

MASARYK UNIVERSITY
FACULTY OF INFORMATICS



Framework for Developing Offline HTML5 Applications

DIPLOMA THESIS

Petr Kunc

Brno, 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: doc. RNDr. Tomás Pitner, PhD.

Acknowledgement

Above all, I would like to thank my advisor doc. RNDr. Tomáš Pitner, PhD. for leading not only this diploma thesis but also for leading me during my studies.

I would also like to thank my colleagues in Laboratory of Software Architectures and Information Systems, especially Mgr. Filip Nguyen and Mgr. Daniel Tovarňák for priceless advice on implementation and for providing their knowledge.

Nevertheless, I would like to thank my colleagues in Celebrio Software company.

Abstract

The aim of this thesis is to provide detailed information about developing offline web applications. The thesis presents important technologies in the development and mostly deals with Application cache technology. It summarizes advantages and also disadvantages and problems of the technology.

Then, it offers solutions to some of the problems and introduces framework for building offline web applications more sophisticatedly. At last, demonstration application is presented which shows the benefits of proposed technology.

Keywords

HTML5, offline, web applications, application cache, manifest, Java 7, nio

Contents

1	Introduction	1
2	Introduction to HTML5	3
2.1	<i>HTML</i>	3
2.2	<i>CSS</i>	4
2.3	<i>JavaScript</i>	5
3	HTML5 and Offline Technologies	7
3.1	<i>Web Storage</i>	7
3.2	<i>Databases</i>	9
3.3	<i>File API</i>	10
3.4	<i>Offline detection</i>	10
3.5	<i>Application cache</i>	11
4	Application cache	12
4.1	<i>The Cache Manifest</i>	12
4.2	<i>Structure of manifest file</i>	13
4.2.1	<i>Explicit section</i>	14
4.2.2	<i>Network section</i>	15
4.2.3	<i>Fallback section</i>	16
4.2.4	<i>Settings section</i>	17
4.3	<i>Processing the manifest file</i>	17
4.3.1	<i>Downloading manifest</i>	17
4.3.2	<i>JavaScript API</i>	19
5	Drawbacks	21
5.1	<i>Changes of resources</i>	21
5.2	<i>Double reload</i>	23
5.3	<i>Modularity</i>	24
5.4	<i>Error prone lists</i>	25
5.5	<i>Files from cache</i>	26
5.6	<i>Control of process</i>	27
6	Proposed solution	28
6.1	<i>Modifications of language</i>	28
6.1.1	<i>Connection between languages</i>	28
6.1.2	<i>Import support</i>	29
6.1.3	<i>Parsing filters</i>	29
6.1.4	<i>Language extensibility</i>	30
6.2	<i>Existing filters</i>	30
6.3	<i>Processing framework</i>	31
6.4	<i>Solved problems</i>	32
7	Processing tool	33
7.1	<i>Used technologies</i>	33
7.1.1	<i>Language enhancements in JDK 7</i>	33

7.1.2	New File System API	34
7.1.3	Maven	34
7.1.4	Apache Commons Lang	35
7.1.5	Apache log4j	35
7.1.6	JUnit	35
7.2	<i>Architecture</i>	35
7.2.1	Packages	35
7.2.2	Class structure	36
7.3	<i>Modules</i>	37
7.4	<i>Importing</i>	41
7.5	<i>Filters</i>	42
7.5.1	GlobFilter and RegexFilter implementation	43
7.5.2	RGlobFilter and RRegexFilter implementation	45
7.6	<i>Execution</i>	45
8	Demonstration	46
8.1	<i>Celebrio</i>	46
8.2	<i>Demonstration application</i>	47
8.2.1	Description of the application	47
8.2.2	Offline Web Application	48
8.3	<i>Benefits of framework</i>	49
8.3.1	Application modules	49
8.3.2	Reduced number of lines	50
8.3.3	Automatization	50
9	Conclusion	52
	Bibliography	55
A	Attached CD	1
B	Example of lesscache and generated appcache file	2
C	JavaScript APIs Specification	4

Chapter 1

Introduction

Development of web technologies and web browsers quickly aims to fully support HTML5 Candidate Recommendation published by W3C [3]. One of the main topics of recommendation is a chapter about Offline Web Applications. It is possible, in most browsers, to develop a complex application which can work even offline and synchronize its data with a distant storage when the user goes online.

The term offline web application sounds like a contradiction. All web applications are dependent on internet connection and download data every time user demands a web page. Modern technologies allowed to migrate even basic desktop applications like text or spreadsheet editors, calendars, notes or books to complex web applications which allow easy cooperation, sharing, data exchange and access to important resources on multiple devices.

Users quickly adopted this trend and use applications like Google Drive¹, Office 365², Dropbox³ and many others. As internet connection became more accessible by mobile internet access, people moved their data and even everyday tools on the web. But this approach has also its disadvantages. When loss of internet connection happens, users are completely cut from their data and programs. While travelling people cannot even create a presentation because they rely on web applications. A click in bad time when a person is on a train and their train enters a tunnel and computer is out of signal can lead to a loss of their several hours work [13].

Computer would become a useless machine when there is no internet connection. One solution would be to keep desktop applications too and work with them when access to web is not possible. This solution creates redundancy, a need to work with more tools than necessary and users lose all the advantages mentioned above.

HTML5 offers a set of tools developers can make use of and create applications which are not dependent on stable connection. These new tools can be capable of automatical detection of online/offline status, they can also store data on computer's file system and in local browser databases and even automatically synchronize data when computer goes online.

This thesis focuses on describing how the offline technologies and techniques work with a main focus on application cache technology. Application cache uses manifest files to determine which files should be stored in browser and therefore reachable offline [23]. It points out the pro et contra of appcache and also proposes own solution how to overcome the

-
1. <<http://drive.google.com>>
 2. <<http://office.microsoft.com/>>
 3. <<http://www.dropbox.com/>>

greatest disadvantages of this technology.

Theoretical part of this thesis is a little longer than usually is as the technologies and syntax is not well known. Therefore, the technology, its advantages and problems, are thoroughly introduced. Chapter two describes arrival of HTML5 and its features. Third chapter focuses on offline technologies available in HTML5. Next chapter describes the main topic of this thesis – Application Cache. In chapter five, drawbacks and problems of development using Application cache are mentioned. Following chapter proposes a solution to the mentioned problems and chapter 7 shows architecture and implementation of the framework, which was developed as part of this thesis. Last chapter introduces a demonstration application showing results of the framework use.

Framework is comprised of language extending possibilities of current implementation and processing tool – Java application parsing this language.

Chapter 2

Introduction to HTML5

A brief overview of HTML5 will be presented in the beginning of this thesis. As this topic is vastly covered in many other publications, it will be introduced just to span the very basic terms and link to proper books and articles. New HTML5 features can be divided into three main categories: those connected with markup, styles and scripting.



Figure 2.1: The W3C HTML5 logo

2.1 HTML

HTML (Hypertext Markup Language) is a markup language mostly used for webpages. It is about to enter its fifth revision of the standard and in December 2012 it moved from Working draft to Candidate Recommendation phase. Developers use the word HTML5 even when they refer to standards which are not parts of HTML5 standards as CSS3, Geo Location, WebGL and many others.

The history of HTML5 is quite complicated. HTML was supposed to be an SGML (Standard Generalized Markup Language) language in the beginning. HTML 4.0 completely fulfilled ISO 8879 - SGML and became part of SGML family in 1997. In 1998, the W3C stopped working on HTML and focused on XML-based XHTML, which was supposed to be more strict than its predecessor.

XHTML was supposed to clean the original HTML and enforce stricter rules. It was a helpful quality of XHTML as it helped coders to find minor bugs which could cause inconsistency among browsers. As XHTML was heavily XML influenced, some more ideas and

benefits were supposed to become true: interoperability, easy page processing, portability and extensibility, but these features never fulfilled the expectations [20].

The main problem of XHTML was that no browsers enforced the strict error-checking so developers could keep creating not valid codes. If the code contains errors, browsers switch to a quirk mode which handle the problems as the browser wants. The solution should be XHTML2 which forced the browsers to reject invalid pages. It did not work in the end as developers were urged to change every existing page of theirs, to support the standards.

The experts from Apple, Opera and Mozilla formed their own group WHATWG (Web Hypertext Application Technology Working Group) which started working on their own extension of HTML as they considered XHTML2 to be too strict and does not bring the tools the developers want and need. Later, W3C dismissed XHTML2 group and started to cooperate with WHATWG on standardization of HTML5 [29].

HTML5 is much looser than HTML used to be. There is no need for elements like `<html>`, `<head>` or `<body>`, the specification does not require them. Tags and attributes are case insensitive and attribute values do not have to be in quotes. The key principle of HTML5 is “Do not break the Web!” which means – fully support everything what worked until now and do not enforce strict rules where it is not necessary but just gives recommendations (lower case, use the elements mentioned above and so on).

The biggest value of HTML5 is its great support for modern web applications [10].

- HTML5 supports semantic elements for structuring a page like `<article>`, `<section>`, `<figure>`, `<header>`, `<footer>` and more.
- It has better support for forms like auto-completion, new types of inputs for automatic checking and to support mobile devices,
- `<audio>` and `<video>` elements to play multimedia without a third party. HTML5 also describes the use of SVG in web applications.
- `<canvas>` to support drawing in web browser to create games or flash-like animations.
- Support for microformats, own `data-` prefix to connect data with some element, drag and drop, ARIA attributes and many others.

2.2 CSS

Cascading Style Sheets (CSS) is a style sheet language very often used for styling web pages. As HTML code covers the semantic part of data, style sheets describe how the data should be presented. CSS allows designers to define how parts of the page should be displayed: fonts, sizes, positioning, colors, margins, borders etc.

As the web developers need more and more tools to cleanly visualize content and also adapt their pages to new devices and mobile phones, CSS definitions and standards grow.

