndexPage (index.go:31)
at initializePage (client.go:45)
at run (client.go:100)
at main (client.go:112)
at $init (client.js:127543)
at $goroutine (client.js:1471)
at $runScheduled (client.js:1511)
at $schedule (client.js:1527)
at $go (client.js:1503)
at client.js:127554
at client.js:127557
```

The areas of interest in the panic stack trace are shown in bold. From the panic stack trace, we can ascertain that the carousel cog failed to render, since it appears that something is wrong with the `TemplateSet`. By further inspecting the panic stack trace, we can identify that the call was made to the `run` function at line 112 in the `client.go` source file. The `run` function is where we had introduced the error by setting the `TemplateSet` field of the `env` object to `nil`. From this debugging exercise, we can see that in this situation, the panic stack trace did not reveal the exact line of the problem but it provided us with enough clues to rectify the issue.

A good practice to follow when developing on the client-side is to always have the web browser's console open, so that you will be able to see problems as they occur.

## Tracing code to pinpoint the source of issues

Another good client-side debugging practice is **tracing**, the practice of printing out key steps in the flow of a program. In a debugging scenario, this would consist of strategically making calls to the `println` (or `fmt.Println`) function around suspected areas of problematic code. You can use the web browser's console to verify that these print statements are reached, which will give you a better understanding of how the client-side program is functioning while it is running.

For example, when debugging the issue introduced in the previous section, we can place the following `println` calls in the `run` function:

```
func run() {
    //println("IGWEB Client Application")
    println("Reached the run function")
    // Fetch the template set
    templateSetChannel := make(chan *isokit.TemplateSet)
    funcMap := template.FuncMap{"rubyformat": templatefuncs.RubyDate,
        "unixformat": templatefuncs.UnixTime, "productionmode":
        templatefuncs.IsProduction}
    go isokit.FetchTemplateBundleWithSuppliedFunctionMap(templateSetChannel,
        funcMap)
    // ts := <-templateSetChannel
    println("Value of template set received over templateSetChannel: ", <-
        templateSetChannel)
    env := common.Env{}
    // env.TemplateSet = ts
    env.TemplateSet = nil
    env.Window = dom.GetWindow()
    env.Document = dom.GetWindow().Document()
    env.PrimaryContent = env.Document.GetElementByID("primaryContent")
    env.Location = env.Window.Location()
    println("Value of template set: ", env.TemplateSet)
    registerRoutes(&env)
    initializePage(&env)
}
```

We performed tracing by printing key steps in the flow of the program, by making strategic `println` function calls. The first `println` call is used to verify that we reach the `run` function. The second `println` call is used to check the health of the template set that is returned to us from the template set channel. The third, and final `println` call, is used to check the health of the template set after we have completed prepped the `env` object by populating its fields.

Figure A3 shows the web console with the print statements displayed, along with the respective line number in the `client.go` source file where the `println` call was made:

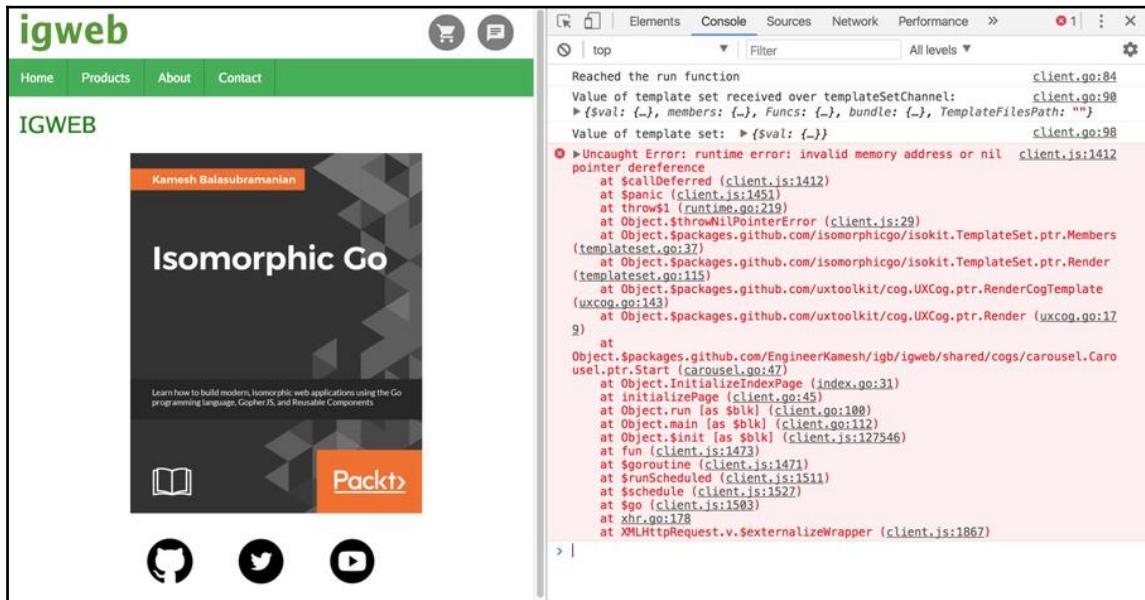


Figure A3: The print statements displayed in the web console

