Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

What's New?

Information and Entertainment Application for Android and iPhone

Semester Thesis
Dominik Bucher, D-ITET
dobucher@ee.ethz.ch
January 2012

Distributed Computing Group, ETH Zurich, 8092 Zurich

Supervisors Samuel Welten and Mihai Calin
Professor Dr. Roger Wattenhofer

Abstract

In this thesis a script language is developed that allows processing of arbitrary HTML documents. The processing is done to extract important elements from the documents. These elements include headlines, paragraphs and images. In the process, advertisements, specific design elements and other unwanted parts are discarded. This allows extraction of specific content from any regular web site. The script language is finally used within a news application that crawls news sites, extracts important parts of them and displays the articles. For the whole process no content from content providers has to be stored on the server of the application developer. The only things that are stored on the server are links pointing to the respective html pages and files on about how to process this html. This can also be beneficial for copyright issues as the whole content extraction is done on the end-user device.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Some years ago we used to read our daily news by having a look at the newspaper every morning. Most of us haven't changed this habit, but the newspapers themselves nowadays often look quite different. With the trend that everyone carries a computer wherever he goes, with those devices getting increasingly powerful and with them being connected to the internet all the time, more and more news are being distributed over this new channel.

Suppose you want to read news on your mobile device. You'll have several possibilities how to do this:

- You could visit the web site of the news provider, preferably the mobile version. Often they are rather clumsy and filled with advertisements though, and even if they are not, the content displayed will always be coming from the one and only provider who created the web site.
- So, to improve your user experience, you'd probably install a news application. Again, news applications are often provided by the news publishers themselves. These applications have the disadvantage of restricting the articles displayed to those of one provider.
- In search for a general news reader application, you will stumble upon RSS readers, which are probably the most widely used readers nowadays. They will display an excerpt of the full article and lead to the web page of the provider if you are interested in reading more.
- As this isn't really satisfying, there is a recent trend for applications that try to make agreements with news providers. The content providers allow them to publish their articles, often providing a nice and clean way to show content. But this is a very recent trend and it looks like it leads to content where you pay a fee to access it.

Whilst the latest solution seems agreeable from a customer point of view, taken that the customer is willingly to pay for the news he wants, it requires the application developer to have many agreements with the content providers. Also, the data either has to be stored on the application server or the content provider has to provide an interface where to get the data. For many applications this is a huge overhead, especially because to become a useful application one has to have many agreements with content providers. Also, agreements may not always be found.

From a developer point of view it would be nice to crawl all the content that users are interested in, not requiring agreements with each of the providers. Since it's usually not allowed to copy content to one's own server and distribute it from there, this process of fetching and rearranging content would have to be done on the customer device.

The news application in this thesis tries to solve this problem by only providing a file that describes where the important parts of a webpage are. Like this, a user will download the normal web site of the news provider but only see the content he's interested in, for the rest will automatically be discarded. For the application developer this has the advantage that he doesn't have to store any data from other companies on his servers, nor does he use it in any other way than as input to a processing function on the end-users device.

This approach gives the user all the benefits of a general reader and the developer complete freedom which sources to process and include in his application. Figure 1 gives an overview of the application structure.

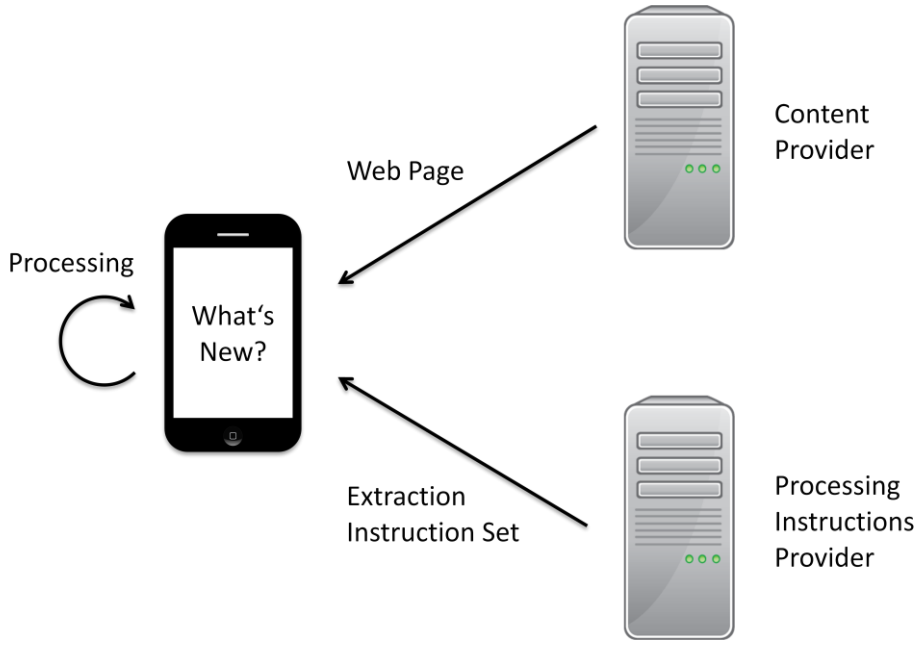

Figure 1: Application Structure

2 Problem Definition

Web sites are often clustered with a lot of things besides relevant content. Have a look at Figure 2, there are a lot of advertisements, links and gadgets you usually don't need nor want to see. The actual article consists of different parts that are spread around, between and in the middle of all these other things.

The goal is to extract the relevant parts of a web site. These parts will be put into classes such as title, subtitles, images, paragraphs or videos. The displaying of this data can then easily be achieved and completely customized, thus allowing for very user friendly designs and a high user experience.

Many on-the-fly processors are very generalized in order to be able to process as many different content sources as possible. The approach chosen for this thesis is a different one. The processing instructions are bound to a certain web site. Most content providers build their sites using a content framework which will always produce output of the same structure. Thus a processing instruction set can be used for all the content of a specific provider. Compared to a generalized processor this allows to take care of specialties in the content provider's web sites, making it possible to easily process all the content for which an instruction set exists.

Further, as the instruction sets are very specific, it would be nice if the community could edit and provide these files, so everyone who whishes a new content to be available could write an extraction file and include it in the application.

This introduces some requirements for the extraction instruction sets.

Figure 2: New York Times Article, www.nytimes.com

3 Requirements for an Extraction Instruction Set

The instruction sets define extraction and processing rules for the web sites fetched. Since a new extraction rule set is needed for each content provider, the sets have to be **simple**. The rule sets must be readable and producible by anyone. It's a requirement that in the end every user has to be able to create and submit his own rule sets.

In order to be able to handle all content from a specific provider, the instruction sets have to be **generalized** to a certain degree, i.e. they have to take into account different document structures or at least give the possibility to do so. Basically, an instruction set should reflect how a content publisher has his web pages created, thus being applicable to all sort of content that is published using the same generator.

As a proof of concept, the content processor will have to cope with three different types of content in this thesis:

- News Article: Article with mixed content of images, titles, paragraphs and embedded videos (articles from New York Times, CNN, Gizmodo, Techcrunch, The Sartorialist)
- Image Article: Single-image article (articles from Flickr)
- Video Article: Single-video article (articles from YouTube, Vimeo)

The latter two can be reduced to the first type for a lot of the processing, but they will need some special processing rules to allow for embedded video display. As for the video display, it turns out either the embed links from the respective providers have to be used or the user has to be forwarded to the native application or the browser. To simplify this, the extraction instructions have to be able to provide custom built elements where the creator can specify the final display of the data.

The extraction instructions sent to the mobile device should contain information about how to extract the following fields from an arbitrary web site.

Title	Title of the article
Date and Time	Date and time when the article has been created
Short Description	Short description to be displayed in an article overview
Thumbnail	A thumbnail previewing the article
Content	Content of the categories subtitle, text, video, music, image

Table 1: Extraction Types

The content will be the biggest and most difficult part to specify, since the extraction has to be generalized for any type of content the content extractor will encounter.