CSS3 is divided into separate standards called modules which made possible to develop the most interesting and important modules first in iterations. There are over fifty modules from the CSS Working Group and four of these are now formal recommendations and some others are stated to be Candidate Recommendations [6]. CSS3 is not part of HTML5 but is strongly recommended by W3C to use them along.

Some of the most important enhancements in CSS3 are:

- Selectors are improved so designers can select every even or odd element, they can be much more specific as CSS3 adopted selectors very similar to those jQuery [footnote!] use.
- Media queries allow developers to change their styles or even stylesheets according to what device displays the page. It can be based on device type, orientation or resolution.
- By setting `@font-face`, designers can finally link their own fonts so they are no more restricted to a set of “web safe” fonts. CSS3 supports several font types like woff, true-type or svg.
- Many enhancements to styling and positioning were made. Backgrounds, borders, columns, box modeling, rounded corners, colors, shadows, opacity, transitions and transformations were improved so it is much easier to create almost any page developers want and by using graphic card hardware acceleration powerful animations can be created by using just CSS.

2.3 JavaScript

JavaScript is a dominant programming language in web browsers. It is the third of fundamental web technologies. HTML for content, CSS for presentation and JavaScript to specify the behavior of web applications. JavaScript can be used for input operations, interacting with user, asynchronous communication, animations, DOM and style manipulation, event handling etc. [18] JavaScript has almost nothing in common with Java (except C inspired syntax). It is a dynamic, untyped, interpreted language and supports object-oriented as well as functional programming style and is usually hosted in web browsers.

A short list of some JavaScript features connected with HTML5:

- **Web Workers** interface makes possible to create threads in JavaScript. Web browsers execute all JavaScript in only one thread so when developer needs to execute a computation heavy operation the browser will stop responding to input and output operations and stop drawing animations or any other changes on the page. Web Worker is a background operation in separate thread which cannot directly work with DOM (Document Object Model) but can communicate with the main application by sending and receiving messages.

- **WebSocket** interface finally enables creating two-way communication channel over one HTTP socket. Its main usage will be in real time communication applications and in event driven programming. Developers can now stop using long polling techniques like Comet or AJAX to create responsive applications [5].
- **Native Drag and Drop.** HTML5 standardizes a feature which developers used by external libraries. Dragging in the pages became very common as desktop programs started moving to the web. The standard is based on original implementation in Internet Explorer 5. Drag and drop is still more popular with external solutions like jQuery or Prototype as the native HTML5 solution is quite confusing and complicated but is expected that these libraries will implement the native dragging on their lower level to take advantage of acceleration [9].
- **Geolocation** is not part of HTML5 standard but is now a proposed recommendation by WHATWG and W3C. Geolocation is a feature that let the application find (after explicit permission) where the device is located. The position can be pinpointed by IP address, wireless access point location, cellphone triangulation or GPS. Geolocation API always returns data about latitude, longitude and accuracy and some devices or service providers can also provide data about altitude, heading or speed [29].
- **History API.** URL (Uniform resource locator) was created to link (and identify) a resource (most often a web page) [30]. When the content of web was static, it was quite a simple task. Techniques of AJAX and more exactly XMLHttpRequest object allowed to create applications which silently (in background) downloaded data and improved the experience of working with web. Huge disadvantage is that the URL of the page does not change and makes bookmarking and other tasks a big problem. History API enables the developers to change the address in browser without a reload and work with history stack.

Chapter 3

HTML5 and Offline Technologies

Caching data in web browsers is nothing new. The majority of web browsers uses some kind of cache to store often needed data as images, stylesheets or scripts. HTML5 gives developers an advantage of manual and explicit control of certain type of browser's cache to clear the way for creating webpages capable of running offline. HTML5 also goes much further and offers technologies as Web Storage, web databases and sandbox file system, which will be introduced in this chapter.

Creating an offline web application also brings one related advantage: The application will almost certainly run much more smoothly as less data has to be exchanged between client and server and data synchronization can take place in background of user's working process.

Main reasons why develop web applications capable of working offline were mentioned in the introduction of this thesis. Complex applications can benefit from these technologies by lowering latency, saving bandwidth and deliver much more satisfying experience for users with mobile or unreliable connection. This chapter takes a closer look to techniques and technologies which can be used for creating such experience.

3.1 Web Storage

Applications (desktop or web) need to store data. Desktop applications can store data on the machine by saving it in local files (.conf, .ini, .dat, .xml or any other developers do design), registries and embedded databases, or developers can implement almost any solution that comes in their mind.

Web applications were designed to store all data on the server and just visualize the data on client. Cookies were invented to remember the state of the website and can be used as a persistent storage but they are limited to 4 KB of data and are included in every HTTP request to related site, which unnecessarily loads the network and can be compromised.

First possibility to store data right in the browser without cookies brought Internet Explorer with its JavaScript object `userData` which managed to save up to 64 KB and even 640 KB on trusted domains. LSOs (Local Shared Objects) in Flash could keep up to 100 KB per domain. Google Gears offered API to an embedded SQLite database and some more solutions were established (Adobe AIR, Silverlight) as developers desired a tool to store data. All of these solutions are either platform dependent or third party plugins. HTML5 standard solves these difficulties and offers unified solution [29].

Webstorage interface is the standard which tries to solve this inconsistency. It is widely supported even in nowadays browsers (even Internet Explorer which usually adopts new technologies only when they are declared as official standard supports this feature since version 8).

The interface is really simple as it stores data as an associative array (in terms of Java it reminds Map) – key-value format. The official interface proposed by W3C looks like this [32]:

```
interface Storage {
    readonly attribute unsigned long length;
    DOMString? key(unsigned long index);
    getter DOMString getItem(DOMString key);
    setter creator void setItem(DOMString key, DOMString value);
    deleter void removeItem(DOMString key);
    void clear();
};
```

Web Storage interface is in every browser implemented by two JavaScript objects:

- **localStorage** object manages data which is supposed to be stored permanently unless users clear the browser cache.
- **sessionStorage** object manages temporary data supposed to be remembered only for current session – one window or tab. After the end of session data vanishes.

As both objects implement the same interface the only difference is how long the data should remain. The data is stored for each website domain and other browsers or even users cannot manipulate or obtain the data. JavaScript syntax is quite benevolent and all objects are traversable so the next example of getters and setters are equivalent. This example checks the availability of feature and then sets and get the data from local storage:

```
if (window.localStorage) {
    //these three examples of setting data is equivalent
    localStorage["key"] = "data";
    localStorage.setItem("key", "data");
    localStorage.key = "data";

    //these three examples of getting data is equivalent
    output = localStorage["key"];
    output = localStorage.getItem("key");
    output = localStorage.key;
}
```

Example 3.1.1: Web Storage – getting and setting data

The data stored in Web Storage is saved in web browser and can be accessed any time even when the browser is offline and is trying to access some web page (techniques how to

save whole pages, images, styles etc. will be described in next chapter [Chapter 4, “Application cache”](#)), which makes this technology a good use to store dynamic data on client’s side and synchronize it when the internet connection is available again.

Storage quota is not determined by standard but most browsers’ limit is 5 MB per origin but some browsers allow users to permit applications to exceed the limit.

3.2 Databases

The Web Storage is useful for storing small amounts of unstructured data but when developers need to emulate storage capabilities of server there is a better solution.

There were two standard proposals in the beginning. The Web SQL Database and Indexed Database API (IndexedDB). The Web SQL Database was meant to create databases and query them with a dialect of Structured Query Language but now is not furthermore developed and standardization process stopped. The official disclaimer of W3C: “This document was on the W3C Recommendation track but specification work has stopped. The specification reached an impasse: all interested implementors have used the same SQL backend (Sqlite), but we need multiple independent implementations to proceed along a standardisation path. [31]”

IndexedDB introduces an object store – mechanism by which data is stored in the database. IndexedDB is not a relational database and is very similar to NoSQL server databases or more precisely object-oriented databases. IndexedDB does not use Structured Query Language but the object store has methods to create cursors or ranged queries. IndexedDB makes possible to store whole JavaScript objects (in not relational schema) and create indexes and sorting on them. The API is completely asynchronous that means that developers does not obtain data but they only request data and when the data is ready callback function is executed to process it – it reminds of concept of XMLHttpRequest (AJAX) objects. [14]

```
//standard synchronous data access
//usually developers obtain data by assigning
//a returned value of some method to a variable
result = database.get("key");

//Indexed DB asynchronous request:
var request = objectStore.get("key");
request.onerror = function(event) {
    //when IndexedDB fails when obtaining the key,
    //it triggers the onerror function
};
request.onsuccess = function(event) {
    console.log("Value of key is " + request.result.value);
};
```

Example 3.2.1: Example of asynchronous retrieval of data

The IndexedDB can currently use up to 50 MB per origin but user interface will just ask permission for storing blobs bigger than 50 MB. [21]

3.3 File API

JavaScript was not supposed to access any files on a computer because of security concerns. HTML5 (or more precisely standard related to HTML5) brings a standardized API to read and write files in a secure way. Read and write of files can be made to a sandbox inside of computer's file system so applications can read only files which were programatically stored. The API does not have a direct access to the standard file system but users can select file(s) (in a dialog window or by drag and drop) and these files can be accessed by the API.

The specification declares an interface of `FileReader` which is capable of reading the contents of files. There are four options of reading:

- `readAsBinaryString`
- `readAsText`
- `readAsDataURL`
- `readAsArrayBuffer`

The reading is asynchronous and `FileReader` informs about changes by triggering events [12].

File API can be used for unzipping a file or for using any file imported in the application (and the file does not have to be uploaded to a server), e.g. creating thumbnails or verifying mime types. When the user is offline and wants to upload a file, API can easily save it to the sandbox and attempt the upload when connection is working. Any files can be saved in the sandbox file system and accessed while offline.