From the tracing exercise, we can first verify that we have successfully reached the `run` function. Secondly, we can verify the health of the `TemplateSet` object received over the `templateSetChannel` by noticing that properties of the object appear (such as `members`, `Funcs`, and `bundle`). The third, and final print statement, also verifies the health of the `TemplateSet` object after the `env` object has been prepped. This print statement reveals the source of the problem by showing us that the `TemplateSet` object has not been initialized, since we don't see any of the properties of the object appear in the printed statement.

# Index

## A

about page  
gopher team Rest API endpoint 166, 167, 168  
rendering, from client side 164, 165  
rendering, from server side 156, 157, 158  
AboutHandler function  
responsibilities 156  
AJAX web application architecture  
about 19  
advantages 20, 21  
disadvantages 21, 22, 23  
application programming interfaces (APIs) 76  
application root environment variable  
setting up 51, 52  
Artificial Intelligence (AI) 290  
Asynchronous JavaScript And XML (AJAX) 19

## C

carousel cog 363, 364, 365, 366, 367, 369  
CasperJS  
about 389, 390  
carousel cog, verifying 429  
client-side routing, verifying 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403  
contact form, verifying 403, 404, 405, 406, 407, 408, 409, 410, 411  
date picker cog, verifying 427, 429  
live chat feature, verifying 416, 417, 418, 419, 420, 421  
live clock cog, verifying 424, 425  
notify log, verifying 432, 434  
shopping cart functionality, verifying 411, 412, 413, 414, 415, 416  
template rendering, verifying 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403

time ago cog, verifying 422, 424  
channels 12  
chat agent conversation 319, 321, 322, 323  
classic web application architecture  
about 13, 14  
advantages 15, 16  
disadvantages 17, 18  
client side  
shopping cart functionality, implementing on 231  
client-side application  
transpiling 52  
client-side considerations 272, 275, 277  
client-side functionality  
testing 388  
client-side handler functions  
about 188  
for product detail page 191, 192  
for products listing page 188, 189  
list of products, fetching 190  
product detail, fetching 192, 193  
client-side routes  
registering, with isokit router 187, 188  
client-side validation  
checking 279, 283  
result, tampering 284, 286  
code organization 65, 66, 67, 68  
code  
tracing, to pinpoint source of issues 475, 476, 477  
cogs  
about 327  
anatomy 331, 332, 333, 334  
life cycle 338, 339  
pure cogs 339, 340  
time ago cog 344  
UX toolkit 328, 329, 330  
virtual DOM tree 335, 337

command-line interface (CLI) 54  
compiler errors  
  identifying 471, 472  
Contact form Rest API endpoint 277, 279  
contact form  
  accessibility 259, 265  
  contact confirmation route handler 256  
  contact form submission, processing 257  
  contact route handler 254, 256  
  contact route, registering 253  
  designing 244, 246  
  functioning, without JavaScript 266, 272  
  implementing 250, 253  
  templates, implementing 246, 248  
content template  
  data, feeding 144, 146  
  for about page 141, 143  
custom datastore 69  
custom template functions 143, 144

## D

data object 128  
date picker cog 354, 355, 356, 357, 358, 359, 361, 362  
dependency injections 71, 73  
Docker Compose  
  about 463  
  running 465, 468  
Docker  
  installing 459  
  reference 458  
  used, for deploying Isomorphic Go web application 458  
Dockerfile  
  about 459, 460, 461  
  for closed source project 462  
dockerized IGWEB service  
  setting up 468  
Document Object Model (DOM)  
  about 17, 76, 77  
  accessing 79  
