UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Marti Kaljuve

Cross-Browser Document Capture System

Master's Thesis (30 ECTS)

Supervisors: Marlon Dumas, Prof

Kaspar Loog, MSc

Author: .... „......“ May 2013

Supervisor: .... „......“ May 2013

Professor: .... „......“ May 2013

Tartu 2013

Abstract

A web page is seldom displayed in the exact same manner in different browser and operating system combinations. There are several reasons for different rendering outcomes: interpretation of web standards by the browser, the browser's rendering engine, available fonts in the operating system, plugins installed in the browser, screen resolution etc. Neglecting to consider these differences as a web designer may lead to webpage layout issues that result in lost customers.

Web designers might consider it common practice to test webpages on several browsers to eliminate cross-browser layout issues. Experiments show that finding visual differences is a dull and cumbersome task for people. Knowing this, another member working at Browserbite has created an algorithm that has proved to be much faster and more accurate at finding layout issues compared to humans. The algorithm works by comparing a baseline (*oracle* in software testing terms) webpage in image form to other image captures of the same webpage in different browsers, finding differences in layout and position that a human might consider erroneous.

This thesis concentrates on the problem of creating the input to the aforementioned algorithm. A selective overview of existing solutions and services for webpage capture and automation is given, measuring their performance where possible. A list of requirements are established for a cross-platform capture solution to be commercialized. A fast and cross-platform method of capturing full webpages is then introduced, and an overview of a scalable Software-as-a-Service system implemented for cross-browser and cross-platform capture in several virtual and physical machines asynchronously is given.

Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
AJAX	Asynchronous JavaScript and XML
AMQP	Advanced Message Queuing Protocol
ANSI	American National Standards Institute
API	Application Programming Interface
BitBlt	Bit-level block transfer
BSD	Berkeley Software Distribution
CSS	Cascading Style Sheets
CPU	Central Processing Unit
DOM	Document Object Model
EC2	Elastic Compute Cloud
ESB	Enterprise Service Bus
IE	Internet Explorer
GDI	Graphics Device Interface
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
MRI	Matz's Ruby Interpreter
MVC	Model-View-Controller
PDF	Portable Document Format
PPM	Portable pixmap
RAM	Random Access Memory
REST	Representational State Transfer
OS	Operating System
SOA	Service-oriented architecture
SQL	Structured Query Language
UI	User Interface
URL	Uniform Resource Locator
XML	Extensible Markup Language

Chapter 1 Contents

Chapter 1

Introduction

The share of web browsers usage is fragmented and very dynamic. The four most popular desktop browser families in use today are Microsoft Internet Explorer, Mozilla Firefox, Google Chrome and Safari, which today account for over 95% of the desktop browser market [1]. Several of these browsers are available for more than one operating system, which means that several browser and OS combinations can be used to access a web page. Market share of mobile browsers is also increasing at a rapid pace, resulting in an even larger fragmentation of operating systems, browsers and screen resolutions.

Despite web standards [2] that describe how mark-up languages and style sheet languages should be translated into visual web pages, in reality the implementations of these standards by different browser engines and versions are inconsistent. This often results in layout or behaviour discrepancies on the same web page. For example, in the beginning of 2011, it was discovered that when the Estonian Air website was viewed in Firefox on Mac, the button for booking flights was not visible, as can be seen in Figure 1.

These layout differences have long been a problem for web designers. In some cases they have brought about campaigns to encourage users to upgrade their browser software. A popular example is the Internet Explorer 6 countdown [3], an official campaign by Microsoft encouraging users to replace the over 10 year old browser, stating that it will save hours of work for web developers.

Experience has shown that it is illusory to expect that all users will continuously upgrade their browsers or that users will settle for one single browser and version. Software companies are pressed to support a wide array of browsers and browser versions and in different operating systems. On the other hand, the emergence of different types of devices, ranging from smartphones to lightweight notebooks, entails that these browsers will run in different configurations, to cater for different screen sizes, resolutions, input capabilities, etc.

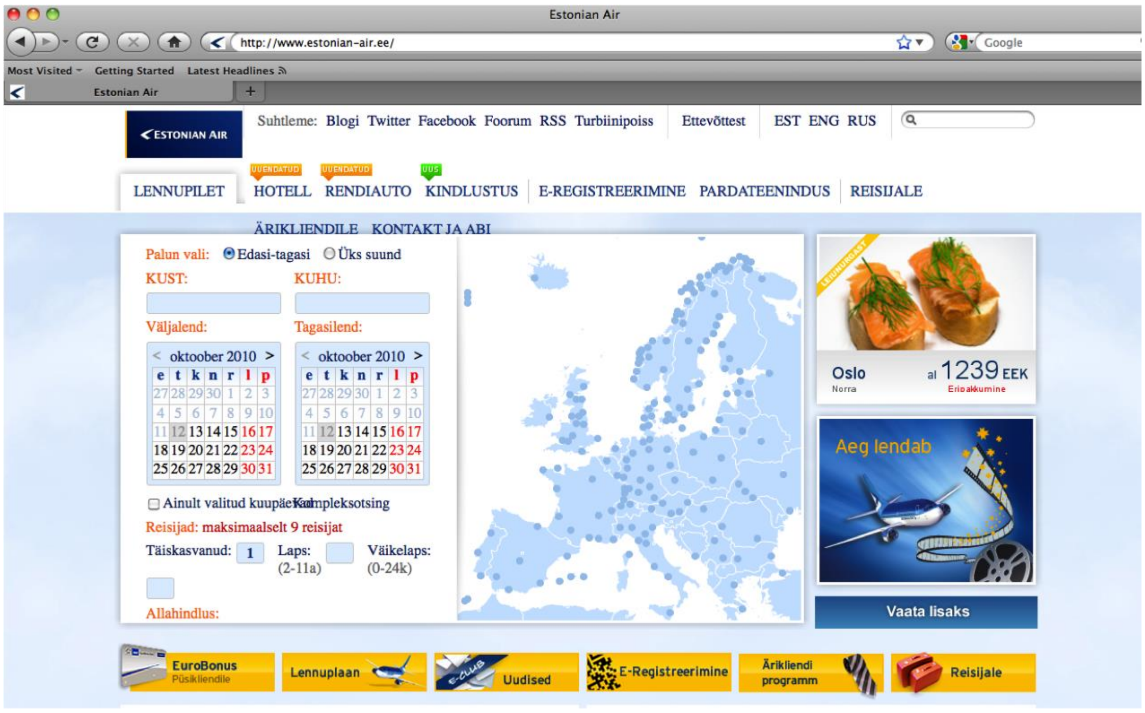


FIGURE 1 A SCREENSHOT OF FIREFOX ON MAC OS X SHOWING THE ESTONIAN AIR HOMEPAGE WITH THE "BOOK NOW" BUTTON MISSING FROM THE LEFT CONTAINER.

Faced with such imperatives, software development projects are forced to introduce cross-browser compatibility testing as an integral step in their quality assurance process. Generally speaking, cross-browser compatibility testing is the act of verifying (via test cases) that a given web page can be adequately rendered in different browsers. The notion of adequacy will greatly vary from one stakeholder to the other as beauty is in the eyes of the beholder. What is an adequate rendering for a developer or a user might not be an adequate rendering for a Web designer, who would typically have higher expectations on the fidelity of the rendering relative to their initial design.

Traditionally, cross-browser compatibility testing has been a manual task. Testers take a number of Uniform Resource Locators (URLs) and simply render them manually in different browsers and configurations, and check that the corresponding rendering meets their set expectations. They then report any potential incompatibilities back to developers who devise and implement a resolution. A range of solutions for partially automating the cross-browser compatibility workflow have emerged over recent years, including Mogotest [4], BrowserStack [5] and Browsershots [6]. These solutions automate the process of opening a given URL on multiple browsers and configurations, and taking a screenshot of the rendering. They then aggregate these screenshots and show them to the user. As an example, Figure 2 shows the rendering of the web page http://www.apple.com in Internet Explorer 9 running on Windows 7 given by the BrowserStack service.

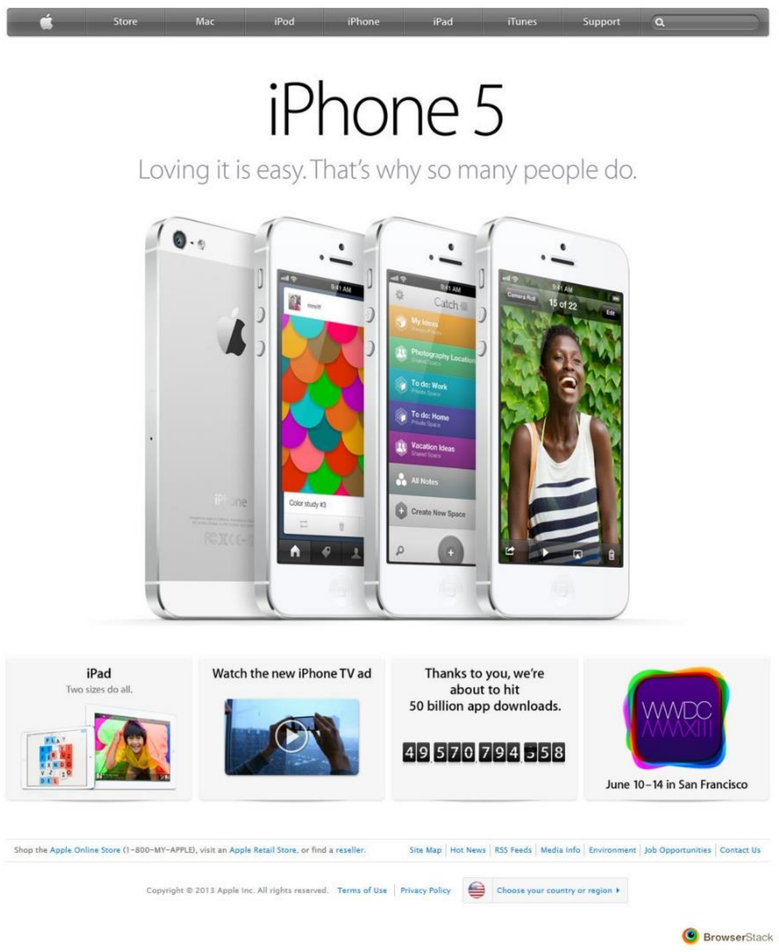

Store
Mac
iPod
iPhone
iPad
iTunes
Support
iPhone 5
Loving it is easy. That's why so many people do.
iPad
Two sizes do all.
Watch the new iPhone TV ad
Thanks to you, we're about to hit 50 billion app downloads.
49,570,794,558
WWDC MMXIII
June 10–14 in San Francisco
Shop the Apple Online Store (1-800-MY-APPLE), visit an Apple Retail Store, or find a reseller.
Site Map | Hot News | RSS Feeds | Media Info | Environment | Job Opportunities | Contact Us
Copyright © 2013 Apple Inc. All rights reserved. Terms of Use | Privacy Policy
Choose your country or region
BrowserStack