3.4 Offline detection

Developers often need to find out, whether browser is online or not. JavaScript defines attribute `onLine` in object `navigator` recognizing connection of the computer. It also defines two events (`offline` and `online`) which can be listened on object `window`. The update occurs on click on a link or when a script requests a page.

The problem is that browsers implement this behavior differently and uselessly. Firefox and Internet Explorer sets `onLine` to false when the browser is in offline mode, true otherwise. Chrome and Safari (Webkit core) sets offline flag when browser is not able to reach Local Area Network or router, therefore when router is working but internet provider is off it sets the attribute `true` and developer gets false positive. [22]

This is the main reason why JavaScript developers created their own solutions to detect whether browsers are offline. Very simple solutions are detecting an error state of Application cache (see [Chapter 4, "Application cache"](#) for more details).

```
window.applicationCache.addEventListener("error", function(event) {  
    //if the developer is sure that manifest exists  
    //then the browser cannot reach the web and therefore is offline  
});
```

Example 3.4.1: Example of detecting offline browser by Application Cache

The solution above is sufficient when the developer needs to check online state on load of the web page. If the checking is needed even after, developers have to use other techniques. The best known are polling with XMLHttpRequest or using WebSockets. The application sends a request every e.g. 30 seconds and defines maximum timeout. If the response does not come in the timeout, application is probably offline or the server is off. These states are almost equivalent as the site is not reachable [7].

3.5 Application cache

This thesis is mainly focused on the application cache and that is why it gets its own (following) chapter.

Chapter 4

Application cache

In the previous chapter, this thesis discussed possibilities of holding data in various storages but did not discuss how to store the web page itself. Every offline web application concentrates on a cache manifest file. The manifest file is a list of all the files the application has to download and store and also a set of rules how the application should behave when it reaches an unavailable resource. The application cache technology is officially named *Offline Web Applications* but this thesis will refer to it as Application cache because the term Offline Web Applications is interchangeable with the applications capable of running offline which can use other technologies mentioned above.

4.1 The Cache Manifest

The manifest file is a cache file which has to be set up and can be controlled explicitly by authors of the application. This file has to be linked by `manifest` attribute on `<html>`.

```
<!DOCTYPE HTML>
<html manifest="manifest.appcache">
    ...
</html>
```

Example 4.1.1: Linkage of html document and manifest file

Every page which is supposed to be accessible offline has to link the `manifest` attribute with cache manifest file. Cache manifest file has to be on the same domain and can have any filename extension. (The most common are: `.appcache` and `.manifest` and the first one is recommended as the second one is an existing extension in Windows environment.) The only mandatory setting is: The web server has to serve manifest files with the MIME type `text/cache-manifest` [23]. [Example 4.1.2](#) and [Example 4.1.3](#) shows how to set up Apache and IIS servers.

```
AddType text/cache-manifest .appcache
```

Example 4.1.2: Adding a directive in `.htaccess` file for Apache

4.2. STRUCTURE OF MANIFEST FILE

```
<mimeMap fileExtension=".appcache" mimeType="text/cache-manifest" />
```

Example 4.1.3: Adding a directive in `web.config` file for IIS

The Application cache technology is now widely implemented by browsers but is not supported in some important browsers like Internet Explorer 8 and Internet Explorer 9. Windows XP is still very popular operation system in 2013 and as Internet Explorer 10 is not supported on this OS, its users will not be able to work with the web pages offline. [Table 4.1](#) shows the minimal versions of the most common browsers, which support the Application cache.

Browser	Since version
Internet Explorer	10
Firefox	3.5
Chrome	4
Safari	4
Opera	10.6

Table 4.1: Browser support for Application cache technology [\[4\]](#)

Different browsers applies different limitations of size of cached files [\[20\]](#):

- Android browser has no limit.
- Safari Mobile can use up to 10 MB.
- Desktop Safari has an unlimited storage.
- Mozilla Firefox limits applications to 50 MB and users can increase the limit when prompted.
- Google Chrome creates a shared pool for application cache. The limit can be set up to gigabytes.

4.2 Structure of manifest file

As mentioned above – the manifest file has to be served with certain MIME type. Also, the file has to be UTF-8 encoded and must keep specific syntax. The first line of every manifest file has to be

```
CACHE MANIFEST
```

It is a compulsory requirement (BOM character can occur in the beginning of the file). Any of the subsequent lines must be one of the following:

- **Blank lines** are ignored.
- **Comment lines** start with the hash character (#). The preceding characters can and must be whitespaces only. Authors are not allowed to add comments in the same line as resources or other sections. The comment line must be on their own line.
- **Section headers** change the current section of manifest. There are four possible headers which will be introduced in a moment. Section headers can appear multiple times and their content can be empty.
- **Section data.** Its syntax depends on section the manifest is currently in.

```
CACHE MANIFEST
#VERSION: 2013-04-18 16:45
#the explicit section is default, no need for CACHE:
CACHE:
styles/default.css
scripts/main.js
http://code.jquery.com/jquery-1.9.1.min.js
FALLBACK:
/images/avatars/ images/offline.png
/ /offline.html
NETWORK:
*
SETTINGS:
prefer-online
```

Example 4.2.1: Example of complete manifest file.

4.2.1 Explicit section

As mentioned above, there are four sections defined in the standard of Application Cache and can occur in any order. The first and most important one is so called explicit section. It is a default section of the manifest file so after `CACHE MANIFEST` line this section is automatically set. However, developers can explicitly set this section by line

`CACHE:`

The explicit section lists all resources which should be accessible offline. They are cached after they were downloaded for the first time and are never deleted until the user deletes his cache. The HTML file referencing the manifest file is automatically cached (it is called master entry) and does not have to be listed in the manifest file¹.

1. but it is strongly recommended when developers have many documents and they want to download all of them at once and not only when users access the document with `manifest` attribute in `<html>` element. The

4.2. STRUCTURE OF MANIFEST FILE

Every resource listed in the explicit section must be a valid URL, either relative or absolute, and it can be even cross-origin unless the manifest file is obtained over SSL (TLS) protocol. Response of each URL must return a valid resource and status code needs to indicate success. When any URL responds with 4xx or 5xx status, all downloaded and cached data are deleted, the caching process fails and no data is accessible offline [20].

```
http://example.tld/path/to/resource.html
path/to/resource.html
/path/to/resource.html
```

Example 4.2.2: Example of possible references in explicit section.

4.2.2 Network section

Network section lists all resources which are meant to be accessed online. They are called “white-listed resources” that require a connection to web server. The records in the manifest file can be either URLs as described in Section 4.2.1 or URL prefixes. The prefix tells browser that any resource starting with this prefix should not be cached or looked up in the cache but accessed on server. The network section is introduced by line `NETWORK :` [13].

```
NETWORK:
# login page should not be cached
# credentials need to be sent to server
login.php
# forbid to cache any resources in authorized section of web
/authorized
# access to any API URL has to be online
http://api.twitter.com
```

Example 4.2.3: Example of possible references in network section.

The last possible form of entry in network section is an asterisk character (*). It tells browser that any URLs which are not listed in cache manifest should be accessed online. When the asterisk sign is not used, the online whitelist wildcard flag is set as blocking. It means that any resource not listed as cached should be treated as unavailable and not be accessed. The asterisk sign sets the flag as open. Any resource not cached should be treated as listed in the network section.

Suppose there is an application which allows users to upload videos and the developers do not want to cache them because of the required size. The manifest file cannot include an URL prefix path network rule like `/videos` because each user can upload any type of data

application cache downloads only the master entry and resources listed in the explicit section, so some other document which is not listed in the section but have the attribute set is added to the cache only when user browses to it.

and they are stored in directories belonging to users like `/usr/user1` and developers want to cache text and image files. If the asterisk sign is omitted, the videos will not be accessible even when users are online. For the first time they will work ordinarily but on page reload application will load data from cache and the video will not be mentioned in explicit nor network section and the application will not even try to load it online. It completely ignores all uncached files throwing not found error (even when the resource is on the web server). This is why the asterisk character is needed – it tells browser cache to load any URLs not matching the cache files from server with the original URL.

4.2.3 Fallback section

The explicit section defines which resources are supposed to be cached and the network section points out all URLs which should be always accessed online. The fallback section allows to substitute some resources depending on whether the browser can obtain the file online.

Unlike the sections mentioned above, the fallback section (beginning with line `FALLBACK:`) syntax differs slightly. Two resources have to be mentioned on one line separated by at least one whitespace character. The first name defines a single resource (or more using the prefix pattern like in [Section 4.2.2](#)). When it is not accessible online browser should fallback to the second resource and load it. Every resource in the fallback section has to keep same origin policy.

When using prefix patterns, fallback namespaces are created. Every URL which falls in the namespace will be routed to the second file name when offline. Each namespace cannot be mentioned more than once but the namespaces can coincide. The most specific namespace is chosen by the specification.

```
FALLBACK:
#everything from /images/avatars prefix link to default image
/images/avatars/ /images/avatars/default.png
#every other resource from /images/ prefix link to offline.png
/images/ /images/offline.png
#any URL not matching the two above link to offline.html
/ /offline.html
```

Example 4.2.4: Example of fallback namespaces.

The web browser automatically caches the fallback file and there is no need to list it in the explicit section. Some browsers allow developers to use the asterisk character for more specific URLs (as `*.html`) but in April 2013 this syntax is not part of the official standard [\[29\]](#).

4.2.4 Settings section

This section is used for changing preferences of the cache manifest. There is currently only one setting defined. The setting sets a flag which determines in which mode the cache is. The values are `fast` or `prefer-online` and the first one is default [2].

The `prefer-online` setting tells the browser to load master resources from a web server even when the file is cached. If there is no connection, it will use the cached file.

The `fast` mode tells the browser to use the cached file even when there is a working internet connection and the data could be fetched from the server.

The first one is more useful when cached data is changing often and when data is sent from web server with the semantics and logic. The master resources are loaded online when possible and explicitly named resources are loaded from application cache as from atomic HTTP cache, while the fast mode is recommended when developers want to always load templates from cache and load data dynamically when possible.