Apart from the content extraction instructions, the protocol must send along the information that is needed to fetch the content. These information fields are:

URL	Link where the article can be found
Date and Time	Date and time when the article has been created, used to sort content
Publisher	The content publisher, such as New York Times, CNN, ...

Table 2: Content Fetch Information

Since the above two parts of the protocol are decoupled (extraction instructions are decoupled from the information on where to fetch the content), the protocol also has to specify which extraction instruction file to use for a certain type of content:

Content ID	The content publisher's ID, such as nytimes, cnn, ...
Extraction File Version	The version of the extraction file, this is used for easy updating and in case a publisher uses completely different content creation mechanisms for different articles

Table 3: Extraction File Binding

4 Related Work

The thesis mainly treats HTML processing and news applications. HTML processing can be split in two different parts. First, html often has to be sanitized to be available in a clean form that can be further processed. The processing then is done by using xml techniques or CSS selectors.

Further, different content extraction approaches as they appeared in research and some applications as they are available today will be presented along with how they solve the content aggregation problem.

4.1 HTML Sanitizing / Parsing

During the process of sanitizing html, the document is cleaned from unknown tags and bad formatting. During the process of parsing, the input document (in the form of a string) is used to create the document object model (DOM). There exist a lot of different sanitizing and parsing methods, every browser has to implement a method and many libraries are available for different languages. A standardization of html parsing can be found at (WHATWG, Apple Computer, Inc., Mozilla Foundation and Opera Software ASA, 2004-2012) under the section Parsing HTML documents.

Libraries looked at for this thesis are JSoup (Hedley, 2009-2011) and TagSoup (Cowan, 2004-2011). They both provide HTML sanitizing, JSoup additionally provides html parsing according to the WHATWG specification described above (creating a DOM tree). Since JSoup provides more functionality than TagSoup and behaves according to the WHATWG standard, it would have been chosen for the proof-of-concept application of this thesis, if not for the highly optimized browser engines. All the browsers have to do sanitizing and parsing themselves, are usually kept up to date and provide highly optimized sanitizing and parsing. Because of that they offer a nice and convenient alternative to do the processing and therefore were chosen over libraries such as JSoup for this thesis.

4.2 HTML Content Processing

Html content is usually processed within the browser using JavaScript (Mozilla Developer Network, 1995-2011). To make this process easier various libraries such as jQuery (The jQuery Project, 2010) exist. Other ways include the standardized XML path language XPath (Anders Berglund, 2010), for which libraries such as JSoup (Hedley, 2009-2011) exist. The browsers offer a well documented and up-to-date way to process html files using JavaScript, but one might argue that since JavaScript is a scripting language run within a browser it will not offer the same speed as an XPath implementation.

4.3 Content aggregation

A lot of research on content extraction and aggregation has been done when the first PDAs appeared on the market. These devices only had minimal capabilities to display web pages, thus the need for content extraction was great. In (Suhit Gupta G. K., 2003), the extraction is done using filters that hide or remove certain elements of html DOM trees. Whilst this approach is effective, it's more of a content reformatting than an extraction, as operations are performed on the DOM, but the same DOM is displayed to the user in the end. This has the advantage that unknown elements might still be displayed to the user, but this can just as fast turn into a disadvantage. The framework was further refined in more recent publications (Suhit Gupta H. B., 2006).

(A. F. R. Rahman, 2001), the first work about single document based content extraction, describes fundamental properties a content extraction system should comply with and proposes an approach that uses structural analysis, contextual analysis and summarization. The approach taken there is to split up the html into pieces and use contextual analysis to determine the importance of the different pieces. Via summarization a short summary of each piece is created that is used to create a link to the respective section (again, presenting small chunks of data was important for early PDAs).

More recent research includes (Gottron, 2008) which gives a complete overview and comparison of different concepts used up to 2008. (Fabio Vitali, 2004) describes a rule-based structural analysis of web pages with some similarities to the rule-based approach in this thesis. The rule set was part of IsaWiki and mainly thought to identify parts of websites that would be interesting for a user to edit. It can be used for content extraction in some way though.

An interesting commercial use of html extraction is the *Opera Mini* (Opera Software, 2011) browser, which sends website requests to Opera servers which will only relay important stuff to the users to decrease overhead. It uses the Extensible Rendering Architecture, which usually does only minor changes though (apart from resizing images, which is its main purpose).

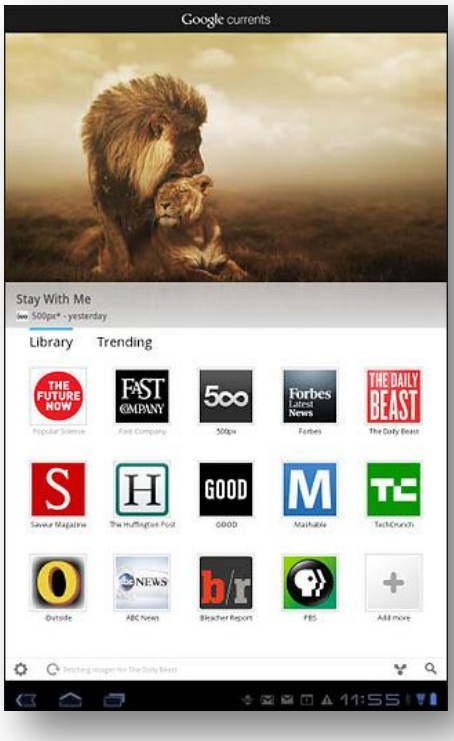

Figure 3: Google Currents Application, http://www.google.com/producer/currents

4.4 News Applications

As for the news applications being developed in this field, most if not all of them (or at least the popular ones) are closed source. Some tests on the Pulse news application (Alphonso Labs, Pulse, 2011) in the scope of this thesis revealed it to be calling public RSS feeds and displaying their content. The testing was performed by routing web traffic of an android device over a wireshark-sniffed interface and looking at the files sent and requested. Other news applications like Taptu (Taptu, 2011) and Google Currents (Google Inc., 2011), shown as example in Figure 3, give strong hints that

they function the same way by only providing content from publically available feeds and blogs. Google Currents, only recently announced, probably intends to incorporate paid subscriptions (the price buttons are already there), but so far providers that would fall into this category only display their regular RSS feeds.

5 HTML Processing Instruction Definition

The following section will outline the scripting language developed in this thesis. It follows the requirements outlined in the problem definition.

For any data passed between the application server and the device xml will be used. This allows fast and easy parsing on the device as a lot of optimized xml parsers are available for many platforms. To reduce overhead and make the system extensible, the whole extraction process is split up into three tasks:

1. A story file is requested from the application server. This file is of a simple structure mostly consisting of the URLs where to fetch the html sites.
2. Depending on the URLs in the story file, html extraction instruction files can be requested on demand and stored locally for later use.
3. Using these extraction instructions, the content is retrieved and made ready for presentation.