FIGURE 2 A RENDERING OF WWW.APPLE.COM IN WINDOWS 7 - INTERNET EXPLORER 9 BY BROWSERSTACK

One of the key challenges that Web page rendering engines have to address is that of scalability. The process of opening pages in different browsers, configurations and operating systems is computationally heavy, due to the cost of launching and running virtual machines in order to reproduce the exact environment in which a Web browser is expected to run. Also, large amounts of data needs to be manipulated as images can be large. A second challenge is to ensure that the entire Web page is rendered, as usually Web pages are not rendered in their entirety in a single view, but rather a partial view is given and page scrollbars are provided by the browser so that the user can view other parts of the Web page. Thirdly, the resolution of different browsers and browser configurations might make the images taken from different browser configurations incomparable and thus a normalization procedure needs to be applied. Fourthly, Web page rendering engines for cross-browser compatibility testing need to be extensible, so that support for additional browsers and browser configurations can be easily introduced, without requiring major recoding efforts.

The thesis at hand describes an architecture and implementation of a Web page rendering engine for cross-browser compatibility testing that tackles the above challenges. The described Web page rendering automation engine is currently running in production mode and is at the kernel of a product developed and marketed by Browserbite.

The rest of the thesis is organized as follows. Chapter 2 describes existing solutions that have taken on the same challenge of cross-browser webpage capture. The performance is briefly measured, where applicable. Chapter 3 introduces the requirements for a cross-browser capture solution that can be commercialized. Chapter 4 describes the implemented solution in detail. Chapter 5 lists some of the features that have yet to be implemented in the software. A conclusion is then made in chapter 6.

Chapter 2

Requirements

The main functional requirement of a cross-browser rendering (or capturing) system is to produce a set of screenshots (image file) that fully and accurately capture the rendering of a given Web document (identified by a URL) on a given set of browsers and browser configurations.

If we analyze this initial functional requirement further, we see that the main two criteria in this requirement are the completeness of the document capture (the “fully” adverb), and the accuracy.

With respect to completeness, an obvious requirement for a capturing solution is that web documents must be captured in full width and height. In other words, the aim is to produce an image that captures every part of the document regardless of its size or given browser viewport.

With respect to accuracy, an important requirement is that the produced image for a given browser and configuration is in all cases identical to the rendering that would be obtained by opening the Web document in question on said browser and configuration. In this context a browser configuration includes a particular device (e.g. notebook with a given resolution), operating system and a given assignment of values to the configuration parameters of the browser.

Additionally, since the aim is to compare screenshots taken across different browsers and configurations, a second requirement is that the screenshots taken for different browsers and configurations should be comparable. In particular, documents must be rendered in the same size as they would be viewed on a typical user’s screen. Based on global statistics, a typical desktop computer was defined as a screen with a 1024x768px [7] resolution and with the browser window maximized. In addition to the default plugins of each browser and operating system, Adobe Flash must be installed. This is a reasonable assumption given that Adobe Flash is reported to have 99% market penetration [8]. Nonetheless, the system should be able to be adaptable so that this requirement can be lifted in future.

In addition to the above core functional requirements, the system should also support HTTP (HyperText Transfer Protocol) Basic Authentication scheme in order to be able to crawl through authenticated pages and capture pages hidden behind those authentication screens.

Coming down to non-functional requirements, and as stated in Chapter 1, the system needs to be highly *extensible* in order to cope with the ever-evolving landscape of browsers and devices. This means that the system architecture should minimize wherever possible the effort required to incorporate support for a new browser or platform. It also implies that the core (screen capturing) components of the system should be *portable* so as to support an evolving set of platforms, including tablet and smartphone browsers.

Secondly, performance (processing-time) and scalability (additional resources required to cope with additional load) should be carefully kept in mind when designing the system. To support highly iterative development processes, the system should be able to capture a web page in a wide number of browsers and configurations in a matter of minutes, if not seconds.

Resource-intensive parts of the system must be horizontally or vertically scalable to handle increasing demand. An obvious way for the system to be scaled horizontally (or scaled out) is to increase the number of machines that capture a specific browser and operating system configuration. This allows the system to handle requests from several users simultaneously, while keeping average waiting times low. Vertical scaling (or scaling up) of the system is possible both by upgrading memory and CPU in physical servers as well as increasing the relevant allotted resources for virtual machines.

Related to scalability is the fact that the system should be deployable on the cloud in order to benefit from the elasticity of computing resources that public clouds such as Amazon Elastic Compute Cloud (EC2) offer [9]. This requirement entails that the system should be compatible to platforms supported by public clouds (*platform compatibility*).

The requirement to deploy on the cloud also imposes *resource constraints,* meaning that the system should be developed while keeping in mind available resources, including CPU, memory and network bandwidth constraints.

Finally, the system should be robust, specifically it should handle errors in browsers and desktops gracefully and recover autonomously to a working state, including after a system-wide restart. In a similar vein, the system should clear the browser cache for every capture so as to avoid interference between two captures.

Chapter 3

Existing solutions

Before building a cross-browser page capturing solution to meet the previously defined requirements, existing web- and desktop-based browser automation solutions were considered and researched to potentially use as input to the visual comparison algorithm. Many services, some even by corporations such as Adobe and Microsoft [10], exist to render webpages in different browsers and operating systems, but only a few provide an Application Programming Interface (API) to use the captures externally.

In 2009, Microsoft introduced SuperPreview [11], a visual debugging tool that renders pages in Windows and Macintosh computers and provides DOM tree information. SuperPreview is included in Microsoft Expression Blend 4, a Windows application for creating graphical interfaces for web and desktop applications. No public API is available.

Mogotest [4] is one of the most prominent web services specializing in cross-browser testing. The browsers are run in a cloud environment, namely Amazon EC2. The list of supported browsers consists of Internet Explorer 6 up to 9, Firefox 3.6 up to 10 and the latest Google Chrome. The service also offers a web API to create captures and retrieve the results.

Browsershots [12] is a web service capable of capturing browsers on Windows, Linux, Mac and Berkeley Software Distribution (BSD) computers. There are two interesting aspects of this project - it is open-sourced and crowd-sourced. By crowd-sourcing the capturing of webpages to volunteers who register their own „shot factories“, the service is able to provide screenshots of a large number of browsers on different operating systems.

The documentation of Browsershots reveals that a single shot factory can process circa one screenshot per minute. The capture method is revealed by looking at the source code [13] – a web page is vertically scrolled and captured in small increments. After every scroll, the new capture is stitched together by analysing lines in PPM format.

PPM [14] is a verbose image file format where every pixel is represented by three decimal numbers for the red, green and blue component, separated by spaces or other “white space” characters.

A comparison of the main solutions considered before implementing the capture system described in this thesis is given in Table 1.

	SuperPreview	**Mogotest**	**Browsershots**	**BrowserStack**
Full page („scrolled“) screenshots	Yes	Partial (only desktop browsers)	Scrolled	Partial (only viewport for Safari, Opera)
Speed of rendering a screenshot	1-2 minutes after submitting request	20-60 seconds	5 minutes to hours, depending on queue size	15-60 seconds
Browser coverage	Internet Explorer, Firefox	Chrome, Firefox, IE, iOS	Chrome, Firefox, IE, Safari, Opera etc.	Chrome, Firefox, IE, Safari, Opera
API availability	No	Yes	Limited	Yes
Real browser or emulated	Unknown	Real browser	Real browser	Real browser
Support for scripting	No	No	No	No

TABLE 1 COMPARISON OF FEATURES OF EXISTING SOLUTIONS (AS OF 19TH OF FEBRUARY 2013)

It should also be mentioned that, as of May 2013, several cloud-based solutions exist for running automated tests on web pages (e.g. TestingBot [15], Nerrvana [16] and browserling [17]), but which do not have full-page capturing capabilities.

Comparing the requirements, existing solutions and potential future needs, a new solution was selected to be implemented since the existing solutions did not meet speed nor future scripting support needs.

Chapter 4

Implemented solution

The capture system described in this thesis consists of a web server, a database, a key-value store and worker processes distributed over several virtual and physical machines. The database, key-value store and web server are running in physical machines, while most of the cross-platform worker processes are in a private cloud of virtual machines.

PostgreSQL [19] is used as the main database to store persisted data, including browser and operating system configurations, user accounts, webpage requests, captures, comparisons and paths to captured images. PostgreSQL is an open-source object-relational database management system that is actively developed and supported on all major operating systems, including Linux, FreeBSD, Microsoft Windows and Mac OS X. The SQL implementation conforms strongly to the SQL:2008 ISO and ANSI standard, is fully ACID compliant and supports foreign keys, joins, views, triggers and several procedural languages to be executed by the database server. ACID stands for Atomicity, Consistency, Isolation and Durability - a set of properties that guarantee that database transactions are processed reliably [20].