  alert message, displaying 80, 81  
  CSS style property, modifying of element 83  
  element, obtaining by ID 82  
  manipulating 79

operations 79  
query selector 82  
URL 78  
dynamic resource 14

**E**

email address syntax  
  validating 248  
ERDA (Encode-Register-Decode-Attach) 199

## F

filesystem-based template rendering  
  limitations 147, 148  
form flow 242  
form interface 249, 250

## G

GetGopherTeamEndpoint function 166  
Go  
  about 38, 39, 40, 41  
  installing 41, 42, 43, 44  
  programs, building 45, 46  
  programs, executing 45, 46  
  references 39  
  testing framework 380  
  workspace, setting up 44, 45  
golang docker image  
  reference 460  
GopherJS documentation  
  URL 81  
GopherJS Playground  
  URL 83  
GopherJS-produced JavaScript file size  
  taming 440, 441  
GopherJS-produced JavaScript source file 439  
GopherJS  
  about 12, 38, 46  
  alert message, displaying 93, 95  
  bindings, installing 48  
  command line 49  
  dom package 48  
  element's CSS style property, modifying 95, 99  
  examples 88, 92  
  GopherJS transpiler 85

installing 47  
JavaScript typeof operator functionality 99  
jsbuiltin package 48  
overview 84  
text, transforming to lowercase with XHR post  
101, 105  
URL 86  
websocket package 49  
xhr package 48  
Gorilla Mux  
server-side routes, registering 181, 182  
goroutines 12  
Graphical User Interface (GUI) 267

## H

hybrid cogs  
carousel cog 363, 364, 365, 366, 367, 369  
date picker cog 354, 355, 356, 357, 358, 359,  
361, 362  
implementing 354  
notify cog 371, 372, 373, 374, 376, 377

## I

IGWEB demo  
about 61  
application root environment variable, setting up  
51, 52  
building 62  
client-side application, transpiling 52  
contact page 64  
executing 54, 55  
kick command, using 59  
live chat feature 64  
Redis, setting up 52, 53, 54  
reusable components 64  
sample dataset, loading 56, 57, 58  
setting up 51  
IGWEB page structure  
about 132  
footer 134  
header 133  
navigation bar 134  
primary content area 134  
top bar 134

IGWEB project  
custom datastore 69, 71  
dependency injections 71, 73  
folder structure 65, 66, 68  
MVC pattern 68, 69  
structure 67  
IGWEB roadmap  
about 62  
about page 63  
home page 62  
products page 63  
shopping cart feature 63  
IGWEB, in production mode  
about 438  
GopherJS-produced JavaScript file size, taming  
440, 441  
GopherJS-produced JavaScript source file 439  
static assets, generating 441, 442, 443  
IGWEB  
dockerizing 459  
in-memory template set 148, 149, 150  
inline template  
cars listing demo 106, 109  
gob encoded data, transmitting 110, 113  
rendering 106, 110  
interactive elements  
initializing, on web page 163  
isokit package 148  
isokit router  
client-side routes, registering 187, 188  
Isomorphic Go toolchain  
Go 39, 40, 41  
GopherJS 46  
installing 38, 39  
Isomorphic Go toolkit  
about 38, 50  
isokit package, installing 50  
URL 50  
Isomorphic Go web application  
deploying, Docker used 458  
Isomorphic Go web application, deploying to  
standalone server  
about 444  
bundle deployment 453  
deployment bundle preparation 451, 452

IGWEB root folder setup 450  
IGWEB, cross-compiling 450, 451  
IGWEB, running 453  
IGWEB, running in background 455  
IGWEB, running in foreground 454  
IGWEB, running with systemd 455, 456, 457  
IGWEB, starting 453  
NGINX reverse proxy setup 446  
Redis database instance setup 445  
server, provisioning 444

**Isomorphic Go**  
about 13  
code, transpiling to JavaScript 10  
front-end advantages 11, 12  
need for 9  
type checking 9  
Vanilla JavaScript, avoiding with transpilers 10