5.1 The Story File

The story file describes all the stories the user can flip through once loaded. This is a file that is either requested by the application on startup or as soon as all previously requested stories have been read. Basically, it's an URL collection with a link to an extraction file for each URL. The stories in the story file have the following fields:

Content ID	ID that uniquely describes the content provider, is used to choose the right extraction rule set (stored in variable *$$contentID*).	Any string
Extraction File Version	The Version of the extraction file, this makes it possible to have different versions for different articles. Updating an instruction rule set gets easy, simply increase the version number and supply a new instruction set (stored in *$$extrVersion*).	Any string
Publishing Date	The date the story was crawled by the server (other values possible, e.g. the publishing date). This field is used on the server to sort the stories and on the client to be displayed if there is no other date on the web site that can be extracted (stored in *$$pubDate*).	string of form EEE, dd MMM yyyy HH:mm:ss ZZZZ, Locale.US
Source	The source URL where the story can be downloaded (stored in *$$storySrc*).	Any URL
Variables	Furthermore, it's possible to send any string along in the story file that can be used to specify more sources, video numbers, embedded stuff... These strings are of a form *variableName="StringValue"* and can be accessed via the respective variable *$$variableName*.	Any string

Table 4: Story File Definition

A story file is displayed in Listing 1 (note the format of the publishing date):

```
<stories>
	<story		contentID="cnn"
			extrVersion="4.0"
			pubDate="Thu, 15 Dec 2011 17:05:42 EST"
			src="http://edition.cnn.com/2011/12/15/world/meast/
			syria-main/index.html?eref=edition"/>
	<story		contentID="cnn"
			extrVersion="4.0"
			pubDate="Thu, 15 Dec 2011 23:57:23 EST"
			src="http://edition.cnn.com/2011/12/15/world/asia/
			japannuclear/index.html?eref=edition"
			customVariable="useBorder" />
</stories>
```

Listing 1: Example Story File

5.2 The Instruction Files

The instruction files describe how to extract content from the fetched html and are the core of this thesis. For each content provider there is one file that should be generalized enough to allow processing of all the content from this provider. This file may appear in multiple versions, probably because different articles have a different structure, but mostly to adapt the extraction process to any changes the content provider makes.

The instruction file is a xml file through which the html extractor will go instruction by instruction. Instructions are nested so that each instruction is applied to the same context as the parent instruction. A simple instruction is defined in Listing 2:

```
<ruleset contentID="cnn" version="4.0">
	<loadURL src="$$storySrc" sandboxValue="allow-same-origin">
		<find query=".article > .tline">
			<title>
				<extrText/>
			</title>
		</find>
	<loadURL/>
<ruleset/>
```

Listing 2: Example Ruleset File

The `ruleset` statement declares this as a rule set which can be applied to stories having the `contentID` "cnn" and the `version` "4.0". When this rule set is applied to a story from the story file, all the xml elements within are processed sequentially in a tree-like manner. The first statement is `loadURL`. This statement allows to load any web resource specified in the `src` attribute. In this case "$$storySrc" is loaded. The *$$* makes the statement a variable access, accessing the variable *storySrc*. This variable was supplied by the story file and is nothing else than the URL where to fetch the story.

After the resources behind the URL have been fetched, all xml elements below the `loadURL` statement will be executed. In this case it's only a `find` statement with further child nodes. The `find` statement locates html elements on the web site. Via the `query` attribute a location can be specified using jQuery (The jQuery Project, 2010) statements. These actually boil down to CSS selector (Bert Bos, 2011) statements for most cases. As CSS selectors are an important part in web development, this provides a well documented and easy to use way to find elements on a web site.

CSS selectors usually return a set of matched elements. It is on this set of elements that child nodes of the instruction set will be applied to. In the case of our example, the title of the story is set to be the text found in the ".article > .tline" html node.

The `title` node sets the title of the story to whatever text is found in its child node. In this case, `extrText` is called which simply extracts and returns the text found in the html elements of its parent node. Since the `title` node does not change the set of html elements, this set will be the one found by the `find` statement, thus `extrText` returns the text found in ".article > .tline", which then in turn will be stored as story title.

5.3 Nodes in the Instruction Files

The following table gives an overview of nodes that can be used in the extraction rule set definition, those are the basic nodes:

Name	Function	Attributes	Child Nodes / Notes
LoadURL	Loads any html/xml source to be processed.	**src**: The URL where to fetch the html/xml. **sandboxValue**: Values for the iFrame sandbox, allows JavaScript functions to be called if *allow-scripts* is set.	All loadURL statements are stored in a list which will be processed sequentially (can cause trouble when variables are shared between multiple loadURL rules). Child node: find
Find	Changes the html element set for all child nodes to the one found by the query.	**query**: The CSS selector to find the element.	Child node: Any
Select	Selects a single element from a html element set.	**nElement**: Specifies which element to take ([0, n-1] for a set containing n elements).	Child node: Any
First	Selects the first element from an element set.		Child node: Any
Last	Selects the last element from an element set.		Child node: Any
Not	Removes elements from the set.	**query**: The CSS selector to find the elements to be excluded from the set.	Child node: Any
Iterate	Iterates over the element set, applying child nodes to all elements.		Child node: Any
Children	Grab all the child elements of the currently active		Child node: Any

Name	Function	Attributes	Child Nodes / Notes
	element.		
Remove	Removes elements from the DOM tree.	**query**: The CSS selector to find the elements to be removed.	Child node: None
RemoveSelf	Removes the element itself from the DOM tree.		Child node: None
Strip	Strips an element from tags (making it plain text).		Child node: None
StripSelf	Strips the selected elements from tags, including their own tag.		Child node: None

Table 5: Basic Nodes in Instruction Set

The following nodes control the flow during processing:

Name	Function	Attributes	Child Nodes / Notes
IfAttr	Checks if a given node fulfills some attribute criteria.	**attr**: Checks if the node has the given attribute. **value**: If set, it additionally checks if the attribute has the given value.	Child node: Any
IfNodeName	Checks the node name.	**name**: Checks if the node name has the given value.	Does not look on lower-, upper-case. Child node: Any
IfStrContains	Checks if a string contains some other string.	**string:** The string to check for occurrences of *value*. **value**: The value that might or might not appear in the string.	Statement only makes sense when used with variables. Child node: Any
IfContains	Checks if a node contains other nodes of the type specified by the CSS selector.	**query**: The CSS selector that defines the html elements to look for.	Child node: Any
Else	Else rule for all if nodes, has to be written right after the if node closes, will be called whenever the if statement is false.		Child node: Any

Table 6: Flow Control Nodes in Instruction Set

The following nodes extract text and attributes from the html:

Name	Function	Attributes	Child Nodes / Notes
ExtrAttr	Extracts an attribute value from the node.	**attr**: The attribute where to extract the value.	Returns a string. Child node: None
ExtrHtml	Extracts html from within a node.		Returns a string. Child node: None
ExtrText	Extracts text from within a node.		Returns a string. Child node: None
customText	Allows to enter custom text.	**txt**: The text to be inserted.	Child node: None
customCDATA	Allows to enter custom	**CDATA**: The CDATA to be	Child node: None

Name	Function	Attributes	Child Nodes / Notes
	CDATA.	inserted.	
Cut	Cuts a string to the part in between *cutoffStart* and *cutoffStop*.	**cutoffStart**: The start of the cut. **cutoffStop**: The end of the cut.	*Cutoff* strings are regular expressions, special characters have to be escaped (escape character '\'). Child node: Any string extraction node

Table 7: Extraction Nodes in Instruction Set

The following nodes will set elements of the story:

Name	Function	Attributes	Child Nodes / Notes
Variable	Defines a variable, can be accessed in any string via $$variablename.	**name**: The variable name. **value**: The value of the variable, if no value is specified, the child node has to provide the value.	Part of story. Can only be accessed after defined (remember tree-like traversal of instructions, and sequential processing of *loadURL*). Child node: Any string extraction node
Title	Sets the story title.		Part of story. Child node: Any string extraction node
Author	Sets the story author.		Part of story. Child node: Any string extraction node
Thumbnail	Sets the story thumbnail URL.		Part of story. Child node: Any string extraction node
ShortDesc	Sets a short description for the story.		Part of story. Child node: Any string extraction node
SubTitle	Sets a story subtitle.		Part of story content. Child node: Any string extraction node
Image	Creates an image to be placed in the story content (URL).	**src**: Optionally; if specified provides an URL where to load the image from.	Part of story content. Child node: Any string extraction node
Text	Creates text to be placed in the story content.		Part of story content. Child node: Any string extraction node
Div	Creates a custom container where you can place whatever html you want.		Part of story content. Child node: Any string extraction node

Table 8: Element Nodes in Instruction Set

Wherever variables turn up in instructions, they will be substituted if they have been defined before (substitution of *$$variableName*). After the content extraction, a story object will be available that contains all the data and is ready to be displayed.