In addition to a relational database, Redis [21] is used in the implementation of the capture system. Redis is a key-value store that by default stores its whole dataset in RAM and allows for optional durability by periodically storing changes to disk. In addition to string values it also supports lists of strings, sets of strings, sorted sets of strings and hashes where keys and values are strings. To increase read scalability and data redundancy, Redis servers can be easily replicated in a master-slave configuration. Benchmarks published on the Redis website promise sustained performance of 50,000 queries/second for more than 60,000 concurrent connections [22]. Redis is sponsored by VMware, Inc., a company providing cloud and virtualization software and services.

Many languages were considered for the cross-platform parts of this project, including Java, C#, C++, JavaScript and Ruby. Ruby was chosen because of the author's familiarity of the language and its dynamic nature, allowing quick prototyping of different solutions. The reference implementation, Ruby MRI (short for Matz’s Ruby Interpreter) [23], is written in C and has support for writing extensions in C and C++. A large user community is actively developing open-source libraries (gems) and tools for the Ruby language, which makes it a popular choice among developers.

In production, the implemented solution currently offers capture and comparison of 15 configurations, consisting of different versions of Google Chrome, Mozilla Firefox, Opera and Internet Explorer running on Windows XP, Windows Vista, Windows 7 and Mac OS X. The choice of configurations was decided based on Browserbite customer feedback.

All shot worker machines were configured to proxy requests through a local server that runs Squid [24], an open-source web cache and proxy server. A proxy has several advantages when used in this system. By caching the web pages accessed by shot workers for a predefined time (currently 60 seconds), it both reduces the load of the remote web server as well as increases the speed at which a webpage is loaded in all shot workers. The proxy is also configured to block certain resources from being loaded, for example executables and large files to increase robustness. This makes it safer for shot workers to open web pages that would otherwise automatically initiate file downloads, as is the case for many software vendors.

Workflow

A simplified flowchart of the implemented capture system is shown in Figure 3.

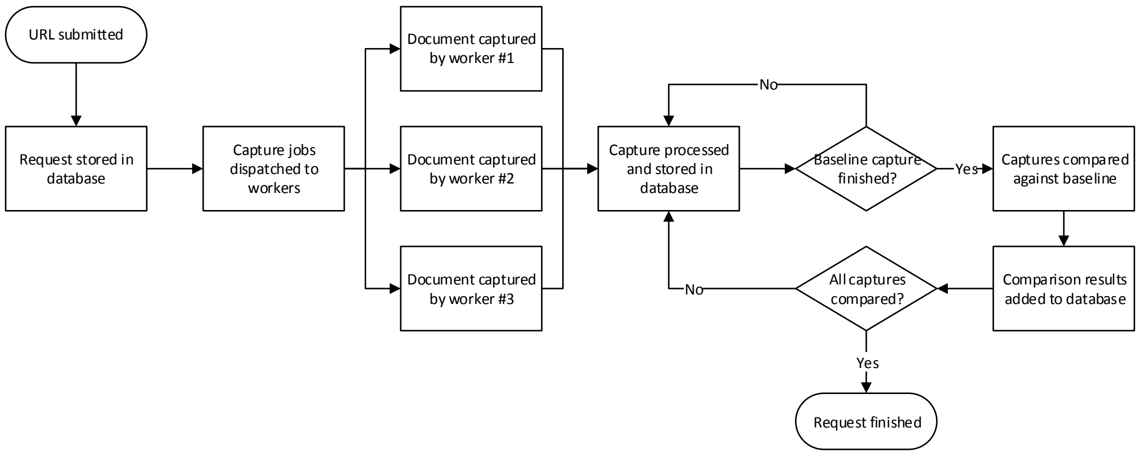

URL submitted
Request stored in database
Capture jobs dispatched to workers
Document captured by worker #1
Document captured by worker #2
Document captured by worker #3
Capture processed and stored in database
No
Baseline capture finished?
Yes
Captures compared against baseline
Comparison results added to database
All captures compared?
No
Yes
Request finished

FIGURE 3 A SIMPLIFIED FLOWCHART OF THE CROSS-BROWSER DOCUMENT CAPTURE SYSTEM.

Based on potential CPU usage and time consumption, the primary bottlenecks in the capture solution's workflow were recognized as shot workers loading and capturing webpages and the comparison algorithm running on completed captures. As usage of the capture system increases, these resource-intensive processes should be parallelized to maintain low average cycle times in the capture system. An elegant way to achieve this is to incorporate a queuing system.

Several protocols and solutions for sending and receiving messages between distributed systems were considered. The selection included the Advanced Message Queuing Protocol [25] (AMQP), an open standard for connecting systems by passing messages between applications or organizations, and ØMQ [26], an asynchronous messaging library that can run without a dedicated message broker. The eventual choice settled on Resque [27], a Redis-backed Ruby library specifically created for placing jobs into queues and processing them in the background.

Resque implements queuing using Redis Lists [28], which are lists of strings sorted by insertion order. A job consists of its name and parameters (the *payload*) that are serialized as a JSON string and stored in a Redis List corresponding to a specific queue. Jobs are performed by workers, which are separate Ruby processes configured to reserve jobs from specific queues. Although the main library includes only Ruby bindings for workers, the queuing system is language-agnostic and worker implementations exist for several other languages, including C, C#, Java and Python.

Adding and removing (pushing and popping) jobs to a queue has O(1) time complexity in Redis, meaning that these actions happen in constant time, independent of the size of a list. A “blocking pop” command is available, which effectively allows multiple workers to wait behind a single queue in a “first come, first served” manner. A fallback solution is also available where workers poll Redis queues every N seconds instead of blocking.

An overview of the queuing system is shown as a sequence diagram in Figure 4 and as a flow diagram in Figure 5. A successful flow of a single request is as follows: A request for capturing a URL is saved in the PostgreSQL database and to a dispatch queue in the Redis key-value store. A worker removes the job from the queue and distributes work to the required browser queues, which are monitored by corresponding Capture Workers. Capture Workers upload their captures to a central storage location and queue their results for processing by a Capture Processing Worker, which updates the database with the location of finished results. This worker also detects whether the baseline capture has finished, in which case it will queue subsequent captures for comparison. A Comparison Worker executes the visual comparison algorithm on captures it is given as parameters and queues its results for processing by a Comparison Processing Worker, which is responsible for updating the state of the request in the database and notifying the user.