**isomorphic handoff procedure**  
about 199  
Attach step 202  
Decode step 202  
Encode step 201  
ERDA strategy 199  
implementing, for product detail page 209, 210, 211, 212  
implementing, for shopping cart 213  
Register step 201

**isomorphic handoff**  
about 198  
implementing, for product-related pages 203  
implementing, for products listing page 205, 206

**isomorphic template rendering**  
about 146  
about page, rendering from client side 164, 165  
about page, rendering from server side 156, 157, 158  
filesystem-based template rendering 148  
filesystem-based template rendering, limitations 147  
in-memory template set 148, 149, 150  
interactive elements, initializing on web page 163  
server-side handlers, registering 154, 155  
template bundle items, serving 155, 156  
template set, setting on client side 159, 160, 161, 162

template set, setting on server side 151, 152, 154  
**isomorphic web application architecture**  
about 29, 30  
live demo 33  
measurable benefits 34  
nomenclature 35  
prerequisites 35  
solutions, for disadvantages 30, 31, 32

## J

JavaScript Object Notation (JSON) 25

## K

**kick command**  
executing 59, 60  
installing 59  
using 59  
verifying 60, 61

## L

**layout template**  
about 135  
web page layout template 136

**Linode**  
reference 444

**live chat box templates**  
implementing 291

**live chat feature** 289, 290

**live chat's client-side functionality**  
close chat control 314  
disconnection event, handling 318  
event listeners, for WebSocket object 314, 315, 317  
event listeners, initializing 313  
implementing 310  
live chat client, creating 310, 311, 312

**live chat's server-side functionality**  
agent's brain 300, 301, 302, 304  
agent's information, exposing to client 309  
chat server, activating 299, 300  
client type 296, 297, 298  
Greeting method 305  
hub type 294, 295

human, greeting 305  
implementing 292, 293  
reply, to human's question 306, 307, 308  
live chatbox  
  designing 290  
live clock cog 347, 348, 349, 350, 351, 352  
local storage  
  about 113  
  client-side functionality, implementing 119  
  demo, executing 123, 125  
  entries, clearing 115  
  inspector web page, initializing 120  
  inspector, building 115  
  key value pairs, obtaining 115  
  key-value pair, setting 114  
  local storage inspector, implementing 120  
  operations 114  
  server-side route, setting up 119  
  user interface, creating 116  
  value, obtaining for given key 114  
Lynx  
  references 259

## M

minification 25  
Model-View-Control (MVC) 68  
mount point 330

## N

NGINX reverse proxy setup  
  GZIP compression, enabling 448  
  proxy settings 448, 449  
notify cog 371, 372, 373, 374, 376, 377

## P

page template  
  for about page 141  
panic stack traces  
  examining 472, 473, 474  
partial templates  
  about 135, 136  
  footer partial template 139  
  header partial template 137  
  navigation bar partial templates 138

top bar partial templates 138  
partials 136  
Pikaday  
  reference 327  
product data  
  accessing 178  
  modeling 177, 178  
product detail page  
  isomorphic handoff, implementing for 209, 210, 211, 212  
product detail  
  retrieving, from datastore 180  
product model  
  sort interface, implementing for 203  
product-related pages  
  design 173, 174  
  isomorphic handoff, implementing for 203  
product-related templates  
  implementing 174  
  templates, implementing for product detail page 176  
  templates, implementing for products listing page 175  
product  
  retrieving, from datastore 178  
products listing page  
  isomorphic handoff, implementing for 205, 206  