4.3 Processing the manifest file

This subchapter discusses how the process of downloading files to cache works and what API web browsers provide to interact with.

4.3.1 Downloading manifest

When the browser observes that `manifest` attribute is set in the document, it triggers the `checking` event, no matter whether it already has any resources downloaded and checks the server for the related file. When the checking event ends, the browser is informed whether the application cache is already stored.

If it is not, the browser downloads the initial version and files mentioned in the manifest file and triggers `downloading` event. While downloading `progress` events are fired periodically which include information about number of total files to download and number of already processed files. After all files have been successfully downloaded, the `cached` event is dispatched and the process ends.

In case the browser has already saved the cache file in its storage, the change of manifest file is checked. If the change did not occur, the browser triggers `noupdate` event, terminates the process and no resources are downloaded even if they have changed. In the other case, browser redownloads the resources with standard `If-Modified-Since` headers to avoid fetching files which did not change since last time and sends the same `downloading` and `progress` events like the file is downloaded for the first time. When done, browser raises `updateready` event and successfully finishes [2].

There are two more events that can be triggered by a browser. `Obsolete` event is fired when the HTTP response status of manifest file URL is 404 (Page Not Found) or 410 (Permanently Gone) and this manifest is already stored. In this case the application cache is being deleted [23].

4.3. PROCESSING THE MANIFEST FILE

The second event is called `error` and can be triggered in four cases:

1. The manifest file HTTP status is 404 or 410.
2. The manifest was found but did not change but the web page which referenced the manifest file could not be downloaded properly.
3. When any of the resources listed in cache manifest returns error status and cannot be cached.
4. When the manifest file has changed on the server while the resources have been downloaded.

If anything above happens, whole process is aborted and the cache is cleared. [20]

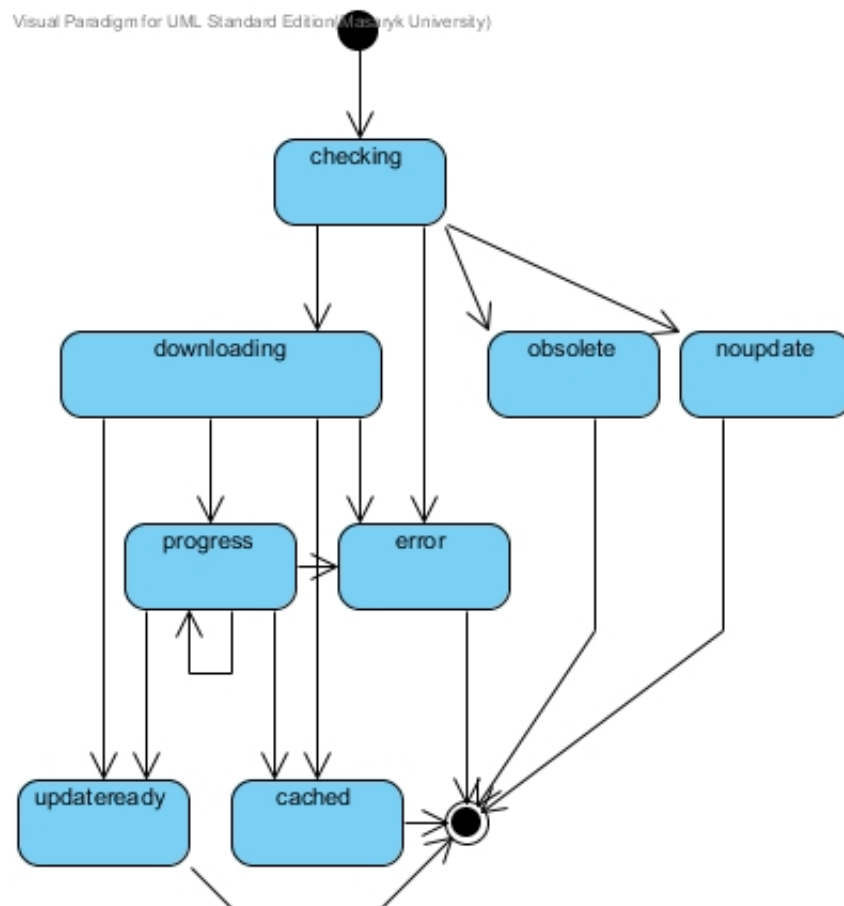


Figure 4.1: UML state diagram of events in appcache

4.3.2 JavaScript API

If browser supports the offline web applications, there exists a `window.applicationCache` object in JavaScript data model. This object provides basic methods for updating the cache manually and to notify web developers about changes and statuses during the process. All events described above are dispatched on this object. Complete WHATWG specification interface can be found in [Appendix C](#).

Application cache API provides event handlers which can be linked with the events described above. All events implement standard `Event` interface, only `progress` event implements `ProgressEvent` which provides attributes to detect total and already loaded number of entries [23].

There are three methods which can be used for programatically operating the process of caching.

Developers can use the `abort()` method to manually stop the process of downloading. There can be a GUI button for users to block the caching when bandwidth or amount of downloaded data must be taken into account.

The `update()` method can manually invoke the downloading process of application cache. Modern web applications can easily be opened in a browser for months without a single reload and communicate with API only by AJAX requests or using web sockets and provide users fresh data. This method can be used for manually checking for any updates of the resources of application and facilitate updates of the page.

Even if new resources are downloaded they are not used by the application instantly. The `swapCache` method must be invoked to enable access of the application to these resources. However, this method does not reload any content already processed by the browser – images do not reload and style sheets are not reparsed, browser uses the new cache for any following requests. Whole application has to be refreshed to obtain new version [1].

```
//shortcut
var appCache = window.applicationCache;
//manually invoke update
appCache.update();

appCache.addEventListener('updateready', function(e) {
    // update is ready
    window.applicationCache.swapCache();
    // or reload page (user's confirmation is appropriate)
    window.location.reload();
}, false);

appCache.addEventListener('progress', function (e) {
    //In case browser does not support counting of files
    if (e.lengthComputable) {
        notify("Downloaded "+ e.loaded +" of "+ e.total);
    } else {
        notify("Downloading");
    }
}, false);
```

Example 4.3.1: Demonstration of typical use of JavaScript API

Chapter 5

Drawbacks

The application cache technology brings the main advantage – web applications are capable of running offline. There is also a great advantage that is often used by developers but which can very easily turn into disadvantage: Developers use the manifest files for speeding up the application. Cached resources are local so they can be loaded much faster than those served by servers. Also server load is reduced as only changed resources are downloaded.

This chapter goes deeper into concept of application caching and describes problems developers can encounter. The most problems developers encounter are not bugs in implementation or problems in the standard but intended behavior of the application cache. It tries not to download or access any resources until they are really needed.

5.1 Changes of resources

The problem number one is the manifest file (and all resources included in it) are redownloaded only if the manifest file has changed. It was mentioned in the previous chapter but probably was not noticed by the reader properly. This leads to two pitfalls associated with this behavior.

The first pitfall is in checking the cache manifest file itself has changed. Developers have to bear in mind that the manifest file is a resource served over HTTP as any other accessed on the web. The web server sends HTTP response headers defining how the browser should treat this resource. `Expire` and `Cache-Control` headers tell the browser how this resource should be cached. The manifest file itself should not be cached, otherwise even when developers have changed it on the server a new version will not be downloaded.

If the manifest file has expired, the browser requests the resource containing date of last modification of it, which browser got in response header when downloading the resource last time. Then, the web server compares dates of last modifications and if nothing has changed it will return an empty response containing status code 304 – Not Modified.

If the change occurred, the web server sends the resource in the body of HTTP response and also sends OK HTTP status code (200) and also headers with cache information and new date of last modification. The web browser checks the contents of the new manifest file against the old one and redownloads the resources listed in the manifest only if the contents have changed.

The developers should reconfigure the web server to ensure that the cache manifest file is not cached itself and the browser always gets a fresh copy. For developing and debug-

ging it is absolutely crucial. Suitable setting for the manifest files is `must-revalidate` or `no-cache` [8].

The second pitfall associated with the changes of manifest file is that just an update on server of cached resource is not sufficient to tell the browser to redownload this resource. For example, when developer changes an image which is used as background of the web application, the web browser will not notice this change and keep the old one in its cache. The reason is again: The manifest file has not changed on the server. Every time a change is made to any resource in the offline web application, the cache manifest also has to be changed. This behavior is intentional – the browser would need to send an HTTP request for every cached resource every time user accesses the web page to determine any changes in them. This is why only change to manifest file matters as browser can check changes of a single file.

The easiest way how to overcome this obstacle is to include a version comment in the manifest file. Suppose there is a manifest file:

```
CACHE MANIFEST:
# v.1
image.png
```

When the image has been changed, developer has to make any change to the manifest file. For example change a comment.

```
CACHE MANIFEST:
# v.2
image.png
```

The other more complicated way is to change URLs of updated resources on the server and make use of far-future caching [11]. The far-future technique was created to speed up applications by using standard HTTP caches. The core principle of this technique is that the contents of one resource never change. For example scripts get their unique name (like `script-1.0.0.js`) and when change is needed a new file with unique name is created (like `script-1.0.1.js`). This has several benefits:

- The resources can be cached forever – expiration can be set in *far future*, this is where this technique got its name.
- The older versions are always accessible (which can be important when other people use the script as a library and newer version is not compatible).
- Developers do not forget to change the manifest file as they do not want to update the original resource but cache a completely new one so they need to change the URL in the manifest. This has also one more advantage which will be introduced in [Section 5.5](#).

However, the technique is applicable to static resources where changes occur rarely. Some resources are updated often and cannot change their URLs (html files, documents dynamically created on web server, etc.).

Far-future caching technique should never be used on manifest files. The reason why not use this technique on cache manifest files is that the application would never access any newer version. The following [Example 5.1.1](#) describes what the reason is.