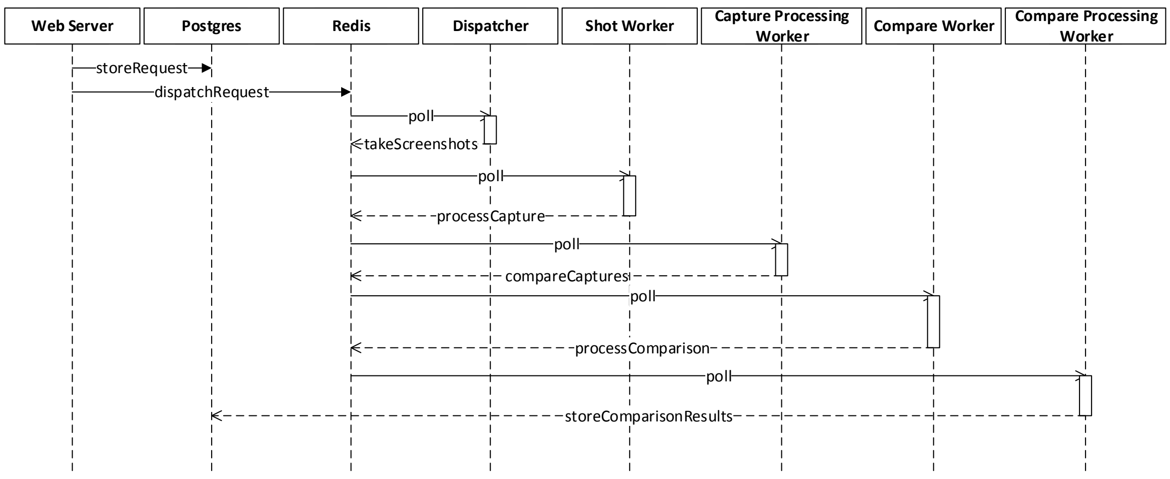


FIGURE 4 A SEQUENCE DIAGRAM OF THE CAPTURE SYSTEM'S WORKFLOW

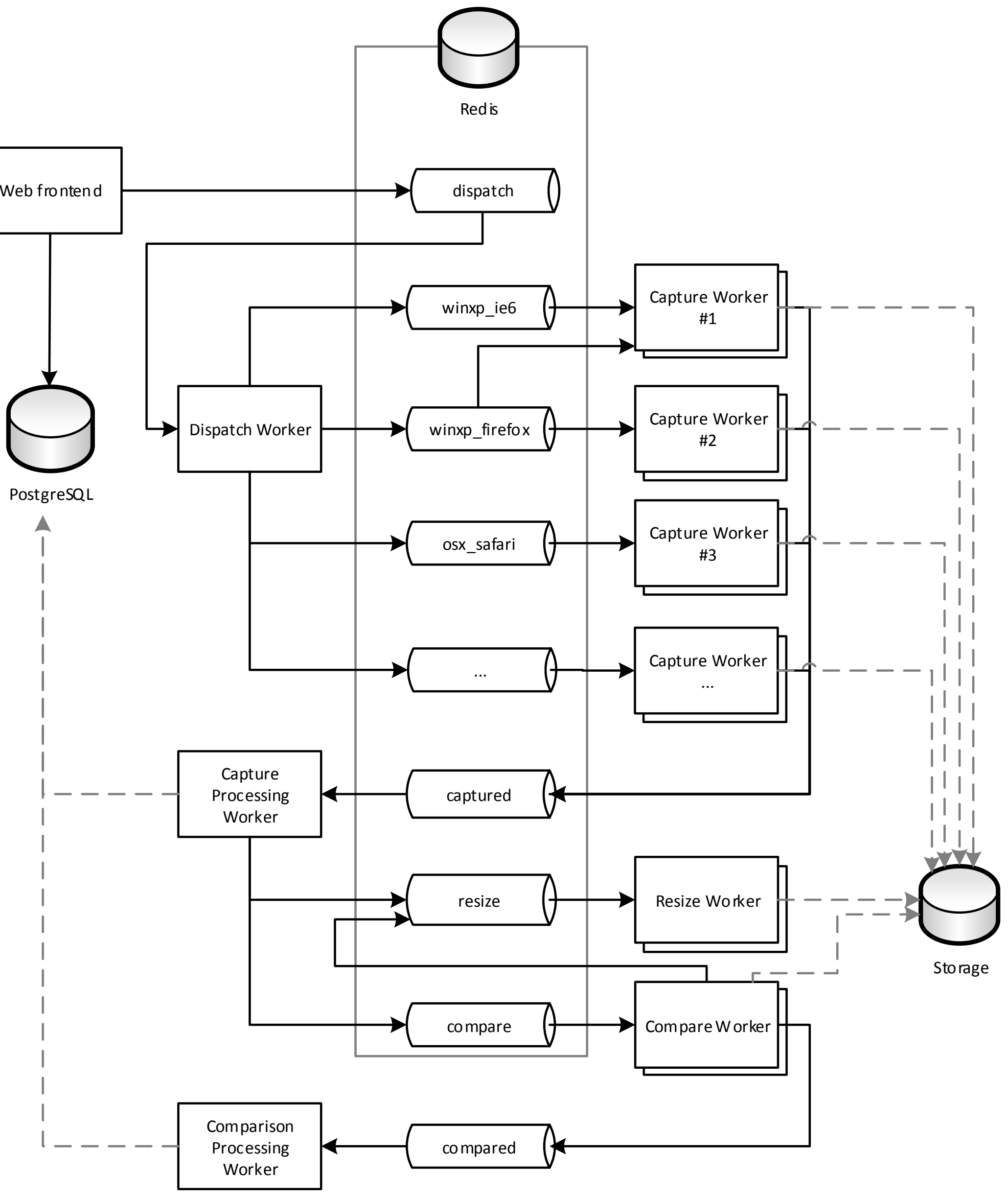


FIGURE 5 A DIAGRAM OF QUEUES AND PROCESSES IN THE SCALABLE CAPTURE SOLUTION.

Workers

Several workers were created to make parts of the capture system asynchronous and easily scalable. In addition to Shot Workers, the cross-platform processes that automate and capture web pages, there are also separate workers for dispatching a new request, processing finished captures, comparing captures against a baseline, processing the results of a comparison and creating thumbnails of captures for use in the frontend. Excluding shot workers, all of the aforementioned workers are configured to run using multiple threads, which allows for more concurrency and more efficient usage of the available hardware resources.

The workflow of a capture job (TakeScreenshots) is as follows: a worker removes the job from the queue and opens the required browser. If there are no local crop settings for this browser, the crop is recalibrated and stored in file. The browser then navigates to the given URL and waits for the document to complete loading. The browser window is maximized and a screenshot of the desktop is captured. The full-page capture follows, either by scrolling or resizing the document using measurements retrieved from the JavaScript DOM. The browser is closed and the resulting images and logs are copied or uploaded to the central storage. A ProcessCapture job is then queued with the created artefacts as parameters and finally, local artefacts are removed.

As there are many steps that can fail, either with browser automation, screen capture or uploading results, exception and timeout handling logic has been added that will requeue a job a certain number of times in case of apparent failure.

Resque workers can be configured to listen to multiple queues, which means that a single worker can perform captures of multiple browsers. This allowed the Browserbite product to offer 15 distinct configurations by only setting up 9 virtual machines. For example, in a real-life scenario the Browserbite capture system has three machines running on Windows 7 and each one performs captures of different versions of Internet Explorer (7, 8 and 9), in addition to a shared version of Google Chrome and Mozilla Firefox.

Web Browser Automation

Several software solutions exist to automate either specific or multiple browsers, including Sahi [30], WatiR [31] and Selenium WebDriver [32].

The solution described in this thesis uses Selenium WebDriver for automating web browsers. WebDriver is based on a client-server architecture communicating via JSON messages, defined in the WebDriver wire protocol [33]. Browser-specific drivers are accessed from client libraries by using their RESTful [34] web service over HTTP. A

RESTful web service can be described as a collection of resources that is hypertext driven and that has a base URI, a defined set of supported HTTP methods (e.g. GET, POST, PUT, DELETE) and a supported media type, e.g. JSON.

As of May 2013, Selenium WebDriver supports most browser families: Internet Explorer, Mozilla Firefox, Google Chrome, Opera, Mac Safari as well as simulated support for Android and iOS devices, as is explained further in this thesis.

At the time of writing the capture system, support for Mac Safari was missing from WebDriver. The WatiR project, however, had a working solution for automating Safari via AppleScript [35], a proprietary scripting language for Macintosh computers. A wrapper server was therefore created by the author, which translates incoming wire protocol requests into WatiR commands. Even with the added overhead of the wrapper setup, the performance of the Safari capture worker on the physical Mac Mini hardware has proved to be better than capture workers in the virtual environment.

Webpage capture

Capturing the screen

Ruby unfortunately lacks an API for screen capture. In the first version of shot workers the author therefore used JRuby [36], an alternate implementation of Ruby that runs on the Java Virtual Machine (JVM). This made it possible to use the `java.awt.Robot` class, which generates native system input events to manipulate the browser and creates screen captures of rectangles on the screen. As the Java API is cross-platform, this capture method worked on both Windows and Mac OS workers.

This method performed sufficiently well on physical machines, but performance issues were observed on virtual machines with shared CPU and memory resources as captures for long webpages would take several minutes to complete. For Windows desktops, a better performing solution was therefore developed using Windows GDI+, an API that includes functions used for graphics and formatted text on both video displays and printers. In particular, this API includes the BitBlt function (short for bit-level block transfer) that *performs a bit-block transfer of the colour data corresponding to a rectangle of pixels from the specified source device context into a destination device* *context* [37]. In the described capture system, the source device context is either the browser window or the browser's child window (the viewport).

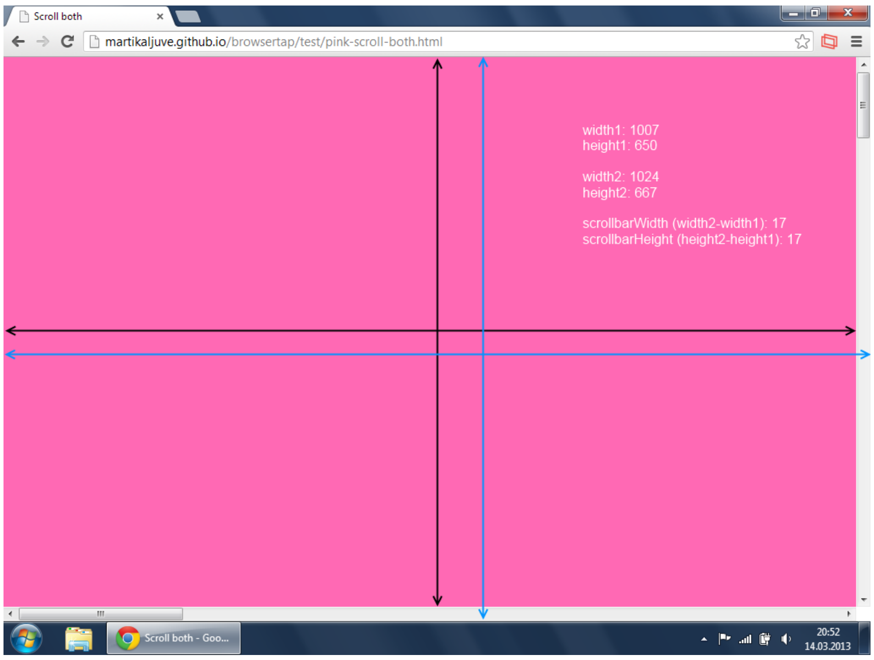
Appendix 2 shows the use of BitBlt to capture browsers and their child windows on Microsoft Windows, written in C++ as a Ruby extension.

Cross-Platform Capture Method - Scrolling

The first and most obvious solution to capture a full web document from a browser's viewport was to capture it gradually. This method entails scrolling to every part of the document horizontally and vertically, capturing the viewport into numbered files after every scroll event, and finally combining the captures into a single full-page image.

This method starts by querying and storing the current scroll position. The method for getting the scroll position can be seen in Appendix 1. The viewport is captured, the document is scrolled horizontally by the width of the viewport and the viewport is captured again. The new scroll position is then queried and compared with the old scroll position. If the amount moved is smaller than the width of the viewport, the page has finished scrolling to the right edge. The page is then scrolled back to the left edge and a vertical scroll is performed by the height of the viewport using the same rules. When the page has been scrolled through, the captured images are combined into a single full-page image using the command-line interface of a cross-platform montage utility in ImageMagick, an open source software suite for editing and displaying images [38].

As described previously, the full-page capture logic only captures the area of the desktop containing the browser viewport. To find the coordinates of this area, a calibration is performed for each browser on the computer by navigating to two bright-coloured pages, one with forced scrollbars and one without rules for scrolling. In both cases, a screenshot of the desktop is captured. Starting from the center of each image, the top, left, right and bottom bounds of the viewport are found. These measurements are then combined to get the viewport bounds and the scrollbar dimensions and cached in a local file for future requests. An illustration of these measurements is shown in Figure 6.


Scroll both
martikaljuve.github.io/browsertap/test/pink-scroll-both.html
Scroll both - Goo...
20:52
14.03.2013

```
width1: 1007
height1: 650

width2: 1024
height2: 667

scrollbarWidth (width2-width1): 17
scrollbarHeight (height2-height1): 17
```

FIGURE 6 A SCREENSHOT OF THE BROWSER VIEWPORT MEASUREMENT PAGE WITH ADDED ANNOTATIONS.