6 A Proof-of-Concept Implementation

The application developed in this thesis should show the possibilities of the extraction language. In addition, part of the thesis was to create a *nice* display of the aggregated content, so that further work on the project would have a good start for being visualized.

The application is split in several parts. On the client side there is the implementation of the extraction language as well as the display of the aggregated content. On the server side there is a simple crawler that crawls some RSS feeds and a servlet that provides the user application with the newest stories available.

6.1 Implementation of the Extraction Language

In order to access the structures of the downloaded web sites they have to be parsed and sanitized. There exist a great number of html sanitizers and libraries that perform this task (see section HTML Sanitizing / Parsing). Another very easy way to do this is the usage of the built-in browser (via the WebView widget available on most platforms; a WebView simply is a browser implementation without user controls). After loading the web page into the WebView it will automatically prepare the html for display and further processing, in the process even load asynchronous requests over JavaScript (Ajax calls). The implementation in this thesis is done using an invisible iframe within a WebView. The pages are loaded into it and processed there, without the user realizing anything about it. The iframe within the WebView is necessary, as the rest of the application will run in the WebView (see description under The Surrounding Application: Implementation).

To parse the html and extract important parts according to the protocol definition, some CSS selection tool has to be used. Again, the built-in browser can be used in combination with JavaScript techniques to query and process elements of the web site. To make the process easier, the jQuery (The jQuery Project, 2010) library is additionally chosen in this thesis.

Another aspect is the JavaScript programming itself, which is needed to control the selection and extraction process. In this thesis Google Web Toolkit (Google Inc., 2011) has been used, since it allows fast development (programming Java) and highly optimized deployment (GWT has a very powerful Java to JavaScript compiler). Also, further ports to non-JavaScript implementations will be easier as the code is already in Java.

The extraction part is written in a single static class featuring different methods for different extraction nodes. Those methods are called recursively, each call adapting the set of html elements and the extraction rules to be further processed.

The first function to be described here is the main entry point `extract()`. This function loads html from a new URL via an asynchronous call. Once the requested data has been received, it is put into the iframe for which an ONLOAD event listener exists. The ONLOAD event will fire as soon as the page is completely loaded, which can be useful when the page itself requests more elements via JavaScript. In case the ONLOAD never fires (which might happen if a resource is not available), a timeout is set when to start with the extraction anyways. The extraction now being done is mainly performed by the `extractElement()` function.

The `extractElement()` function handles *basic nodes*, *control nodes* and *element nodes*. Depending on the node encountered, it performs the requested action (find, select, children,…) and calls itself again on a new set of html elements with subsequent nodes as extraction instructions.

Since most of the nodes perform some jQuery action, the calls are often simple, reducing the chance to introduce errors in the code. Listing 3 shows the call for the *first* node (in this case the jQuery call `root.first()`does all the work):

```
// Selects the first node of the matched elements set
if (nodeType.equals(Constants.NODE_FIRST)) {
        extractElement(iFrameController, node, root.first(), story,
                                container, depth + 1);
}
```

Listing 3: Code Snippet of extractElement()

For nodes that accept a *string extraction node* as child, the respective function `extractString()` is called. This function handles the different *extraction nodes*. If there is more than one extraction node, the extracted strings will be concatenated. Listing 4 shows the extraction for the *extrText* node:

```
// Extract string as text from within current root node
if (nodeType.equals(Constants.NODE_EXTR_TEXT)) {
        retString = retString + root.text();
}
```

Listing 4: Code Snippet of extractString()

There is one more function that replaces variables found in strings with the respective values. It iterates through all the variables stored and checks the input string for occurrences of *$$variableName*.

The code for the HTMLExtractor can be found in the Appendix.

6.2 The Surrounding Application: Requirements

The application requirements are kept simple, a simple application makes it easy for the user to get started. Also, the application should be mostly a proof of concept showing the possibilities of the developed extraction instructions. It lacks user specific content and application customization, apart from changing the font size.

Basically, the application will feature a simple container displaying one story at a time. A new story can be displayed by swiping in this main container (a swipe to the left will show a new story, a swipe to the right will go back one story). The story will be rendered with a big title at the top, an author and date panel below, followed by all the content.

Furthermore, there's an options panel at the top, having one button to display an information dialog, two buttons to specify the font size and a spinner icon when the application is loading data (or when the user has to wait until new stories are available).

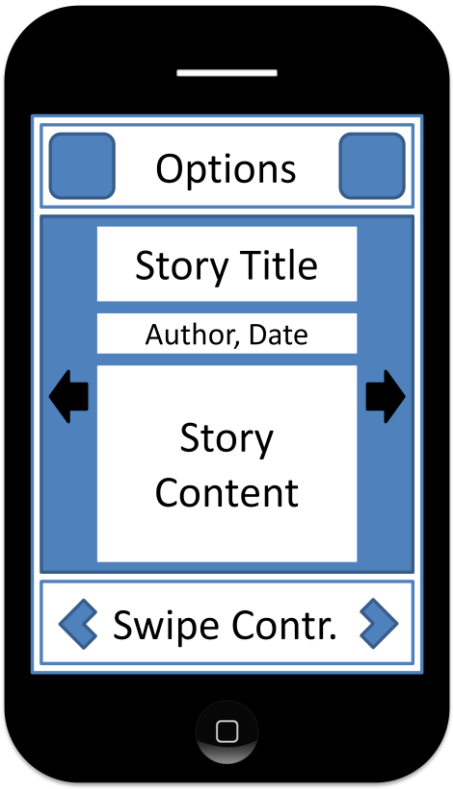

Figure 4: Application Layout

On the bottom, a swipe controller is displayed on devices with big screens. Using the arrows on this panel, one can alternatively flip through the stories by clicking instead of swiping.

The application is visualized in Figure 4.

6.3 The Surrounding Application: Implementation

First of all, a decision had to be made which mobile platforms will be used and supported. In order to allow very broad deployment the first thought was to develop a web page that will do everything on the fly using JavaScript. The problem with this approach is that the same origin policy of browsers does not allow content to be aggregated from different servers outside the same domain. The only way this is allowed is through iframes, but the content of an iframe of foreign origin cannot be accessed.

The most obvious solution would be to program a native application, which is bound to one platform though. Since it would be much more convenient to deploy to multiple platforms at the same time, another solution was chosen. Most platforms support a browser-like widget called WebView, as described above. It's basically the platforms browser without the standard controls (history back button, URL field, reload button, ...), giving the developer all the control over content whilst the user doesn't realize it's a browser. Based on this widget, there are frameworks that allow to program regular HTML/CSS/JavaScript which will be deployed in the WebView. These frameworks take care of the binding to the different platforms. Also, they use native functions to circumvent the same origin problem stated above.

For this thesis, the framework PhoneGap (Adobe Systems Inc. PhoneGap, 2011) was chosen for its simple and intuitive design, its pricing and license, its wide platform support and the possibility to develop using standard web techniques. Other possible frameworks that were evaluated are Rhomobile Rhodes (Rhomobile, 2011), Appcelerator Titanium (Appcelerator Inc., 2011) and MoSync (MoSync AB, 2011). They were discarded for different reasons: Incompatible licenses, missing support for certain platforms, no support for common web technologies.

Summarizing, the used technologies are:

- GWT to write the whole application, is then compiled to JavaScript.
- GWTQuery (The GwtQuery Team, 2011), a jQuery implementation for GWT to make the usage of CSS selectors easy.
- PhoneGap which embeds the JavaScript into the WebView of the respective platform.

The application flow is specified in Figure 5. The dotted lines show asynchronous requests. These requests will time out after some duration, in case a content provider is not online, a web site cannot be parsed or the connectivity of the mobile device is not sufficient.