In this application a manifest file `manifest-v1.appcache` exists. User browses to a document which links this file. The application cache downloads the manifest file and all resources listed in it. Later, developer has made changes to the manifest file and created file `manifest-v2.appcache` and changed every document pointing to the original appcache file to the new one. When user visits the page for the second time, the browser uses the cached version where the `manifest` attribute links the old file. The browser checks whether the original manifest changed and triggers `noupdate` event as the manifest file has not changed since last time and the update process ends. The browser will never notice the new version of manifest file as long as the document stays cached.

Example 5.1.1: Far-future caching manifest files

5.2 Double reload

The diagram [Figure 5.1](#) [24] shows that even when the browser gets updated resources and store them in the cache the user will not access them until next refresh of the page. This behavior is also wilfull as the update is performed in the background and the not updated resources were already served to user from cache. It happens only when cache is set to the `fast` mode and data is retrieved preferably from cache and not from the server. The browser would have to reload the resources while users work with the application which would be confusing and could lead to unexpected problems in it. This was described in [Section 4.3.2](#).

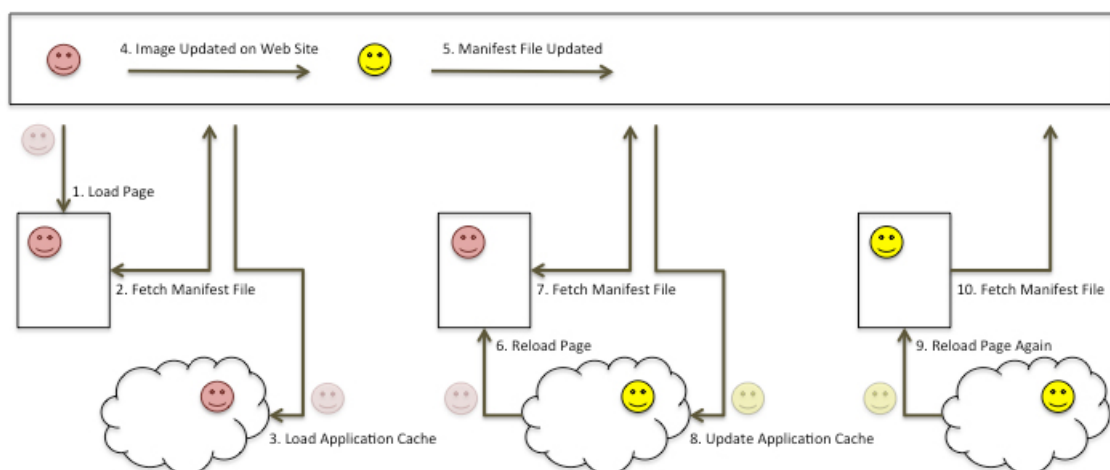


Figure 5.1: Diagram of fetching data by application cache [24]

This style of processing data becomes problem when some resources are accessed always online (Network section). When new version of application is deployed and API serves new format of data or other properties, developers will also update JavaScript files processing this data. However, if network connection is working, for the first time the application fetches new data from API but stale version of script file is accessed from application cache which leads to inconsistency and can lead to fatal errors ([Example 5.2.1](#)).

The API in last version served time data in localized string format containing date and time as two values but developers changed it to provide general timestamp and process it on the client. When user visits the page again after this change of API and is online, new format is obtained from the server and new version of script file is fetched by the application cache downloading mechanism but older script was already got from the application cache. This script is not capable of parsing the timestamp and tries to access non-existing properties causing errors in JavaScript. Date and time is not displayed as should be and application behaves oddly. After page reload when new version of script file is loaded into application, everything works as is supposed to.

Example 5.2.1: Demonstration of double refresh problem

The only possible solution for this problem is to version APIs and provide compatibility for older versions. This is not always possible as the APIs can be developed by third parties. The partial solution is to develop scripts which are very robust and are capable of dealing with errors and create fallbacks in case of problems. The `prefer-online` setting does not help as script files are explicitly stated in the application cache and the setting applies only for master entries ¹.

5.3 Modularity

Each document can include only one manifest file. Other resources like stylesheets or scripts use their own elements in HTML document and can be linked anywhere in the application and even dynamically by JavaScript. The only possibility how to link the manifest file is an attribute in `<html>` element.

The application cache manifest is meant to be only one for whole domain. This document should contain all resources which can be possibly needed while user is offline. But if the application is too complex and consists of many modules, one cache document can be too huge and even exceed quota. Solution would be to create separate files for separate modules which can be downloaded when needed. Developers could easily control which resources are downloaded only when user really accesses provided tools and do not stash not needed files.

The problem comes when resources, e.g. logos, scripts, styles, libraries, are spread over whole application. Application cache technology does not allow any kind of imports of other

1. Review: Master entries are documents added to cache because they have manifest attribute set in the HTML document and user navigated to it.

documents, so the shared resources need to be listed in every cache file. In case the application has 50 modules and new version of library exists, developers would be forced to change every manifest file listing this library. Sustainability would become a problem as creators would have to maintain too many files.

5.4 Error prone lists

Section 4.3.1 claimed that an error event is raised when any of the resources listed in cache manifest returns error status and cannot be cached. If even a single resource mentioned in the cache manifest is not found or returns any HTTP error response, whole process fails. All downloaded entries are thrown away and nothing is cached. Error event does not include any identification of what interrupted the process which can make debugging process troublesome. Fortunately most browsers facilitate this process by logging the events implicitly in the console, although any mistyped character in the long list will break the process of running the application offline.

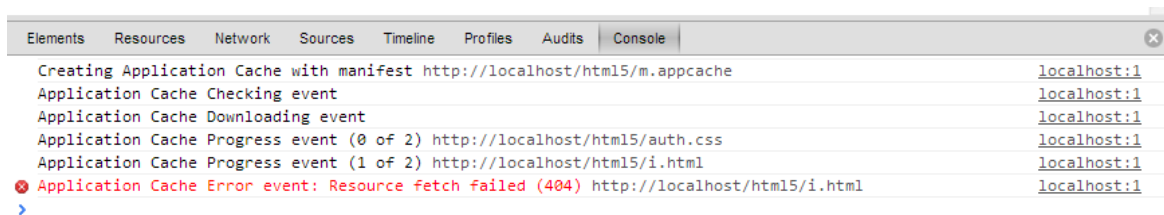


Figure 5.2: Printed screen of console log in Google Chrome

Specification does not support wildcards or prefix matches in the explicit section. Fallback and network sections support prefixes to determine which resources should be redirected to offline available items or always accessed online but the explicit section can define only one entry per line. This is also intended behavior as the creators of specification do not want to allow developers to cache whole vast webpages with just a single line of code [20].

It is also impossible as the manifest is completely processed by browser which is then downloading the resources over HTTP and browser does not have any information about contents of directories of the web server. The manifest would need to be served with some logic on the web server which would walk through directories and generate a new file to send to the browser.

This becomes a problem when developers need to cache dynamically changing directories. A good example is a gallery of pictures. Every time anybody changes the contents of the directory, manifest file would need to change. When a photo is deleted, developers need to change the manifest, otherwise the caching would interrupt as the resource would not be available anymore. When a photo is added, creators must add the file manually to the manifest or it will not be stored in the computer offline.

Also when a creator of the web application wants to cache a hundred of small icons, he must list one hundred items in the manifest document despite the fact that he knows exactly what he is doing and the quota will not be exceeded and he does not want to cache every document of the huge application. Web architects who would do not want to abuse the function of wildcards, would appreciate reduced complexity of creating list of files needed to be accessible offline but manifest itself cannot solve this problem.

5.5 Files from cache

It is impossible to tell browser not to cache the page which referred the manifest file. When the page is dynamically created, developers probably do not want to cache its content offline but they want to allow to cache static content of the page (images, scripts, etc.). The document itself cannot be excluded. This disadvantage is regularly overcome by including an invisible iframe element in the web page which refers to the manifest. This way only blank page in the iframe is cached with resources listed in the manifest [24].

However, in most cases developers want even the dynamically created page to be accessible offline with old data and when user is online, display new data it. The iframe solution does not resolve this issue. The dynamic page would not be cached at all. This is why specification came with the `prefer-online` setting. It tells browser exactly to do this: check whether master resources are reachable online, otherwise serve them from cache. Another problem occurs as the application has to wait for connection timeout (several seconds) before displaying the master entries decreasing the user experience while browsing web applications offline [23].

Another solution is to “fallback” the dynamic content web page to another one which loads data from local storage. This can lead to a robust solution but there is still one problem – at least one page (preferably the home page) must include a link to the manifest file which will be cached. If it includes dynamic content the original issue sustains. The `prefer-online` setting helps to solve it.

The real problem is that the application cache technology is most suitable either for basic static pages or for dynamic pages of next generation. Web applications used to be designed that server generates a web page which mixes the semantics (HTML), logic (JavaScript) and content (text nodes in HTML elements). Modern web applications tend to be designed like that: Server generates an empty template (HTML and CSS) with logic (JavaScript) and uses other technique (like AJAX or WebSockets) to fetch data from web server and client side MVC frameworks (in JavaScript Ember, Knockout and many others²) are responsible to generate appropriate content in HTML [23].

When building web applications in the second style (thick client), they can be very easily stored offline using local storage, indexed database and application cache and they can be more responsive as they can provide basic functions without even accessing web servers and uses the connection just to transmit data, not user’s actions or web pages themselves.

2. <http://www.infoq.com/research/top-javascript-mvc-frameworks>

It is also more portable as the server has to contain an API (or even better RESTful interface) which native applications for mobile phones can use. The disadvantage is that they are JavaScript dependent.

5.6 Control of process

There is no API which allows developers or users of the application to control caching of data. The process is automatically executed when user visits the page with `manifest` attribute set. Nobody can control the flow of downloading. Developers can manually restart the process of downloading or abort it. Users cannot decide whether they want this application accessible offline and web pages can take up space on the hard drive of user's computer even when he will not access the page again. The "Save offline" button cannot be created with present JavaScript API and must be created with hacks.