The scrolling capture solution works sufficiently well on static webpages, but has some drawbacks on more dynamic pages in its current state. For example, several webpages tested on the Browserbite platform include a navigation element or advertisement that is fixed to the browser viewport. This results in full-page captures where the fixed element is repeated as many times on the final image as the document has been scrolled. An additional, albeit minor flaw of the scrolling solution appears when a page has dynamically loaded content at the bottom of the page, in which case the algorithm might scroll the page an uncertain amount and create overlapping areas on the full-page capture.

Capture Method for Windows – Resizing

An unusual feature of the Windows operating system is that a window can be positioned or resized so that its dimensions exceed the bounds of the desktop. This is not possible by regular user interaction using a mouse and keyboard, but can be done programmatically using the Windows API `SetWindowPos` [39] method and setting a `SWP_NOSENDCHANGING` flag that prevents the window from receiving a specific message about its size and position changing. Using this method, it is possible to make a browser window larger than the desktop so that the viewport is as large as the entire web document, which can then be captured all at once.

The window resizing method requires a handle to a window as its first parameter. In Windows, every window has a unique handle, represented by either a 32-bit or 64-bit signed integer based on the operating system's version. As Selenium WebDriver does not expose the window handle of a browser, the solution described in this thesis has implemented a custom solution for finding the correct window. After a web document has been opened in the browser, the title of the document is changed via JavaScript to a unique name consisting of a prefix and a random number, e.g. browserbite-1409. A function then iterates over the handles of all open windows using the Windows API `EnumWindows` method, checking the title of each window for a matching title.

To determine how large the viewport must be to fit the whole document, attribute values from the Document Object Model are queried via JavaScript. For example, the document's height is set to the largest value of these attributes:

- document.body.scrollHeight
- document.body.offsetHeight
- document.documentElement.scrollHeight
- document.documentElement.offsetHeight
- document.documentElement.clientHeight

The corresponding attributes for width are used to get the document's desired width. The dimensions reported by JavaScript are then combined with the measurements of the browser window (found via calibration as described in the scrolling method description) to get the full required size of the window. The window is then resized, captured and restored to its original size.

The method of capturing a full web document by resizing is used in the Browserbite product for all browsers running on Windows XP, Vista, Windows 7 and 8.

Capture Methods for Mobile Devices

Android

Support for automating web documents in Android devices was added using the AndroidDriver package included in Selenium WebDriver. Instead of automating the native browser of a device, this driver is implemented as a separate Android application consisting of an HTTP server for translating incoming wire protocol commands and a

WebView [40] object where the desired webpage is opened and automated. It can be run on both physical devices and emulators, but only the latter are used in the capture system described in this thesis. Though replicating the real browser of an operating system was desired, the solution used by AndroidDriver was deemed suitable, as the native browser and WebView used the same WebKit rendering engine and would therefore display webpages identically.

The WebView object includes a method for capturing the full document without scrolling, but since the driver works by encoding the image in Base64 and sending it as a JSON response, on large pages the driver would often crash or reach a timeout imposed by the HTTP protocol. To prevent this, the AndroidDriver was modified in the described capture solution to instead save the captured document to a file on the device's storage and copy it using a command line tool included in the Android Software Development Kit (SDK).

As of May 2013, developers of Chromium, the open source project from which Google Chrome is derived from, were working on ChromeDriver2, a new driver based on the WebKit Remote Debugging Protocol [41] that is included in both desktop and mobile releases of Chrome. When this driver is released, additional Android shot workers can be added to the Browserbite product, using the same control logic as the existing workers.

Apple iOS

Similarly to Android support, the Selenium project includes IPhoneDriver for automating webpages in iOS devices. This driver does not automate the built-in Mobile Safari browser of an iOS device - instead, it is implemented as a native iOS application that displays a fullscreen UIWebView object. As the same UIWebView is used by the built-in Safari browser, there are no differences when comparing a webpage rendered by IPhoneDriver and Mobile Safari.

There is currently no known method for capturing the whole contents of a UIWebView object, so full document capture on the iOS shot worker was implemented using the same scrolling method outlined previously.

As of May 2013, an open-source project named "ios-driver" [42] has been in development that enables automation of both native and web applications on iOS devices using the WebDriver Wire protocol. Internally, ios-driver uses Apple's UIAutomation framework [43] to control native applications, while webpages are opened in the Mobile Safari browser and controlled using the Remote Debugging Protocol built into WebKit.

As one of the requirements of the document capture solution is to render webpages in real browsers and operating systems, the ios-driver implementation should be preferred over the Selenium IPhoneDriver implementation. The ios-driver project will therefore be used in the next version of the capture system’s iOS shot worker.

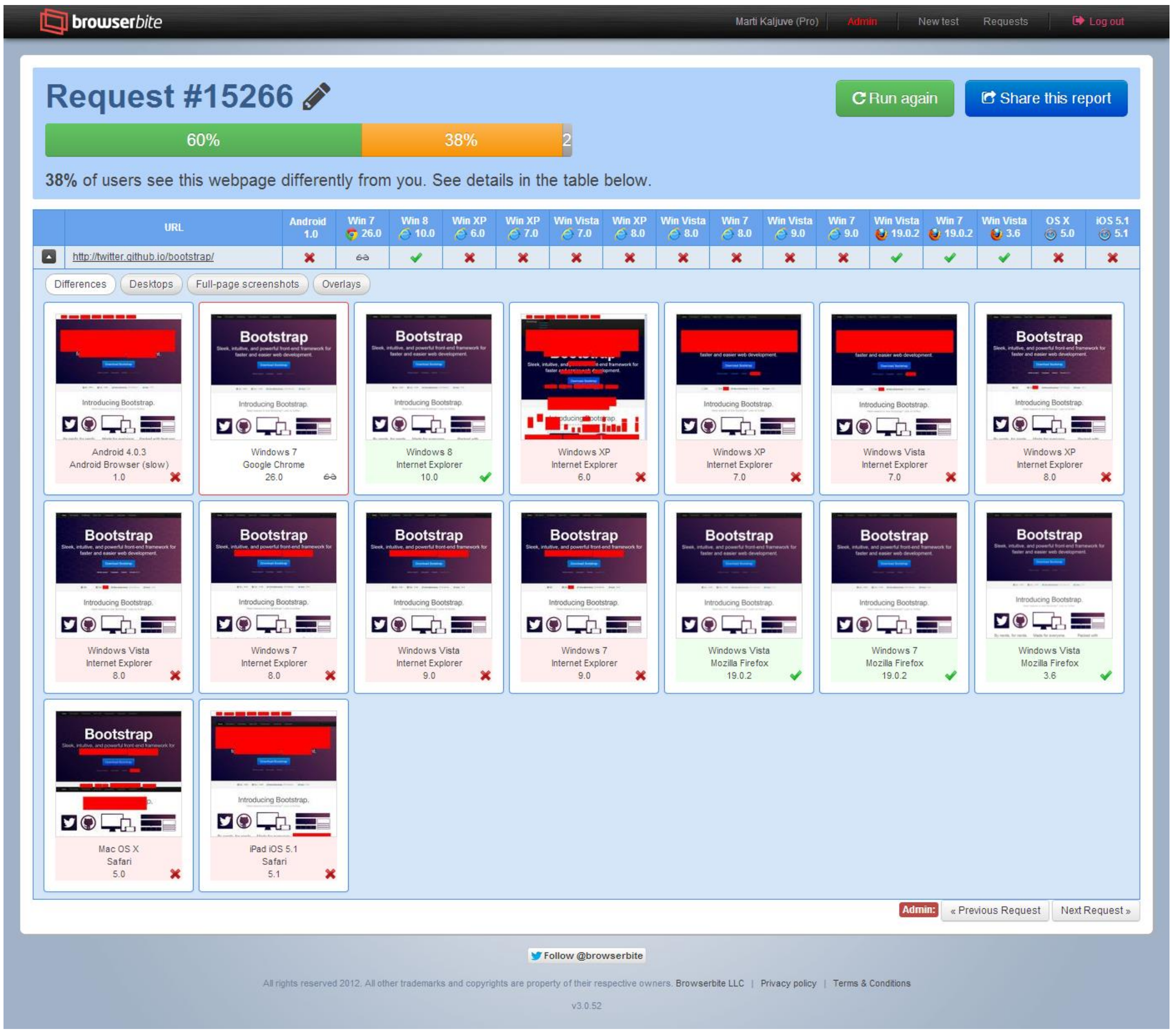
Frontend

The frontend for the capture system was developed using Ruby on Rails [44], a popular open source web application framework that runs on the Ruby programming language. The Rails framework includes components needed to create database-backed web applications according to the Model-View-Controller (MVC) pattern. The MVC pattern divides an application into three layers:

- Model layer, encapsulating the business logic and domain model of the application;
- View layer, consisting of "templates" that provide representations of the application's resources;
- Controller layer, handling incoming HTTP requests and responding with a rendered template from the View layer.

A model in Rails is typically a class using the Active Record [45] pattern, which maps a row in a database table to a Ruby object and can be embellished with additional business logic methods. A view in Rails is usually an HTML file with embedded Ruby code, but depending on the HTTP request, a controller can output other formats, for example XML, JSON or PDF.

When a URL is submitted for capturing from the frontend, the user is redirected to an overview page showing placeholders for all queued captures. As captures are finished by shot workers, these placeholders are automatically replaced with thumbnails of the resulting images from different configurations. This is accomplished by having the rendered overview page periodically poll for changes from the web server using Asynchronous JavaScript and XML [46] (AJAX), a popular technique to achieve asynchronous communication between a client browser and a web server. A screenshot of the web frontend used in the Browserbite product is shown in Figure 7.


Request #15266

60% 38% 2

38% of users see this webpage differently from you. See details in the table below.