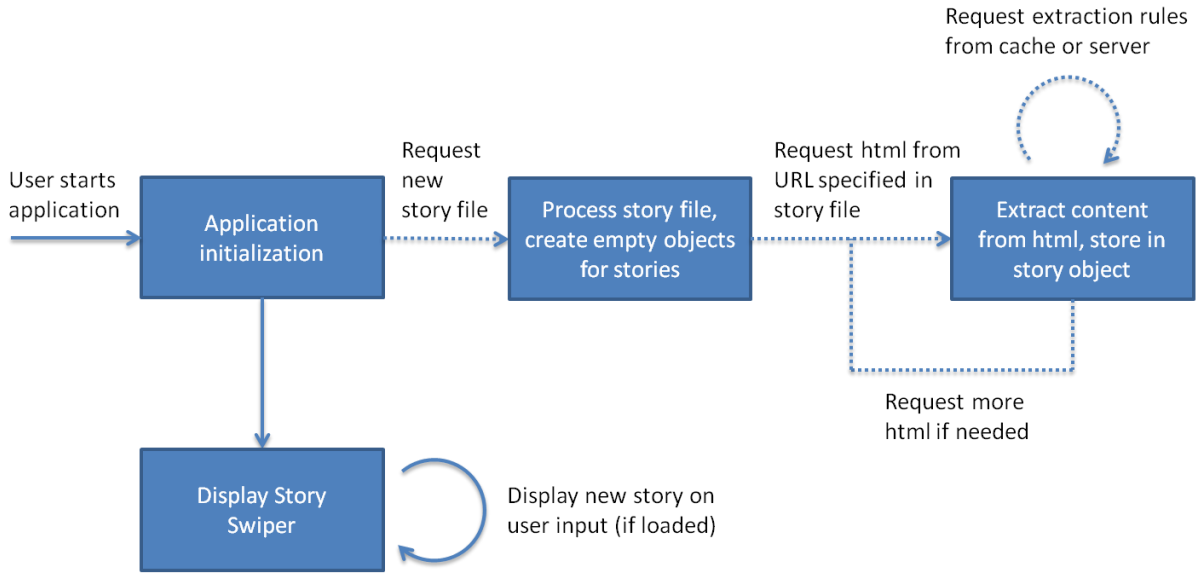

Figure 5: Application Flow

The loaded stories are displayed in the story swiper, a widget that allows swiping through its content. The application tries to stay ahead of the user by loading stories in the background, so no delay is introduced when the user swipes the main container to view a new story. If the application doesn't manage to load stories fast enough, the spinner icon in the options panel will show, showing to the user that he has to wait for the application to finish extraction.

The story handling of the application is roughly split along a model-view-controller scheme, but as a proof-of-concept and entry point for further research in the area it wasn’t an absolutely necessary requirement.

Figure 6 shows the final application running in the emulator, displaying some debug story that was used to optimize the layout and on a Samsung Galaxy S displaying a story from New York Times.

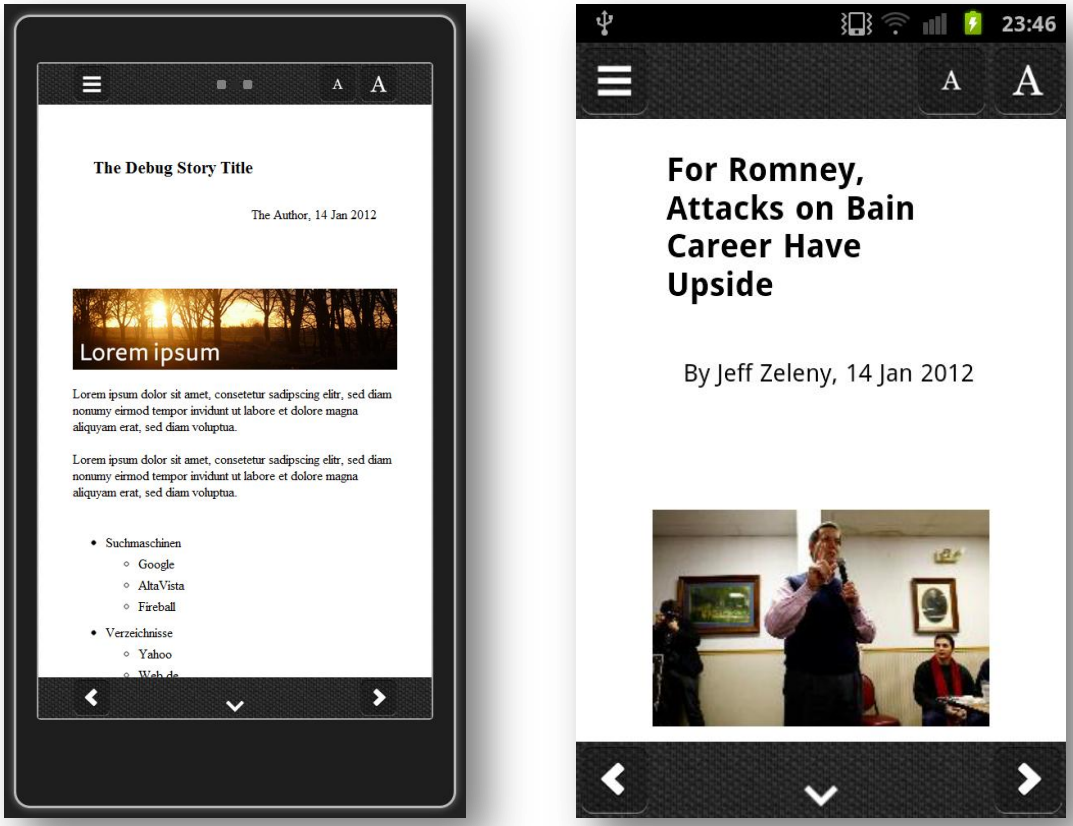

Figure 6: Application Screenshots

6.4 The Server Part

The RSS crawler of the server part is a simple servlet that fetches a defined set of RSS feeds every 5 minutes. It stores the found stories in xml files named after the content ID of the content providers. It's on these xml files the story fetcher operates. Whenever a user requests a new set of stories, the fetcher is arranging the stories in the xml files according to some defined criteria (mostly publishing date, but for some high frequent content providers like youtube, some artificial reduction) and sending the thus created file back to the client.

The client also can request extraction rules from the server, just by specifying the content ID and extraction file version number. It's another servlet that handles these requests.

For the creation of extraction rule files, it's recommended to use the *developer tools* shipped with all modern browsers. They usually allow changing of the user agent (important to see which files a content provider will send on your request), show CSS selectors of the elements and give a good insight into the website.

7 Conclusions

The application developed shows that the extraction rule definitions work well. From the test set of 7 content providers all content could be retrieved and displayed in a nice format. Sometimes some workarounds had to be used to make data transmission overhead as small as possible and because

the web sites would respond with different sites based on the user agent of the device (string that identifies the device, it's capabilities, the browser used, …), but this was always possible without any great effort. Also, the parsing goes reasonably fast (especially since it runs with JavaScript in a browser, which is single-threaded and a scripting language), at least on more advanced and powerful devices as they are sold today.

Further work could include a more powerful server that is able to aggregate more news sources and user specific content. On the client side, more customization would be preferable, especially to select which content providers should be displayed and how their stories should be visualized. Also, measuring the time a user spends reading a story would be interesting, especially for selection of user specific content. Within regard to this, an interesting improvement would be the automatic selection of stories similar to those the user spent a lot of time with.

One has to think of possible advantages of a native application version. For example those could be the possibility to synchronize whenever WLAN is available and to store the stories locally. Also, a more responsive user interface could be possible since the extraction could be decoupled from the display.

Automated generation of extraction files would be desirable, or at least tools to help users who want to create their own extraction files. In this field, a mechanism for users to upload, share, correct and expand extraction rule sets would also be interesting to develop.

8 Acknowledgements

Special thanks to my supervisors Samuel Welten and Mihai Calin, who helped me a lot finding the right extraction rule set, also with all the technical stuff, technologies and the design and layout of the application. Credits for the icons used within the application go to Mihai, thank you!

Thanks to Professor Dr. Roger Wattenhofer for giving the opportunity to write this interesting thesis in his group.