The API to add, edit or remove items of application cache is not available. In case of need to cache resources dynamically, Web Storage technology has to be used or the manifest files would need to be generated dynamically.

Chapter 6

Proposed solution

This chapter proposes a new, modified language of application cache manifests Less-Appcache, which helps to overcome some of the problems mentioned in the previous chapter, and describes framework processing the extended language allowing users to add their own processing modules.

6.1 Modifications of language

To enable more effective building of manifest files, there is a need to create a language and processing tool which supports automatization of routine tasks. The language should also support easy extensibility as new requirements can come with utilization of offline web technologies.

The approach to a solution was heavily influenced by LESS CSS framework¹ which extends CSS with dynamic capabilities developers can take advantage of. LESS CSS solves a very similar problem as this thesis does. It takes an existing almost native technology for browsers and adds new features on top of it to make most common tasks more accessible. LESS CSS defines its own language based on the original one and provide parsers to generate CSS files from LESS files which browsers can interpret.

The Less-Appcache language provides very similar feature: It allows users to create more sophisticated cache manifest documents which are then parsed by the processing tool and a valid manifest file is generated.

There were several key points discovered while analyzing the introduced problem. Following next sections present crucial requirements laid on the language.

6.1.1 Connection between languages

New language should fully support the application manifest language described by the specification so users do not have to change their habits as they are most probably used to work with the original language. So every valid manifest file written in plain application cache manifest language, which the browser can process, can be also considered as written in the Less-Appcache language. The parser must work exactly as the browser's one and

1. <<http://lesscss.org/>>

follow the algorithm defined in Parsing cache manifests in the specification².

This new language should use new extension `.lesscache` to distinguish the application cache manifest and extended type of language. The `.lesscache` extension should never be served to the client as the clients are not capable of working with this type of files. The processing tool should generate `.appcache` files from `.lesscache` every time they are requested or in production environment `.appcache` files should be pre-generated by the tool if they are not needed to be generated dynamically.

6.1.2 Import support

The language should support imports of other manifest files, no matter whether they are written in the plain application cache manifest language or in Less-Appcache language. This feature should support recursive importing so the imported documents can state another manifest files to be imported.

Definition. The import clause can be described by following regular expression:

```
^\s*@import\s+('(.*)'|"(.*)"|[\^'" ](.*)[\^'" ])\s$
```

Line can start with any number of U+0020 SPACE and U+0009 CHARACTER TABULATION (tab) characters followed by U+0040 COMMERCIAL AT (@) character and case sensitive word *import*. Then any number of U+0020 SPACE and U+0009 CHARACTER TABULATION characters can come and then a single URL path to the file that should be imported which can be wrapped in U+0027 APOSTROPHE or U+0022 QUOTATION MARK characters.

```
@import "path/to/file"  
@import path/to/file  
@import 'path/to/file'
```

Example 6.1.1: Example of valid import clauses

6.1.3 Parsing filters

The language should support definition of filters enabling to dynamically generate resources added in the application cache manifest. Filters are basically references to a certain program code which takes various number of parameters and returns lines of strings that are added to the manifest.

Import is a specific type of filter which takes one parameter (a path to manifest file) and returns lines which are built from the contents of the file.

Definition of the general filter is very similar to the definition of import clause, however various filters can accept various number of parameters.

2. <http://www.whatwg.org/specs/web-apps/current-work/multipage/offline.html>
(work in progress)

```
^\s*@filter-name(\s+('(.*)'|"(.*)"|['"](.*)[\'"]))) *$
```

6.1.4 Language extensibility

Language should be extensible so developers can easily add the functionality they need. Developers can create their own filters which extends facilities of the application. Each filter has to start with the @ sign and can accept any number of arguments. The processing tool is then responsible for generating appropriate content of the file (based on definition of the filter) which matches the application cache manifest syntax.

Currently, the language is defined by syntax of the application cache manifests and this syntax is extended with filters defined above, but the language is also prepared to be extended beyond possibilities of the filters. Filters are limited – there can be just one on each line, the line must start with the declaration of filter and they can generate zero or more lines added to the output when processed. If needed, developers can introduce completely new syntax by adding a module to the processing tool. The language itself is not limited at all. New capabilities can be added to the language by defining new special symbols that can be parsed. The only limitation is that new syntax should not collide with the manifest file syntax or filter syntax. Proper special symbols are those that should not be present in the URL so the parser do not process valid URLs as a kind of language extension.

For example when developer wants to add support for variables to the language, he should define a correct syntax of setting and getting values using formal specification or regular expressions and also define behavior of this new construct (relation to other modules and language concepts and proper use as in following paragraph) and then create a module for the processing tool which ensures the declared behavior.

Introducing a variable named “varname” and/or setting value “val” to it (regex):

```
^\{\$varname=val\}$
```

Getting a value of variable “varname” (regex):

```
\{\$varname\}
```

Setting a value of variable must be on separate line, getting value can be on any line anywhere. All values are treated as strings. When the value is got, the parser should insert the value in the place where the value was retrieved and application should continue its processing. The processing of variables should take place first, before filters or normal syntax is processed. If the value is required and the variable does not exist, file processing should end with an error stating the line where the problem occurred and information that variable was not introduced.

6.2 Existing filters

There are currently four filters present in the processing tool and defined by the language, namely `glob`, `r-glob`, `regex` and `r-regex`. These filters are very similar – they all take

```
CACHE MANIFEST
{$version=1.0}
CACHE:
@import "manifest-{$version}.appcache"
css/styles-{$version}.css
```

Example 6.1.2: Usage of variables:

two parameters, the first one is a path to a directory, the second is a pattern. These filters walk through the given directory and return all resources which *file name* matches the pattern.

The glob filter uses glob syntax to specify pattern-matching behavior and regex filter uses the regular expressions syntax. The rules of supported glob and regular expressions syntax are defined by Java ³. Glob syntax is much easier and faster but can be insufficient, that is why the regular expression filter is also present.

These two filters just walk the given directory and ignore any subdirectories. If developer also wants to access subdirectories, recursive filters r-glob and r-regex are available.

This is very useful as developers do not have to state each resource manually but this feature should not be abused. Developers should never use filters to import complete static content of their application as that could very easily mean that they include content which is not really needed and just takes space on user's hard disk.

```
@glob ../images *.png
@r-glob apps/icons *.png
```

The first line adds to the manifest file every file in the ../images directory with file extension png and ignores subdirectories. The second line adds to the generated manifest file every resource with png extension in the apps/icons directory and its subdirectories.

Example 6.2.1: Filters

6.3 Processing framework

The framework is responsible for generating a valid plain application cache manifest from Less-Appcache files. It ensures easy extensibility and modularity of the language and produces more effective and pleasant way to create manifest files.

The processing tool should be a helper in debugging and warn the developer about anything which might not work correctly (as mistyped characters in the manifest file) and about statements that do nothing (like not supported setting). Developer should be informed that it is not considered as an error but the manifest may behave otherwise than expected.

3. <<http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html#glob>>
<<http://docs.oracle.com/javase/tutorial/essential/regex/>>

The processing tool automatically detects already imported documents and prevents infinite loops. It also ensures that the listed resources are imported with correct context as relative paths would otherwise not work in the generated file when the main manifest and imported manifests are in distinct directories.

The framework automatically checks whether the resources listed in the explicit and fallback sections do exist and reports any resources that were not reachable. This behavior should not be enforced as some resources do not have to be physical files but can be dynamically generated by the web server and framework can falsely detect them as missing.

The framework detects multiply defined fallback namespaces and reports them as an error as the specification allows only one rule per namespace. It also checks the resource and ensures that each resource is not listed more than once.

The tool also adds an automatically generated comment in the application cache manifest that changes whenever any resource listed in the manifest has changed. This ensures that the browser's application cache is updated every time the offlined resources change on the server.

The processing framework tool can be easily extended by user's plugins by including their own classes on the classpath.

If the file tries to use a filter that the processing framework does not know, the parsing should end with an error identifying line of file which triggered the error and name of filter which could not be found. If any other fatal error happens the processing is stopped and user is informed about the problem. If the error is not fatal, the parser generates output put includes a log that informs the developer about possible problems.

6.4 Solved problems

Proposed language and framework helps to overcome several pitfalls of developing an offline web application. Solved problems are: change of resources by checking modification of resources listed in the manifest, modularity by introducing @import clause, error prone lists by checking existence of listed resources. It also enables to list resources dynamically and to create manifest files much easier.

Chapter 7

Processing tool

The architecture and implementation of the tool processing the language is introduced in this chapter. The processing tool has one main task – to generate correct cache manifest file from lesscache file. Besides, it checks whether the included resources exist, report any possible problems and generate version comment based on last modified file to keep cache fresh even when no new resources are listed. This tool is basically a parser of text files in certain format.

7.1 Used technologies

The processing tool was developed using Java Platform, Standard Edition 7 and Maven. Following subsections of this chapter introduces technologies used for developing the project.

7.1.1 Language enhancements in JDK 7

Some new features, introduced in JDK 7 to simplify development, were used in the project. As these features do not have to be well known by the readers a brief summary of the used ones is given.

The *diamond operator* (<>) is introduced to remove the need for explicit type arguments in constructor calls to generic classes. The compiler infers the type [16].

```
// explicit types
List<Map<String, Set<String>>> v =
    new ArrayList<Map<String, Set<String>>>();
// diamond operator
List<Map<String, Set<String>>> v2 = new ArrayList<>();
```

Example 7.1.1: Diamond operator

Switch statements were able to work with primitive types or enumerations. JDK in version 7 now supports the `String` type. The objects are compared using the `equals()` method.