URL	Android 1.0	Win 7 26.0	Win 8 10.0	Win XP 6.0	Win XP 7.0	Win Vista 7.0	Win XP 8.0	Win Vista 8.0	Win 7 8.0	Win Vista 9.0	Win 7 9.0	Win Vista 19.0.2	Win 7 19.0.2	Win Vista 3.6	OS X 5.0	iOS 5.1 5.1
http://twitter.github.io/bootstrap/	✗	⚯	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗

browserbite
Marti Kaljuve (Pro)
Admin
New test
Requests
Log out
Run again
Share this report
Differences
Desktops
Full-page screenshots
Overlays
Android 4.0.3 Android Browser (slow) 1.0
Windows 7 Google Chrome 26.0
Windows 8 Internet Explorer 10.0
Windows XP Internet Explorer 6.0
Windows XP Internet Explorer 7.0
Windows Vista Internet Explorer 7.0
Windows XP Internet Explorer 8.0
Windows Vista Internet Explorer 8.0
Windows 7 Internet Explorer 8.0
Windows Vista Internet Explorer 9.0
Windows 7 Internet Explorer 9.0
Windows Vista Mozilla Firefox 19.0.2
Windows 7 Mozilla Firefox 19.0.2
Windows Vista Mozilla Firefox 3.6
Mac OS X Safari 5.0
iPad iOS 5.1 Safari 5.1
Admin:
« Previous Request
Next Request »
Follow @browserbite
All rights reserved 2012. All other trademarks and copyrights are property of their respective owners. Browserbite LLC | Privacy policy | Terms & Conditions
v3.0.52

FIGURE 7 BROWSERBITE FRONTEND SHOWING 16 CAPTURED CONFIGURATIONS OF A REQUEST.

The frontend is served by Unicorn [47] workers running behind an nginx [48] reverse proxy server. Unicorn is an HTTP server for Ruby applications that takes advantage of features like forking found in Unix-like kernels to serve clients. Nginx is a popular open source web server and a reverse proxy that is focused on high performance and low memory usage. According to Netcraft statistics for May 2013 [49], nginx served or proxied 13.54% busiest sites in the world.

Webpage automation

A major supplementary feature of the capturing system is support for scripting user actions on a webpage before capturing. In addition to capturing the flow of visitors on a webpage, automation enables capturing web documents that are behind custom login forms, as well as capturing states of a web application where a state does not have a URL.

Recording

Due to the majority of Browserbite customers using Google Chrome, the recording solution has been implemented as a Chrome browser extension. However, as the Chrome-specific features were deliberately kept in separate service classes, a major part of the extension source code can be reused when implementing a similar solution for other browsers with JavaScript-based extensions, such as Mozilla Firefox, Opera and Safari.

The Chrome extension consists of a content script that runs in the context of a recorded webpage, an invisible background page that stores the recorded steps and a popup page showing the currently recorded steps (shown in Figure 9). An overview of how the recorded data from the webpage is transferred to shot workers is shown in Figure 8. When recording is started from the popup page, a content script is injected into the active webpage - this is JavaScript code that adds several event handlers, e.g. for clicks, form field changes and mouse hover events, to the document. When any of these actions are performed by the user, the handlers forward information about these events to the background page so that they can be replayed as steps in other browsers. The forwarded data includes unique locators for the event's target, along with its coordinates and changed value, where applicable.

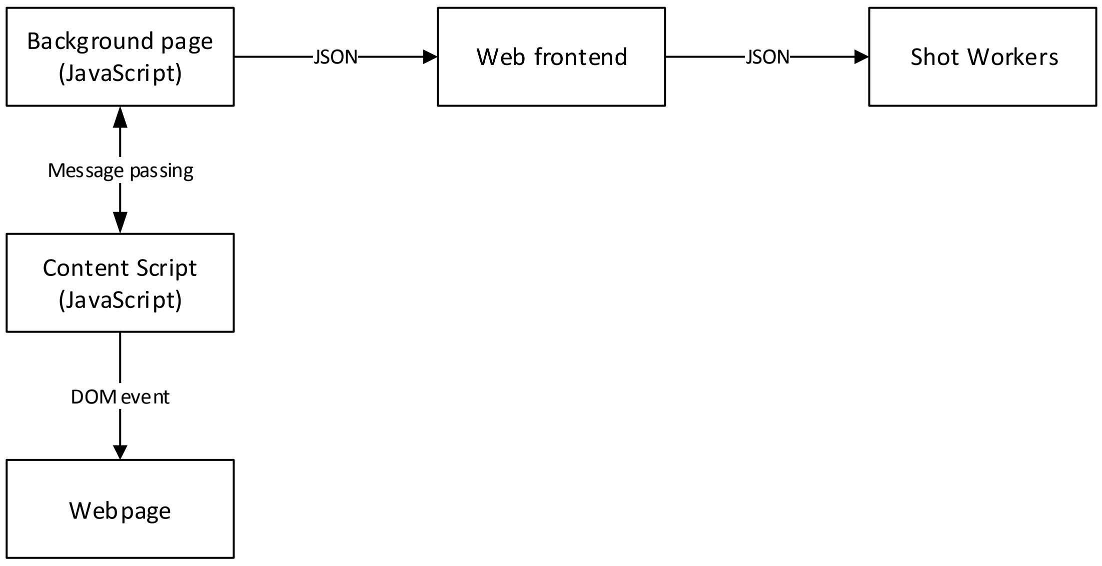
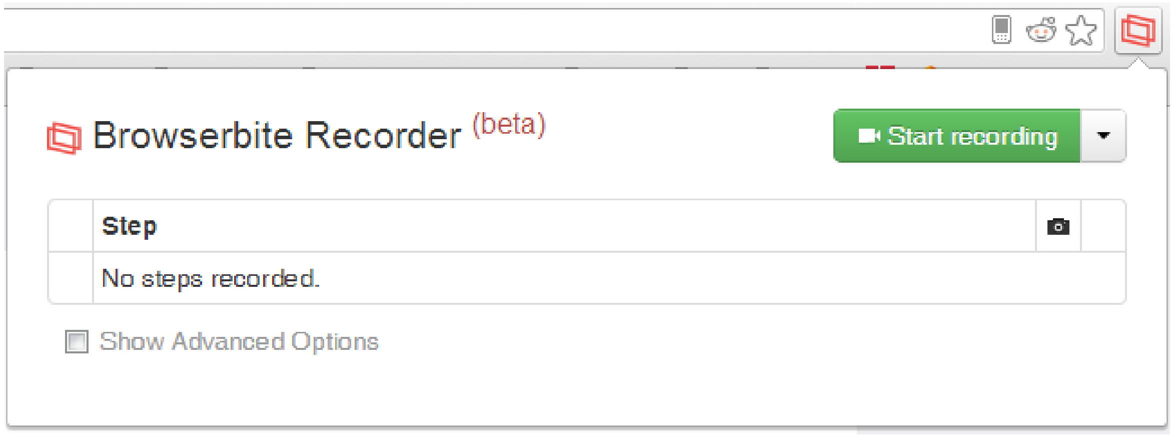
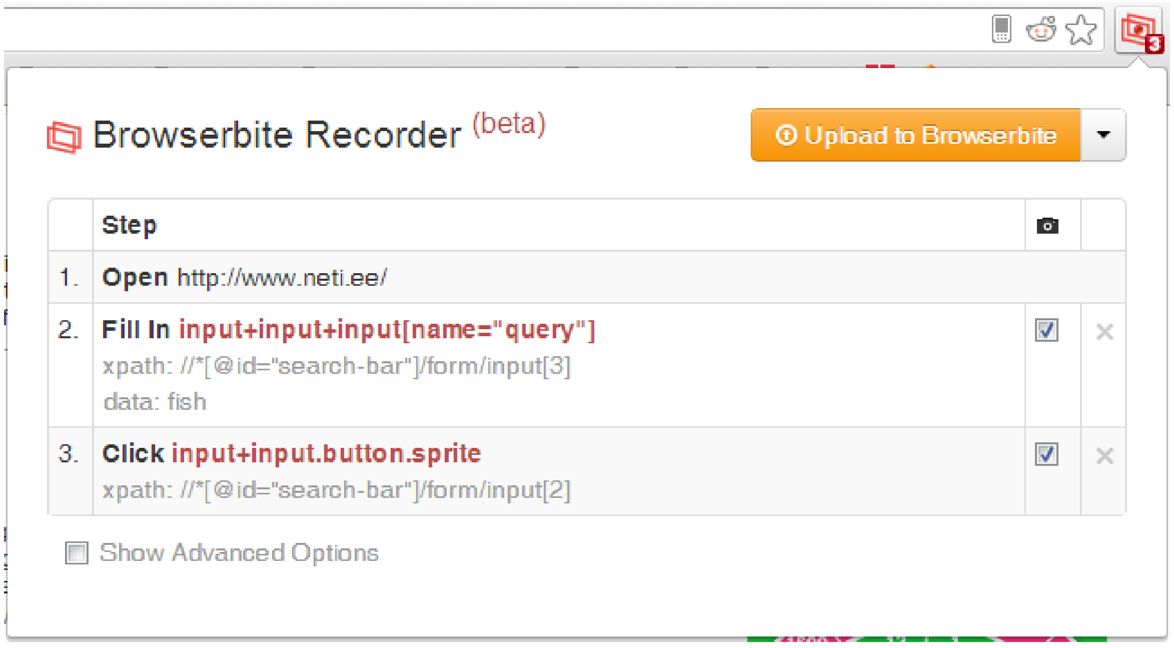


FIGURE 8 DIAGRAM OF DATA FLOW FROM THE RECORDER TO THE CAPTURE SYSTEM.

As these elements need to be found on multiple platforms and browsers with possibly different DOM-s, both unique CSS and XPath selectors are recorded. To find a unique CSS selector for an element, the DOM path of that element is traversed in reverse, starting from the element itself. For each element in the path, a selector is constructed by concatenating the element's tag name with the first attribute value that is either ID, NAME, CLASS, TYPE, ALT, TITLE or VALUE. This selector is then prefixed with a list of preceding siblings' tag names, separated by plus-signs. After every element traversal, the document is queried using the JavaScript document.querySelector() [50] method and if it returns the desired element, a unique CSS selector has been found. An example result is shown in Appendix 3. The method of using plus-signs to target sibling elements instead of more terse :nth-child() [51] CSS pseudo-selectors was chosen for compatibility with old browsers, in particular Internet Explorer 7 and 8.