And last but not least thanks to my family and friends, who helped me with ideas, application testing and proof-reading of the thesis.

9 References

A. F. R. Rahman, H. A. (2001). Content Extraction from HTML Documents. *1st Int. Workshop on Web, WDA 2001*, (pp. 7-10). Seattle, Washington, USA.

Adobe Systems Inc. PhoneGap. (2011). *PhoneGap*. Retrieved January 13, 2012, from PhoneGap: http://phonegap.com/contact

Alphonso Labs, Pulse. (2011). *Pulse News*. Retrieved January 13, 2012, from Pulse News: http://www.pulse.me/

Anders Berglund, S. B. (2010, December 14). *W3C: XML Path Language (XPath) 2.0 (Second Edition).* Retrieved January 13, 2011, from XML Path Language (XPath) 2.0 (Second Edition): http://www.w3.org/TR/2010/REC-xpath20-20101214/

Appcelerator Inc. (2011). *Use Appcelerator Titanium to build mobile apps for iPhone & Android and desktop apps for Windows, Mac OS X & Linux from Web technologies*. Retrieved January 13, 2012, from Use Appcelerator Titanium to build mobile apps for iPhone & Android and desktop apps for Windows, Mac OS X & Linux from Web technologies: http://www.appcelerator.com/

Bert Bos, T. C. (2011, June 07). *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. Retrieved January 13, 2012, from Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification: http://www.w3.org/TR/CSS2/cover.html

Cowan, J. (2004-2011). *TagSoup*. Retrieved January 13, 2012, from TagSoup: Just Keep On Truckin': http://ccil.org/~cowan/XML/tagsoup/#other

Fabio Vitali, A. D. (2004, July). Rule-based structural analysis of web pages. *DAS 2004: Proceedings of the 6th International Workshop on Document Analysis Systems* .

Google Inc. (2011). *Google Currents*. Retrieved January 13, 2012, from Google Currents: http://www.google.com/producer/currents

Google Inc. (2011). *Google Web Toolkit - Google Code*. Retrieved January 13, 2012, from Google Web Toolkit - Google Code: http://code.google.com/intl/de/webtoolkit/

Gottron, T. (2008). *Content Extraction: Identifying the Main Content in HTML Documents.* Johannes-Gutenberg-Universität Mainz.

Hedley, J. (2009-2011). *JSoup*. Retrieved January 13, 2012, from JSoup: Java HTML Parser: http://jsoup.org/

MoSync AB. (2011). *Create iPhone and Android apps with JavaScript and C++ | cross-platform mobile application development*. Retrieved January 13, 2012, from Create iPhone and Android apps with JavaScript and C++ | cross-platform mobile application development: http://www.mosync.com/

Mozilla Developer Network. (1995-2011). *JavaScript Official Documentation*. Retrieved January 13, 2012, from JavaScript Official Documentation: https://developer.mozilla.org/en/JavaScript

Opera Software. (2011). *Opera Mini & Mobile*. Retrieved January 13, 2012, from Opera Mini & Mobile: http://www.opera.com/mobile/

Rhomobile. (2011). *Cross-Platform Mobile Development | Rhomobile*. Retrieved January 13, 2012, from Cross-Platform Mobile Development | Rhomobile: http://rhomobile.com/

Suhit Gupta, G. K. (2003, May 20-24). DOM-based Content Extraction of HTML Documents. *WWW '03 Proceedings of the 12th international conference on World Wide Web* , pp. 207-214.

Suhit Gupta, H. B. (2006). Verifying Genre-based Clustering Approach. *WWW '06: Proceedings of the 15th International Conference on World Wide Web* , pp. 875-876.

Taptu. (2011). *Taptu - News DJ!* Retrieved January 13, 2012, from Taptu - News DJ!: http://www.taptu.com/

The GwtQuery Team. (2011). *gwtquery - A jQuery clone for GWT, and much more*. Retrieved January 13, 2012, from gwtquery - A jQuery clone for GWT, and much more: http://code.google.com/p/gwtquery/

The jQuery Project. (2010). *jQuery.* Retrieved January 13, 2012, from jQuery: Write less, do more: http://jquery.org/

WHATWG, Apple Computer, Inc., Mozilla Foundation and Opera Software ASA. (2004-2012). *WHATWG HTML Living Standard*. Retrieved January 13, 2012, from WHATWG HTML Living Standard: http://www.whatwg.org/specs/web-apps/current-work/multipage/index.html