External resources such as connections or streams had to be closed manually no matter the Java program ended correctly or unexpectedly. The *try-catch-finally* blocks were used for handling these resources but the code in the finally section is usually the same. In Java SE 7 the *try-with-resources* statement can be used for automatical care of these resources, no

matter whether the code throws exception. When the program exits the try clause, resources are closed. Objects that can be used in this statement must implement `AutoClosable` interface [17].

```
try (BufferedReader br =  
    new BufferedReader(fr) {  
        return br.readLine();  
    }  
)
```

Example 7.1.2: Try-with-resources

New version of Java SE also supports *multicatch* – one catch block can deal with exceptions of various types at once by separating their type in catch statement by “|” character [25].

7.1.2 New File System API

Work with file system was partially problematic prior to the Java SE 7. Some methods behaved unexpectedly in various file systems and across operation systems. The NIO 2.0 introduced new API for accessing files and dealing with I/O operations.

The API revolves around `java.nio.Path` objects represent path on the file system and contain methods for inspection, comparing and converting the paths. The Path object does not have to correspond with a real file, directory or link on the file system, the class does not detect existence of the path automatically. Developers can manipulate with the Path object arbitrarily and without any consequences to any files. Path is just a virtual object managing operations over paths in the system [27].

Path class provides methods to access parts of the path like subpaths, parent, root, file names and more. Method `normalize` removes any redundancies in the path such as `./` or `dir/./`. The path can also be converted to File or URI object, absolute path and can deal with symbolic links [28].

Real operations on files like checking existence, accessibility, reading attributes, reading and writing of files deleting, copying, moving and others are accessed through static methods on `Files` class, which works with Path instances [26].

As the processing tool is basically a parser dealing with files and resources on the file system of server, features of new API were vastly employed.

7.1.3 Maven

Maven is a framework for complex project management. It consists of set of standards, formats and programs for managing and describing projects. It defines a standard life cycle for building, testing and deploying projects [19].

The project uses Maven to declare dependencies and makes use of assembly plugin to create jar executable archive.

7.1.4 Apache Commons Lang

`StringUtils` and `WordUtils` libraries from Apache Commons are also used in the project as the tool needs to parse lines of text files, but most of the time basic methods of `String` class are sufficient.

7.1.5 Apache log4j

The project uses `log4j` library to log errors and warnings of the application. There are also info logs available in the debugging mode but in production only warnings and errors are displayed to the user.

7.1.6 JUnit

JUnit testing framework is used for unit testing of the application.

7.2 Architecture

Architecture of the processing tool was built on premise that extensibility should be a key feature. As each developer may need other set of functions, there should be no problem to create own plugins and extensions and add them to the tool.

7.2.1 Packages

The library consists of five basic packages and their relations can be observed on [Figure 7.1](#).

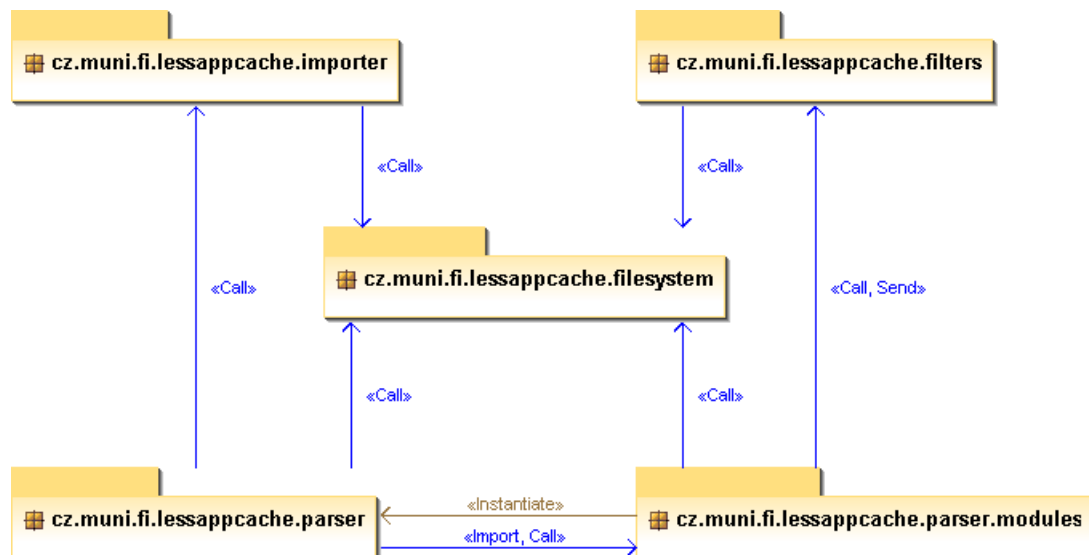


Figure 7.1: Package diagram

- Package parser contains the classes responsible for reading the file line by line and generating output.
- Importer package deals with importing files in right context and prevents circular references.
- Package filesystem comprises of helper classes responsible for loading and writing files and dealing with various paths on the file system.
- Modules package contains classes that are capable of parsing each line and deliver processed lines and resources to the parser.
- Filter package is a set of filters described in [Section 6.1.3](#) and also filter factory and class loader are present to provide loading of assigned filters by reflection.

7.2.2 Class structure

ManifestParser is a class responsible for loading Lesscache manifest files in current file system context. It delegates loading the files to Importer class and FileUtils class. Then, it takes each line of the file and sends it to modules to generate content of the standard manifest file. Modules are loaded via ModuleLoader and are executed in given order. Each module takes care of line in specified format and produces output in form of ModuleOutput which contains generated lines as well as information about state of the parser and instructions about further processing of current line.

[Figure 7.2](#) shows class diagram of the core of the processing tool and [Figure 7.3](#) represents the basic sequence of file processing by the tool.

Manifest parser needs to keep state of currently processed file. It must remember the mode in which the manifest, on current line of execution, is – whether it is an explicit, network or other section. It can also store an absolute path describing the root of server where manifest file will run.

ManifestParser also takes care of redundancies in resources. As other manifest files can be imported in Lesscache manifests, it ensures that each resource stated in the manifest file in explicit or fallback sections is no more than once in the outputted file. It checks whether the resource really exists and if it does not it logs a warning that file can contain a typo.

The last function the ManifestParser class procures, is generating the version comment. It examines each loaded resource across all imported files and finds the last modified one. When all files are parsed and application cache manifest is generated, it adds one line containing information about the file which was last modified. This function ensures that manifest file is always fresh despite the fact that the contents of it did not change, but also ensures that in case no file has changed since last time the manifest was generated, the browser will not have to redownload resources.

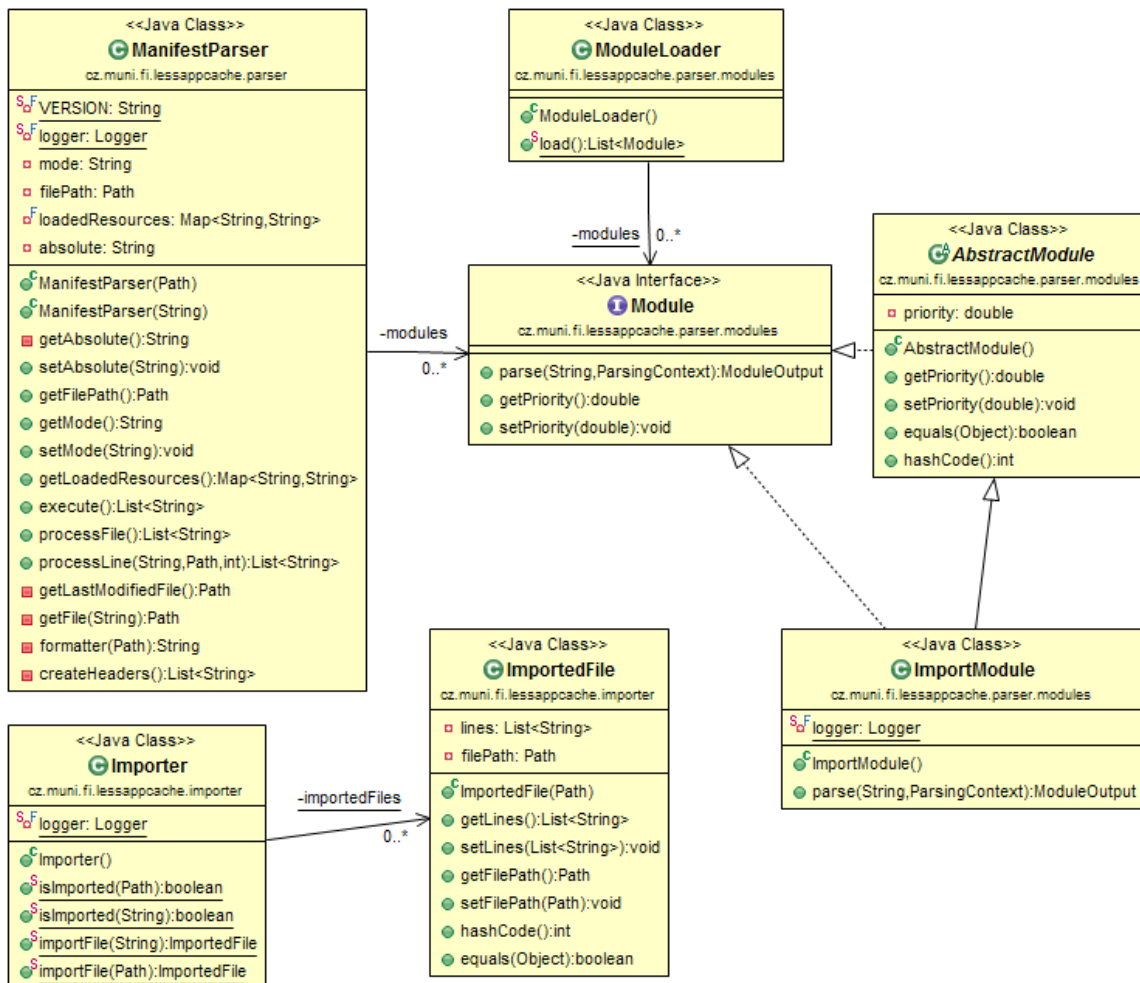


Figure 7.2: Class diagram of core of the application

7.3 Modules

Modules, as stated before, process only the lines that meet the format defined by the module. Current application is composed of eight modules providing the main tasks of the processing tools.