1. **Open** http://www.neti.ee/
2. **Fill In** input+input+input[name="query"]
   xpath: //*[@id="search-bar"]/form/input[3]
   data: fish
3. **Click** input+input.button.sprite
   xpath: //*[@id="search-bar"]/form/input[2]

FIGURE 9 THE BROWSERBITE CHROME EXTENSION POPUP BEFORE AND AFTER RECORDING

Playback

Shot workers replay recorded steps using built-in methods in Selenium WebDriver. An element is first looked for using the CSS selector in the recording, using the XPath selector as a failback. If the element is found, the recorded action is performed on it, after which the document is measured and fully captured. Both finding an element and performing an action on it is retried up to 3 times to account for dynamically loaded elements that might not be immediately available after the page is first loaded or after the previously performed step.

Capture Processing jobs are created after every performed step, which allows the frontend to show finished steps a few seconds after they have been repeated by shot workers.

Cloud computing

As the need to scale grows, it may be more feasible to migrate parts or all of the project to an external cloud provider such as Amazon Elastic Compute Cloud. The main challenge with this decision is that most cloud providers only offer either Windows Server or Enterprise Linux virtual machines instead of desktop operating systems. Running desktop virtual machines inside the cloud provider's Windows Server virtual machines can result in severely degraded performance, but makes the system horizontally scalable.

Due to licencing constraints not all parts of the system can be migrated to a cloud provider. For instance, the Software License Agreement for Mac OS X [29] grants a license *to install, use and run up to two (2) additional copies or instances of the Apple Software within virtual operating system environments on each Mac Computer you own or control that is already running the Apple Software*. This means that OS X capture workers can only run inside Mac computers or virtual machines running on Mac Computers, which severely limits the available options of cloud providers.

Except for the legal constraint introduced by Apple Inc., all other parts of the system can be migrated. Since the shot workers are processing in the background asynchronously from users, no real threat is imposed by migrating parts of the capture system to cloud services in the future. An architecture that has running parts in both an external cloud and on private hardware can be defined as a hybrid cloud solution.

Chapter 5

Conclusion

It is rare to have a web page perform exactly the same in different browsers. Though most differences across platforms can be considered insignificant, for example text rendered in different fonts, page elements missing background gradients or rounded vs. rectangular corners on buttons, there are sometimes more severe differences. In worst cases, layout differences or bugs can result in lost revenue. Knowing this, a commercial offering by Browserbite was created that can capture full-page screenshots of web pages in a wide variety of web browsers on different platforms and can algorithmically find discrepancies between them.

This thesis focused on the implementation aspects of the cross-platform capture system in the product offered by Browserbite. First, a set of requirements were established for cross-browser document capturing. Existing web services and applications were compared against these requirements. It was determined that none of the existing solutions completely satisfied the previously set criteria.

A new solution architecture was created that enabled horizontal and vertical scaling. Established queuing solution was introduced to enable asynchronous job processing and separate front-end and back-end specific tasks. Methods for capturing a web-page in multiple platforms and browsers were then introduced, using operating system-specific functions where required to manipulate a browser window more efficiently and capture a full page document and the desktop.

A scalable architecture consisting of a web frontend, a relational database, a key-value store and distributed background workers was described that is already being successfully used in a production environment. The architecture enables to migrate nearly all of the subsystems into a cloud service provider to ease the deployment and other options.

The solution is in daily use by both corporate and freelance customers. It is commercially available at: http://browserbite.com.

Future work

As a commercial offering, new features are mainly put into work based on user feedback. The capture system has been considered fast and stable enough for production use with prospective loads, but several new features are planned in the near future:

- Supporting desktop resolutions other than 1024x768, either by programmatically changing the screen resolution before captures or by having capture workers running on computers with different resolutions.
- Banner detection and coverage: pages that have dynamic banner ads or animations will often produce irrelevant comparison results. The solution should therefore try to detect the dynamic parts of a web page and cover these parts on each capture before sending them for comparison. Preliminary tests to detect these changes by capturing the page twice with a delay or page refresh and finding the pixel-by-pixel differences have yielded positive results.
- Support for capturing different browsers on Android emulators or physical devices, as currently only the WebKit based browser is used.
- Support for testing private webpages and intranet sites behind firewalls by establishing a reverse network tunnel to the customer's computer.
- Support for capturing desktop browsers on popular Linux distributions.
- Creating a REST API for the capture solution which can be used outside the web frontend. This would enable more seamless integration into current workflows of the system's users.

Most of the tools and libraries used in the implemented cross-browser capture system are open-source. The capturing solution has advantages over many existing services, but unlike the visual comparison algorithm, given time, it is not difficult to reproduce. The capturing solution could therefore be considered for release as an open-source project to enable outside contributions that improve and extend its capabilities.

Mitmeplatvormiline veebidokumentide pildistamise lahendus

Magistritöö (30 EAP)

Marti Kaljuve

Resümee

Veebilehte kuvatakse harva täpselt samasugusena erinevates brauseri ja operatsioonisüsteemi kombinatsioonides. Sellel on mitmeid põhjuseid: veebistandardite tõlgendamine brauseri poolt, brauseri visualiseerimismootor, operatsioonisüsteemi vaikefondid, brauserisse installeeritud pistikprogrammid, ekraani eraldusvõime jms. Nende erinevuste tähelepanuta jätmine võib tekitada probleeme veebilehe kujunduses, mille tagajärjeks on klientide kaotamine.

Veebidisaineritele võib tunduda veebilehtede testimine mitmes brauseris tavapärase praktikana, et leida brauseritevahelised kujunduse probleemid. Katsed näitavad, et visuaalsete erinevuste käsitsi leidmine on tülikas ja kohmakas ülesanne. Seda teades on meie meeskonna liige loonud algoritmi, mis on osutunud inimestega võrreldes märkimisväärselt kiiremaks ja täpsemaks kujunduses vigade leidmisel. Algoritm töötab selliselt, et veebilehest tehtud aluspilti (tarkvara testimise mõistes oraaklit) võrreldakse samast veebilehest teiste brauseritega tehtud piltidega, leides nendes paigutuse erinevusi, mida ka inimsilm arvestaks väärana.

Käesolev töö keskendub probleemile, kuidas eelnevalt mainitud algoritmile sisendit luua. Töö annab valikulise ülevaate olemasolevatest lahendustest ja teenustest, mis tagastavad veebilehe sisu pildi kujul, ning võimalusel mõõdab nende jõudlust. Tuvastatakse nimekiri nõuetest, mis on vajalikud mitmeplatvormilise veebidokumentide pildistamise lahenduse kommertsialiseerimiseks. Seejärel tutvustab töö kiiret ja mitmeplatvormilist meetodit veebilehe täispikkuses pildistamiseks ning annab ülevaate skaleeritava arhitektuuriga veebiteenusest, mis pildistab veebilehti virtuaalsetes ja füüsilistes masinates ning erinevates brauserites ja operatsioonisüsteemides.

Bibliography

[1] StatCounter, “Top 12 Browser Versions from Apr 2012 to Apr 2013,” 2013. [Online]. Available: http://gs.statcounter.com/#browser_version-ww-monthly-201204-201304. [Accessed April 2013].

[2] W3C, “HTML & CSS - W3C,” May 2013. [Online]. Available: http://www.w3.org/standards/webdesign/htmlcss. [Accessed May 2013].

[3] Microsoft, “Internet Explorer 6 Countdown | Death to IE 6 | IE6 Countdown,” 2011. [Online]. Available: http://www.ie6countdown.com/. [Accessed April 2013].

[4] Mogoterra, Inc., “Mogotest,” 2013. [Online]. Available: http://mogotest.com/. [Accessed May 2013].

[5] BrowserStack, “Cross Browser Testing Tool. 200+ Browsers, Mobile, Real IE.,” 2013. [Online]. Available: http://www.browserstack.com/. [Accessed May 2013].

[6] Browsershots, “Check Browser Compatibility, Cross Platform Browser Test,” 2013. [Online]. Available: http://browsershots.org/. [Accessed May 2013].

[7] NetMarketShare, “Screen Resolutions - Report,” May 2011. [Online]. Available: http://www.netmarketshare.com/report.aspx?qprid=17&qptimeframe=M&qpsp=148&qpch=350&qpmr=100&qpdt=1&qpct=3&qpcid=fw79667&qpf=1. [Accessed January 2013].

[8] Adobe Systems, “Millward Brown survey,” July 2011. [Online]. Available: http://www.adobe.com/products/flashplatformruntimes/statistics.html. [Accessed

April 2013].

[9] Amazon, “Amazon Elastic Compute Cloud (Amazon EC2),” 2013. [Online]. Available: http://aws.amazon.com/ec2/. [Accessed May 2013].

[10] Microsoft, “Expression Web SuperPreview,” 17 March 2011. [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=2020. [Accessed March 2013].

[11] Microsoft, “Microsoft Expression Web SuperPreview - Expression Web team blog,” March 2009. [Online]. Available: http://blogs.msdn.com/b/xweb/archive/2009/03/18/microsoft-expression-web-superpreview-for-windows-internet-explorer.aspx. [Accessed December 2012].

[12] Browsershots.org, "Check Browser Compatibility, Cross Platform Browser Test," [Online]. Available: http://browsershots.org/. [Accessed April 2013].

[13] browsershots, “browsershots - Test your web design in different browsers - Google Project Hosting,” 23 October 2009. [Online]. Available: https://code.google.com/p/browsershots/source/browse/#svn%2Ftrunk. [Accessed March 2013].

[14] “PPM Format Specification,” 3 October 2003. [Online]. Available: http://netpbm.sourceforge.net/doc/ppm.html. [Accessed May 2013].