pure cogs  
  implementing 339, 340  
  live clock cog 347, 348, 349, 350, 351, 352  
  time ago cog 340, 341, 343, 346, 347

## Q

query selector 82

## R

ready state property  
  reference link 159  
reconciliation 337  
Redis  
  setting up 52, 53, 54  
regular template  
  about 135, 139  
  content template, for about page 141, 143  
  page template, for about page 141

## Rest API endpoints

- about 193
- client-side routing functionality, verifying 196
- list of products, obtaining 194
- product detail, obtaining 194
- route handler 15
- routing
  - about 170
  - on client side 171, 172
  - on server side 170

## S

### sample dataset

- loading 56, 57, 58

### search engine optimization (SEO) 174

### semantic URL 174

### server-side functionality

- contact form submission, verifying 386
- contact form's validation functionality, verifying 383, 385
- routing, verifying 381
- template rendering, verifying 381
- testing 380

### server-side handler functions

- about 182
- for product detail page 185, 186, 187
- for products listing page 183, 184

### server-side handlers

- registering 154, 155

### server-side routes

- registering, with Gorilla Mux 181, 182

### server-side routing

- verifying 381

### server-side shopping cart handler function 223, 224, 225

### server-side template rendering

- example 147

### session store 222

### shopping cart endpoints

- about 226
- endpoints, for adding items to shopping cart 227
- endpoints, for obtaining items in shopping cart 226
- endpoints, for removing from shopping cart 229

### shopping cart functionality

### implementing, on client side 231

- verifying 236, 237, 238

### shopping cart page

- designing 213, 215

### shopping cart routes

- about 220
- item, adding 221
- item, removing 221
- list of items, fetching 221

### shopping cart templates

- implementing 215, 216

### shopping cart

- isomorphic handoff, implementing for 213
- item, adding to 235
- item, removing from 233, 234
- modeling 218, 219
- rendering 231, 233

### single page application (SPA) architecture

- about 13, 23, 24
- advantages 25
- disadvantages 25, 26, 27, 28

### sort interface

- implementing, for product model 203
- static resource 14

## T

### template bundle items

- serving 156

### template categories

- about 135

### layout templates 135

### partial templates 135, 136

### regular templates 135, 139

### template data object 129, 131, 132

### template engine 128, 130

### template renderer 15

### template rendering

- verifying 381

### template set

- client-side route, registering 163

- client-side router, creating 162

- setting, on client side 159, 160, 161, 162

- setting, on server side 151, 152, 154

### templates

- about 128, 129, 132

implementing, for products listing page 175  
implementing, product detail page 176  
testing framework, Go 380  
time ago cog 340, 341, 343, 344, 346, 347  
tracing 475  
Transmission Control Protocol (TCP) 26  
transpiler errors  
    identifying 471, 472  
transpilers  
    about 10  
    Vanilla JavaScript, avoiding 10

## U

user experience (UX) 26  
UX toolkit, cogs  
    reference 331  
UX toolkit  
    about 38, 51, 328, 329, 330  
    cog package, installing 51  
URL 51

## V

virtual DOM tree 327, 335, 337  
virtual private server (VPS) 444

## W

web application architectures  
    overview 13  
web template system  
    about 128  
    template data object 131, 132  
    template engine 130  
    templates 132  
World Wide Web Consortium (W3C) 23

## X

XHR post  
    used, for transforming text to lowercase 101, 105  
XMLHttpRequest (XHR) 19