10 Appendix

10.1 HTMLExtractor.java

```
package ch.ethz.disco.whatsnew.client.htmlextractor;

import static com.google.gwt.query.client.GQuery.$;

import java.util.ArrayList;
import java.util.Set;

import ch.ethz.disco.whatsnew.client.IFrameController;
import ch.ethz.disco.whatsnew.client.siteelement.DivElement;
import ch.ethz.disco.whatsnew.client.siteelement.ImageElement;
import ch.ethz.disco.whatsnew.client.siteelement.SiteElement;
import ch.ethz.disco.whatsnew.client.siteelement.SubTitleElement;
import ch.ethz.disco.whatsnew.client.siteelement.TextElement;
import ch.ethz.disco.whatsnew.client.siteelement.URLLoaderElement;
import ch.ethz.disco.whatsnew.client.story.Story;
import ch.ethz.disco.whatsnew.client.utils.Constants;
import ch.ethz.disco.whatsnew.client.utils.Utils;

import com.google.gwt.core.client.GWT;
import com.google.gwt.http.client.Request;
import com.google.gwt.http.client.RequestBuilder;
import com.google.gwt.http.client.RequestCallback;
import com.google.gwt.http.client.RequestException;
import com.google.gwt.http.client.Response;
import com.google.gwt.query.client.GQuery;
import com.google.gwt.user.client.DOM;
import com.google.gwt.user.client.Event;
import com.google.gwt.user.client.EventListener;
import com.google.gwt.user.client.Timer;
import com.google.gwt.xml.client.Element;
import com.google.gwt.xml.client.Node;
import com.google.gwt.xml.client.NodeList;

/**
 * @author Dominik Bucher, D-ITET, DCG
 *
 *         A class for automated extraction of parts of html. Used only static.
 */
public abstract class HTMLExtractor {
        /**
         * Loads a web page into the iframe and starts extracting from it. First,
         * the sandbox value of the iframe is set to ensure scrips are only run when
         * desired. An asynchronous request is made to the URL, fetching the html or xml
```

```
 * at the location. Once the response has been received it is put into the
 * iframe on which an eventlistener is registered that waits for the ONLOAD
 * event (to ensure the iframe can load stuff on it's own if needed). There
 * is also a timer that ensures that the content will be parsed even if no
 * ONLOAD event fires. Once the ONLOAD event has fired
 * or the timer has triggered, a call to extractElement is made where the
 * actual extraction takes place.
 *
 * @param extractionRules
 *            The ruleset of the extraction
 * @param URL
 *            URL where to fetch the html / xml
 * @param iFrameController
 *            Controller that handles interaction with the iframe
 * @param story
 *            Story that defines certain things (i.e. sandbox attributes)
 * @param container
 *            The story container where the parsed elements will be thrown
 *            into
 * @param depth
 *            Function depth, used to determine in which level the xml tree
 *            parsing is
 */
public static void extract(final Element extractionRules, final String URL, final IFrameController iFrameController, final Story story, final
                    ArrayList<SiteElement> container, final int depth) {
        iFrameController.clearIFrame();
        iFrameController.getIFrame().setAttribute("sandbox", extractionRules.getAttribute("sandboxValue"));

        RequestBuilder builder = new RequestBuilder(RequestBuilder.GET, URL);

        try {
                builder.sendRequest(null, new RequestCallback() {
                        boolean requestParsed = false;
                        @Override
                        public void onResponseReceived(Request request,
                                        Response response) {
                                // Can be interesting to see what the actual html received looks like
                                //Window.alert(response.getText());
                                iFrameController.clearIFrame();

                                iFrameController.fillIFrame(response.getText());
                                DOM.sinkEvents((com.google.gwt.user.client.Element) $("iframe.invisibleFrame").elements()[0], Event.ONLOAD);
                                DOM.setEventListener((com.google.gwt.user.client.Element)              $("iframe.invisibleFrame").elements()[0],
                                                                                                        new EventListener() {
                                        @Override
                                        public void onBrowserEvent(Event event) {
                                                if(event.getTypeInt() == Event.ONLOAD) {
                                                        if (!requestParsed) {
                                                                requestParsed = true;
```

```
                                                        extractElement(iFrameController, extractionRules,
                                                                $("iframe.invisibleFrame").contents(), story, container, depth);
                                                }
                                        }
                                }
                        });

                        Timer t = new Timer() {
                        @Override
                                public void run() {
                                        if (!requestParsed) {
                                                requestParsed = true;
                                                extractElement(iFrameController,  extractionRules,  $("iframe.invisibleFrame").contents(),
                                                        story, container, depth);
                                        }
                                }
                        };
                        t.schedule(Constants.MAX_URLLOAD_TIMEOUT);
                }

                @Override
                public void onError(Request request, Throwable exception) {
                        Utils.logToDebugConsole("Error, server does not respond with html file requested", Constants.LOG_ERROR);
                        GWT.log("Error, server does not respond with html file requested",
                                        exception);
                        story.onFinishedLoadingStory(false);
                }
        });
    } catch (RequestException e) {
            Utils.logToDebugConsole("Error requesting xml file, client side", Constants.LOG_ERROR);
            GWT.log("Error requesting xml file, client side", e);
            story.onFinishedLoadingStory(false);
    }
}

/**
 * Fetches content from a web page that has already been loaded via
 * extract(). Iterates through elements in xml, doing the specified actions.
 * For new web sites to load, the current one is being finished, then a new
 * one is loaded via extract(). The function is called recursively on all
 * nodes found.
 *
 * @param iFrameController
 *                      Controller that handles interaction with iframe
 * @param extractionRules
 *            The ruleset found below the loadURL node (or similar)
 * @param root
 *            Topmost jQuery selection for which the function (and xml
 *            nodes) is applied
 * @param story
```

```
 *            Story for which the current extraction is being done
 * @param container
 *            Place where the extracted things are being put
 * @param depth
 *            Function depth, used to determine when recursion is at top
 *            again, to load new URL's. Usually, recursive calls are made
 *            with depth + 1
 */
public static void extractElement(final IFrameController iFrameController, final Element extractionRules, final GQuery root, final Story
                                                                        story, final ArrayList<SiteElement> container, final int depth) {

        NodeList nodes = extractionRules.getChildNodes();

        for (int i = 0; i < nodes.getLength(); i++) {
                // Checks if the node is indeed an element node (not a text node, ...)
                if (nodes.item(i).getNodeType() == Node.ELEMENT_NODE) {
                        final Element node = (Element)nodes.item(i);
                        String nodeType = node.getNodeName();

                        // The loadURL node creates a new URLPanel, that is loaded once the current web site has finished with the extraction
                        if (nodeType.equals("loadURL")) {
                                URLLoaderElement urlLoader = new URLLoaderElement(container.size());
                                if (node.hasAttribute("src")) {
                                        urlLoader.URL = node.getAttribute("src");
                                }
                                urlLoader.extractionRules = node;
                                story.urlLoaders.add(urlLoader);
                                container.add(urlLoader);
                                Utils.logToDebugConsole("Created load URL panel to: " + urlLoader.URL, Constants.LOG_VERBOSE);

                        // The find node searches for a certain jQuery, executing further nodes under this root
                        } else if (nodeType.equals(Constants.NODE_FIND)) {
                                extractElement(iFrameController, node, root.find(replaceVariables(node.getAttribute("query"), story)), story,
                                        container, depth + 1);

                        // The select node selects a single element of a jQuery set, executing further nodes under this root
                        } else if (nodeType.equals(Constants.NODE_SELECT)) {
                                try {
                                        if (!root.eq(Integer.parseInt(node.getAttribute("nElement"))).isEmpty()) {
                                                extractElement(iFrameController,  node,  root.eq(Integer.parseInt(node.getAttribute("nElement"))),
                                                        story,  container, depth + 1);
                                        }
                                } catch (NumberFormatException e) {
                                        Utils.logToDebugConsole("Extraction file spec. wrong, nElement has to be a number",
                                                Constants.LOG_ERROR);
                                }

                        // Selects the first node of the matched elements set
                        } else if (nodeType.equals(Constants.NODE_FIRST)) {
                                extractElement(iFrameController, node, root.first(), story, container, depth + 1);
```

```
// Selects the last node of the matched elements set
} else if (nodeType.equals(Constants.NODE_LAST)) {
	extractElement(iFrameController, node, root.last(), story, container, depth + 1);

// Removes the queried nodes from the previous set
} else if (nodeType.equals(Constants.NODE_NOT)) {
	extractElement(iFrameController, node, root.not(replaceVariables(node.getAttribute("query"), story)), story,
		container, depth + 1);

// The iterate node iterates through a set of jQuery nodes, executing further nodes under the respective html node
} else if (nodeType.equals(Constants.NODE_ITERATE)) {
	for (int j = 0; j < root.length(); j++) {
		extractElement(iFrameController, node, root.eq(j), story, container, depth + 1);
	}

// The children node selects all the children of a given node, executing further nodes under this jQuery set
} else if (nodeType.equals(Constants.NODE_CHILDREN)) {
	if (root.children().length() > 0) {
		extractElement(iFrameController, node, root.children(), story, container, depth + 1);
	}

// The remove node removes the via jQuery selected html nodes
} else if (nodeType.equals(Constants.NODE_REMOVE)) {
	root.remove(replaceVariables(node.getAttribute("query"), story));