[15] TestingBot, “Selenium Testing in the cloud - Run your cross browser tests in our online Selenium Grid,” 2013. [Online]. Available: https://testingbot.com/. [Accessed May 2013].

[16] Deep Shift Labs, “Nerrvana - easy Selenium testing in the cloud,” 2013. [Online]. Available: http://www.nerrvana.com/. [Accessed May 2013].

[17] Browserling Inc, “browserling - interactive cross-browser testing,” 2013. [Online]. Available: https://browserling.com/. [Accessed May 2013].

[18] J. Webber, *Guerilla SOA,* 2007.

[19] The PostgreSQL Global Development Group, “PostgreSQL: The world's most advanced open source database,” May 2013. [Online]. Available: http://www.postgresql.org/. [Accessed May 2013].

[20] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1992.

[21] Redis, “Redis,” May 2013. [Online]. Available: http://redis.io/. [Accessed May 2013].

[22] D. Spezia, “How fast is Redis? - Redis,” 29 July 2012. [Online]. Available: http://redis.io/topics/benchmarks. [Accessed 03 04 2013].

[23] “Ruby Programming Language,” May 2013. [Online]. Available: http://www.ruby-lang.org/en/. [Accessed May 2013].

[24] Squid, “squid : Optimising Web Delivery,” 2013. [Online]. Available: http://www.squid-cache.org/. [Accessed March 2013].

[25] OASIS, “Home | AMQP,” May 2013. [Online]. Available: http://www.amqp.org/. [Accessed May 2013].

[26] iMatix, “The Intelligent Transport Layer - zeromq,” 2013. [Online]. Available: http://www.zeromq.org/. [Accessed May 2013].

[27] Resque, “Resque: the rock-solid job queue,” GitHub, May 2013. [Online]. Available: http://resquework.org/. [Accessed May 2013].

[28] Redis, “Data types - Redis,” [Online]. Available: http://redis.io/topics/data-types. [Accessed May 2013].

[29] Apple Inc, “Software License Agreement For Mac OS X,” 01 07 2012. [Online]. Available: http://www.apple.com/legal/sla/docs/OSX108.pdf. [Accessed 03 04 2013].

[30] Tyto Software Pvt. Ltd., “Sahi Web Test Automation Tool,” 2013. [Online]. Available: http://sahi.co.in/. [Accessed March 2013].

[31] B. Pettichord and P. Rogers, “Watir.com | Web Application Testing in Ruby,” 2013. [Online]. Available: http://watir.com/. [Accessed March 2013].

[32] Selenium, “Selenium - Web Browser Automation,” May 2013. [Online]. Available: http://www.seleniumhq.org/. [Accessed May 2013].

[33] Selenium, “JsonWireProtocol - selenium - A description of the protocol used by WebDriver to communicate with remote instances,” April 2013. [Online].

Available: http://code.google.com/p/selenium/wiki/JsonWireProtocol. [Accessed April 2013].

[34] R. T. Fielding, "REST APIs must be hypertext-driven," 20 October 2008. [Online]. Available: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[35] Apple Inc., "AppleScript Overview: Introduction to AppleScript Overview," 31 October 2007. [Online]. Available: https://developer.apple.com/library/mac/#documentation/AppleScript/Conceptual/AppleScriptX/AppleScriptX.html. [Accessed March 2013].

[36] JRuby, "Home - JRuby.org," May 2013. [Online]. Available: http://www.jruby.org/. [Accessed May 2013].

[37] C. Petzold, "Programming Windows," Microsoft Press, 1998, p. 648.

[38] ImageMagick Studio LLC, "ImageMagick: Convert, Edit, Or Compose Bitmap Images," May 2013. [Online]. Available: http://www.imagemagick.org/script/index.php. [Accessed May 2013].

[39] Microsoft, "SetWindowPos function," February 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms633545(v=vs.85).aspx. [Accessed May 2013].

[40] Google Inc., "WebView | Android Developers," May 2013. [Online]. Available: http://developer.android.com/reference/android/webkit/WebView.html. [Accessed May 2013].

[41] Google Inc, "Remote Debugging Protocol v1.0 - Chrome DevTools - Google Developers," 7 May 2013. [Online]. Available: https://developers.google.com/chrome-developer-tools/docs/protocol/1.0/. [Accessed 19 May 2013].

[42] F. Reynaud, "ios-driver," May 2013. [Online]. Available: http://ios-driver.github.io/ios-driver/. [Accessed May 2013].

[43] Apple Inc., "UI Automation JavaScript Reference," 9 September 2012. [Online]. Available: http://developer.apple.com/library/ios/#documentation/DeveloperTools/Reference

/UIAutomationRef/_index.html. [Accessed May 2013].

[44] D. H. Hansson, “Ruby On Rails,” May 2013. [Online]. Available: http://rubyonrails.org/. [Accessed May 2013].

[45] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.

[46] J. J. Garrett, “Ajax: A New Approach to Web Applications,” 18 February 2005. [Online]. Available: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications. [Accessed March 2013].

[47] E. Wong, “Unicorn: Rack HTTP server for fast clients and Unix,” [Online]. Available: http://unicorn.bogomips.org/. [Accessed May 2013].

[48] nginx, “nginx,” [Online]. Available: http://nginx.org/. [Accessed May 2013].

[49] Netcraft Ltd., “May 2013 Web Server Survey,” May 2013. [Online]. Available: http://news.netcraft.com/archives/2013/05/03/may-2013-web-server-survey.html. [Accessed May 2013].

[50] W3C, “Selectors API Level 1,” 21 February 2013. [Online]. Available: http://www.w3.org/TR/selectors-api/. [Accessed May 2013].

[51] W3C, “Selectors Level 3,” 29 September 2011. [Online]. Available: http://www.w3.org/TR/selectors/#nth-child-pseudo. [Accessed May 2013].

Appendices

APPENDIX 1. FINDING THE SCROLL POSITION

```
var scrOfX = 0, scrOfY = 0;

if (typeof (window.pageYOffset) === 'number') {

      //Netscape compliant

      scrOfY = window.pageYOffset;

      scrOfX = window.pageXOffset;

} else if (document.body && (document.body.scrollLeft ||
document.body.scrollTop)) {

      //DOM compliant

      scrOfY = document.body.scrollTop;

      scrOfX = document.body.scrollLeft;

} else if (document.documentElement &&
(document.documentElement.scrollLeft ||
document.documentElement.scrollTop)) {

      //IE6 standards compliant mode

      scrOfY = document.documentElement.scrollTop;

      scrOfX = document.documentElement.scrollLeft;

}

return { 'x': scrOfX, 'y': scrOfY };
```

```
VALUE capture_and_crop(VALUE self, VALUE hwndInt, VALUE
filenameValue, VALUE leftValue, VALUE topValue, VALUE
rightValue, VALUE bottomValue, VALUE useBitBlt) {
	int x, y, width, height;
	HWND controlHwnd = tohwnd(hwndInt);
	RECT controlRect;
	HDC controlDC;
	HDC compatibleDC;
	HBITMAP compatibleBitmap;

	if(!controlHwnd) {
		printf("Invalid handle value: %d\n", controlHwnd);
		return filenameValue;
	}

	int left = FIX2INT(leftValue);
	int top = FIX2INT(topValue);
	int right = FIX2INT(rightValue);
	int bottom = FIX2INT(bottomValue);

	// Initialize GDI+.
	Gdiplus::GdiplusStartupInput gdiplusStartupInput;
	ULONG_PTR gdiplusToken;
	GdiplusStartup(&gdiplusToken, &gdiplusStartupInput, NULL);

	CLSID encoderClsid;
	Gdiplus::Status stat;

	GetWindowRect(controlHwnd, &controlRect);

	x = controlRect.left;
	y = controlRect.top;
	width = controlRect.right - controlRect.left;
	height = controlRect.bottom - controlRect.top;

	controlDC = GetDC(controlHwnd);
	compatibleDC = CreateCompatibleDC(controlDC);

	int finalWidth = useBitBlt ? (width - left+right) : width;
	int finalHeight = useBitBlt ? (height - top+bottom) :
height;

	compatibleBitmap = CreateCompatibleBitmap(controlDC,
finalWidth, finalHeight);
	SelectObject(compatibleDC, compatibleBitmap);

	if (useBitBlt == true) {
		BitBlt(compatibleDC, 0, 0, finalWidth, finalHeight,
controlDC, left, top, SRCCOPY);
	}
	else {
		PrintWindow(controlHwnd, compatibleDC, 0);
	}
	Gdiplus::Bitmap* image =
Gdiplus::Bitmap::FromHBITMAP(compatibleBitmap, NULL);

	if (useBitBlt == false && (left > 0 || top > 0 || right > 0
|| bottom > 0)) {
		Gdiplus::Bitmap* cropped = image->Clone(
			left, top, width - (left + right), height - (top
+ bottom), image->GetPixelFormat());
		delete image;
		image = cropped;
		printf("Bitmap::Cloned -> %p\n", cropped);
	}

	CLSID clsid;
	wstring filename = StringValuePtr(filenameValue);
	if(GetEncoderClsid(L"image/png", &clsid) == -1) {
		printf("Failed to get encoder clsid\n");
	}
	else if(stat = image->Save(filename, &clsid, NULL)) {
		printf("Capture failed: error %d\n", stat);
	}
	else {
		wprintf(L"Capture: %s\n", filename.c_str());
	}

	delete image;
	DeleteObject(compatibleBitmap);

	DeleteDC(compatibleDC);
	ReleaseDC(controlHwnd, controlDC);

	Gdiplus::GdiplusShutdown(gdiplusToken);

	return filenameValue;
}
```

APPENDIX 3 FINDING UNIQUE CSS SELECTOR FOR DOM ELEMENT

```
<html>
      <head>
            <title>Test page</title>
      </head>
      <body>
            <ul>
                  <li><a href="#">A</a></li>
                  <li><a href="#">B</a></li>
                  <li><a href="#">C</a></li>
            </ul>
      </body>
</html>

CSS selector for list item "C":
li+li+li > a
```