// This removes the matched set itself
} else if (nodeType.equals(Constants.NODE_REMOVE_SELF)) {
	root.remove();

// The strip node strips a html node from all children html tags (leaving the top one intact)
} else if (nodeType.equals(Constants.NODE_STRIP)) {
	root.html(root.text());

// The strip self node strips the currently selected node, meaning it removes the html tag
} else if (nodeType.equals(Constants.NODE_STRIP_SELF)) {
	root.after(root.html());
	root.remove();

// The variable node creates a variable that can later be accessed via $$variableName
} else if (nodeType.equals(Constants.NODE_VARIABLE)) {
	if (node.hasAttribute("name") && node.getAttribute("name").length() > 0) {
		if (node.hasAttribute("value")) {
			story.variables.put(node.getAttribute("name"), node.getAttribute("value"));
		} else {
			story.variables.put(node.getAttribute("name"), extractString(node, root, story, depth + 1));
		}
	} else {
		Utils.logToDebugConsole("Variable error: Either name is incorrect or variable already exists in story",
			Constants.LOG_ERROR);
```

```
        }

// The if attribute node searches for a given attribute and proceeds processing only if the attribute is found
} else if (nodeType.equals(Constants.NODE_IF_ATTR)) {
        if (root.first().attr(replaceVariables(node.getAttribute("attr"),
                        story)).equals(replaceVariables(node.getAttribute("value"), story))) {
                extractElement(iFrameController, node, root.first(), story, container, depth + 1);
        } else {
                // +2 will return the next element node (jumping a text node)
                if (nodes.item(i + 2) != null) {
                        if (nodes.item(i + 2).getNodeName().equals(Constants.NODE_ELSE)) {
                                extractElement(iFrameController, (Element)nodes.item(i + 2), root, story, container,
                                        depth + 1);
                        }
                }
        }

// The if attribute contains node searches for a given attribute and proceeds processing only if the attribute contains
                                                                                                        the given string
} else if (nodeType.equals(Constants.NODE_IF_STR_CONTAINS)) {
        if (replaceVariables(node.getAttribute("string"),
                story).contains(replaceVariables(node.getAttribute("contains"), story))) {
                extractElement(iFrameController, node, root.first(), story, container, depth + 1);
        } else {
                // +2 will return the next element node (jumping a text node)
                if (nodes.item(i + 2) != null) {
                        if (nodes.item(i + 2).getNodeName().equals(Constants.NODE_ELSE)) {
                                extractElement(iFrameController, (Element)nodes.item(i + 2), root, story, container,
                                        depth + 1);
                        }
                }
        }

// The if nodename node checks if the current root has the given name
} else if (nodeType.equals(Constants.NODE_IF_NODE_NAME)) {
        if (root.get(0).getNodeName().toLowerCase().equals(replaceVariables(node.getAttribute("name"), story))) {
                extractElement(iFrameController, node, root.first(), story, container, depth + 1);
        } else {
                // +2 will return the next element node (jumping a text node)
                if (nodes.item(i + 2) != null) {
                        if (nodes.item(i + 2).getNodeName().equals(Constants.NODE_ELSE)) {
                                extractElement(iFrameController, (Element)nodes.item(i + 2), root, story, container,
                                        depth + 1);
                        }
                }
        }

// The if node contains checks if the current root contains nodes of a certain query
} else if (nodeType.equals(Constants.NODE_IF_CONTAINS)) {
        if (root.find(replaceVariables(node.getAttribute("query"), story)).length() > 0) {
```

```
                extractElement(iFrameController, node, root, story, container, depth + 1);
        } else {
                // +2 will return the next element node (jumping a text node)
                if (nodes.item(i + 2) != null) {
                        if (nodes.item(i + 2).getNodeName().equals(Constants.NODE_ELSE)) {
                                extractElement(iFrameController, (Element)nodes.item(i + 2), root, story, container,
                                        depth + 1);
                        }
                }
        }

// The else node is executed if an if node returns false
} else if (nodeType.equals(Constants.NODE_ELSE)) {
        // Do nothing, as else node is handled in if nodes

// Creates an image element in the story container
} else if (nodeType.equals(Constants.NODE_IMG)) {
        ImageElement img;
        if (node.hasAttribute("src")) {
                img = new ImageElement(replaceVariables(node.getAttribute("src"), story));
        } else {
                img = new ImageElement(extractString(node, root.first(), story, depth + 1));
        }
        container.add(img);

// Creates a text element in the story container
} else if (nodeType.equals(Constants.NODE_TXT)) {
        TextElement txt = new TextElement(extractString(node, root.first(), story, depth + 1));
        container.add(txt);

// Creates a div element in the story container
} else if (nodeType.equals(Constants.NODE_DIV)) {
        DivElement divEl = new DivElement(extractString(node, root.first(), story, depth + 1));
        container.add(divEl);

// Sets the story title
} else if (nodeType.equals(Constants.NODE_TITLE)) {
        story.title = extractString(node, root.first(), story, depth + 1);
        Utils.logToDebugConsole("Story title: "+story.title, Constants.LOG_DEBUG);

// Sets a subtitle for the story
} else if (nodeType.equals(Constants.NODE_SUBTITLE)) {
        SubTitleElement subEl = new SubTitleElement(extractString(node, root.first(), story, depth + 1));
        container.add(subEl);

// Sets the src url of the story thumbnail
} else if (nodeType.equals(Constants.NODE_THUMBNAIL)) {
        story.thumbnailSrc = extractString(node, root.first(), story, depth + 1);
        Utils.logToDebugConsole("Story thumb: "+story.thumbnailSrc, Constants.LOG_VERBOSE);
```

```
                        // Sets a short description for the story
                        } else if (nodeType.equals(Constants.NODE_SHORT_DESC)) {
                                story.shortDesc = extractString(node, root.first(), story, depth + 1);
                                Utils.logToDebugConsole("Story sDesc: "+story.shortDesc, Constants.LOG_VERBOSE);

                        // Sets an author for the story
                        } else if (nodeType.equals(Constants.NODE_AUTHOR)) {
                                story.author = extractString(node, root.first(), story, depth + 1);
                                Utils.logToDebugConsole("Story author: "+story.author, Constants.LOG_VERBOSE);

                        // Sets the date for the story
                        } else if (nodeType.equals(Constants.NODE_DATE)) {
                                story.dateStr = extractString(node, root.first(), story, depth + 1);
                                Utils.logToDebugConsole("Story author: "+story.author, Constants.LOG_VERBOSE);
                        }
                }
        }


        // On top recursive level, go on loading other URL's as discovered before; if there are none, set story loaded to true
        if (depth <= 0) {
                iFrameController.clearIFrame();
                if (!story.urlLoaders.isEmpty()) {
                        URLLoaderElement urlLoaderExtr = story.urlLoaders.remove(0);
                        Utils.logToDebugConsole("Loading next url panel: " + urlLoaderExtr.URL, Constants.LOG_VERBOSE);
                        extract(urlLoaderExtr.extractionRules, replaceVariables(urlLoaderExtr.URL, story),iFrameController, story,
                                urlLoaderExtr.subElements, 0);
                        Utils.logToDebugConsole("URLPanels contains " + story.urlLoaders.size() + " elements", Constants.LOG_VERBOSE);
                } else {
                        story.loaded = true;
                        story.waitingForReply = false;
                        story.onFinishedLoadingStory(true);
                }
        }
}

/**
 * Replaces variables ("$$variableName") with all the variables found in
 * story object.
 *
 * @param input
 *            String on which to apply function
 * @param story
 *            Holds the variables to substitute in ArrayList variables
 * @return The string with variables substituted by their respective value
 */
private static String replaceVariables(String input, Story story) {
        Set<String> keySet = story.variables.keySet();
        for (String key : keySet) {
                input = input.replace("$$" + key, story.variables.get(key));
```

```
        }
        return input;
}

/**
 * Extracts a string from a html node (can be text, html, cdata, attribute, ...). Multiple strings can be extracted and will be concatenated.
 *
 * @param extractionRules The extraction rule that specifies how to extract the string
 * @param root JQuery root elements
 * @param story Current story
 * @param depth Recursion depth
 * @return The string extracted
 */
private static String extractString(Element extractionRules, GQuery root, Story story, int depth) {

        if (root.isEmpty()) {
                return "";
        }

        String retString = "";
        NodeList nodes = extractionRules.getChildNodes();

        for (int i = 0; i < nodes.getLength(); i++) {
                if (nodes.item(i).getNodeType() == Node.ELEMENT_NODE) {
                        final Element node = (Element)nodes.item(i);
                        String nodeType = node.getNodeName();

                        // Extract string from attribute
                        if (nodeType.equals(Constants.NODE_EXTR_ATTR)) {
                                retString = retString + root.attr(node.getAttribute("attr"));

                        // Extract string as html from within current root node
                        } else if (nodeType.equals(Constants.NODE_EXTR_HTML)) {
                                retString = retString + root.html();

                        // Extract string as text from within current root node
                        } else if (nodeType.equals(Constants.NODE_EXTR_TEXT)) {
                                retString = retString + root.text();

                        // Insert custom text in node
                        } else if (nodeType.equals(Constants.NODE_CUSTOM_TEXT)) {
                                retString = retString + replaceVariables(node.getAttribute("txt"), story);

                        // Insert custom CDATA in node
                        } else if (nodeType.equals(Constants.NODE_CUSTOM_CDATA)) {
                                retString = retString + replaceVariables(node.getChildNodes().item(1).getNodeValue(), story);

                        // Get any string from current root node and cut it
                        } else if (nodeType.equals(Constants.NODE_MOD_STR_CUT)) {
                                String toCut = extractString(node, root, story, depth + 1);
```

```
                                String[] firstSplit = toCut.split(node.getAttribute("cutoffStart"));
                                String[] secondSplit = firstSplit[1].split(node.getAttribute("cutoffStop"));
                                retString = retString + secondSplit[0];
                            }
                        }
                    }
                    return retString;
                }
            }
```