- **HeaderModule** processes the section header lines. The application adds the section header to the output when the section header is different than the previous one (for example if FALLBACK: header was declared twice in a row it ignores the second one) and switches the ManifestParser mode to the stated one by the line as the processing tool must work differently if it is in the settings or explicit part of the manifest file.

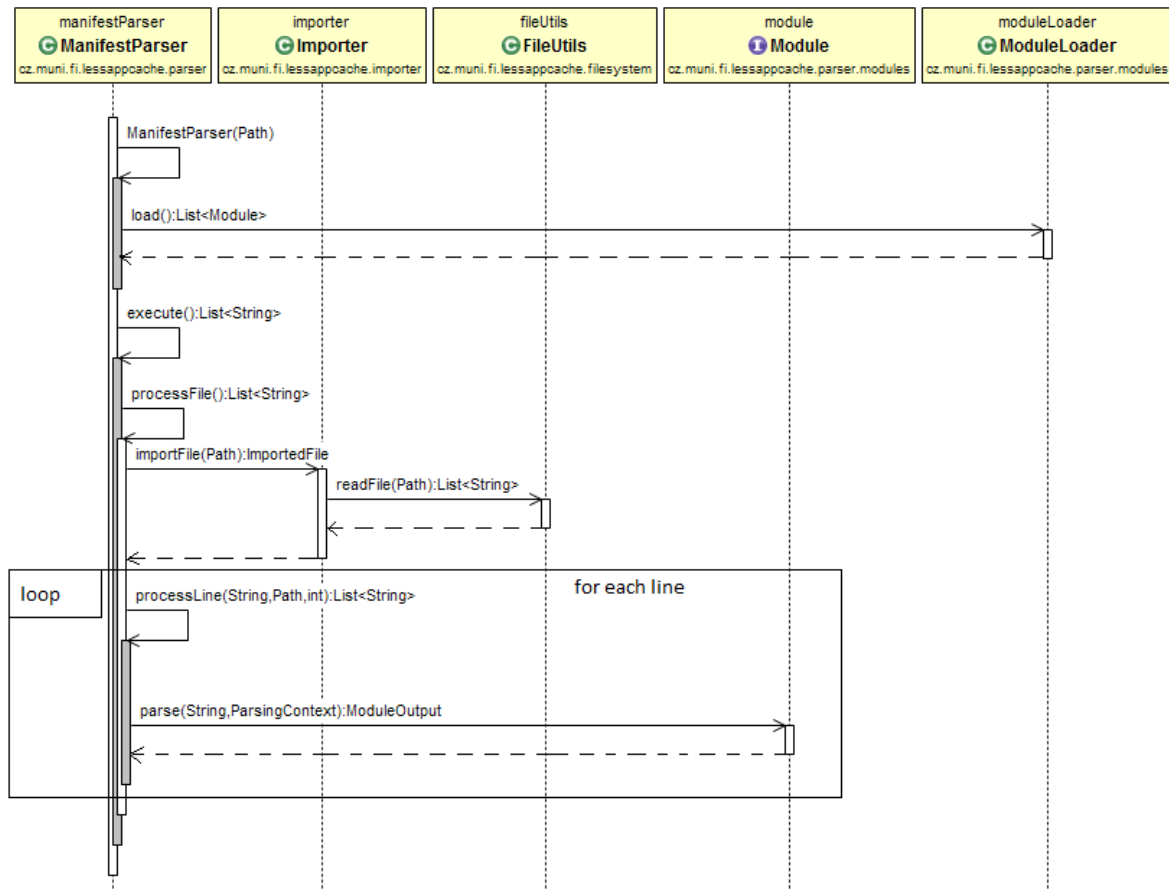


Figure 7.3: Sequence diagram of parsing a manifest file

- **CommentModule** parses lines starting with the # sign and empty lines. Currently all comments are omitted as they are not needed in the generated file.
- **ImportModule** treats lines declaring that contents of other manifest file should be loaded in the current manifest by the @import statement. It creates a new instance of ManifestParser and loads the appropriate file in case the file has not been imported yet.
- **FilterModule** is parsing those lines beginning on @ character. Filters can accept various number of arguments and module ensures that correct filter is called and arguments are passed to it. If the filter specified in the manifest file does not exist or invalid arguments are passed, the line is skipped and error is logged pointing out a correct use of the filter.

- **ExplicitModule** is responsible for processing the resources in explicit section. It must control that the paths of resources are added in correct context of the main file.
- **FallbackModule** covers the fallback section. It has same responsibilities as ExplicitModule and in addition it checks whether the line consists of two resources and that fallback namespaces are defined only once in whole manifest.
- **NetworkModule** parses lines in the network section.
- **SettingsModule** contains a list of supported settings available in the settings section. It ignores any settings not defined by the standard. In future, when more settings are available, this module will be in charge of keeping consistency of settings. When some settings are contradictory, the SettingsModule must warn developer that the manifest file contains errors.

ModuleLoader which ensures loading of modules uses Java's ServiceLoader to provide easy extensibility of the tool. Service in this case is represented by Module interface. Current services implementing Module interface are stored in META-INF.services resource package. When developers want to add their own modules they need to add their own jar package on the classpath when executing the application containing the file describing new modules. ServiceLoader automatically loads these modules in the application and parser executes them in given order [15].

```
public static List<Module> load() {
    if (modules.isEmpty()) {
        ServiceLoader<Module> loadedModules =
            ServiceLoader.load(Module.class);
        for (Module module : loadedModules) {
            modules.add(module);
        }

        Collections.sort(modules, new Comparator<Module>() {
            @Override
            public int compare(Module a, Module b) {
                return (a.getPriority() > b.getPriority()
                    ? -1 : (a.getPriority() == b.getPriority() ? 0 : 1));
            }
        });
    }
    return modules;
}
```

Order of modules is described by module priority. Implicit module order is:

1. HeaderModule
2. CommentModule

3. ImportModule
4. FilterModule
5. Explicit, Fallback, Network, Settings modules according to the mode which application manifest is in the moment of execution

When developers want to integrate their own module, they need to specify when their module should be executed as one line can be parsed by more than one module. For this purpose ModulePhases class defines own constants to define order. These phases are:

1. **START** says that module must be processed before any other.
2. **PRE_COMMENT** defines that module must be called after HeaderModule but before CommentModule.
3. **PRE_IMPORT** constant ensures that module will be executed after CommentModule but before ImportModule.
4. **PRE_FILTER** controls modules after ImportModule but before FilterModule.
5. **PRE_RESOURCE** determines modules that will be used for parsing after FilterModule but before resource modules are executed.
6. **END** defines that module should be executed when resources are processed.

For example when developer creates the module processing variables mentioned before, appropriate phase would be PRE_IMPORT as comments can contain information about variables and should not be parsed but import clauses can contain variables.

In case there are more modules with same priority, the order in which they were added on the classpath and order of files in the service file is considered.

Modules take two arguments. The first is the line of the manifest file which can be parsed if it meets the requirements of the module. The second one is current parsing context of the parser. The context determines in which mode the parser already is (modules need to know whether the manifest file is currently in explicit or fallback section and take actions according to current mode).

Modules return an instance of ModuleOutput. This object contains data about actions the module took while it was parsing the line and includes controlling structures so the manifest parser knows how to work with it. Usually, the most essential attribute of ModuleOutput is output. It is a list of lines which should be added in the result of the parser. ModuleOutput also comprises of new mode (fallback, network etc.), the parser should switch to and ModuleControl enum.

ModuleControl tells parser what to do with produced output. Possible values are:

- **CONTINUE** – Module did not parse the line or the output should be ignored and next module should process the line.

- **REPARSE** – Module parsed the line and generated an output line or lines. Manifest-Parser should take these lines and execute modules on each line.
- **STOP** – Module parsed the line and consider processing of the line as terminated. The parser must not execute any other module on the input line or produced line(s).

For the example of module processing variables, the STOP value for setters and REPARSE value for getters would be suitable. The line containing setter of variable should not be processed further as the module already set its own state according to the value of the variable and the line setting variable should not contain anything else. In case the line contains getter of a variable it should be reparsed as the module generated a line where the variable name was replaced by value of the variable and other modules can work with this line (for example import or resource modules).

In case the module encountered any unrecoverable error, `ModuleException` should be thrown.

7.4 Importing

One of the key features of the extended language is importing other files in the scope of the current file. This asset is quite problematic as each imported file has to be parsed in context of the current directory. [Example 7.4.1](#) demonstrates how the context has to be processed.

Manifest importing file:

```
@import path/file.lesscache
```

Imported file:

```
subpath/resource.ext
```

Result:

```
path/subpath/resource.ext
```

Example 7.4.1: Context of import

Imports can be even recursive – imported files can make another imports and so on.

The import feature is delivered by `ImportModule`. This module creates a new instance of `ManifestParser` with the file name to be imported. The instance of `ManifestParser` holds current context which determines relative path between directory of the processed file and directory of file to be imported. Processed resources of the file are returned to the `ImportModule` and it produces the output in correct context.

```
public static String processResource(String resource, Path context) {
    if (isAbsoluteOrRemote(resource)) {
        return resource;
    }
}
```



```

    }

    Path path = Paths.get(resource);
    return context.resolve(path).normalize().toString();
}

```

Invariant of the nested importing is: outputted resources are relative against the file which imported them. This context is transmitted between parser and modules by instance of `ParsingContext`. Filters also need the context to work with correct directories and files. This way all lines representing resources are generated correctly.

Figure 7.4 shows the sequence diagram of processing line with import statement.

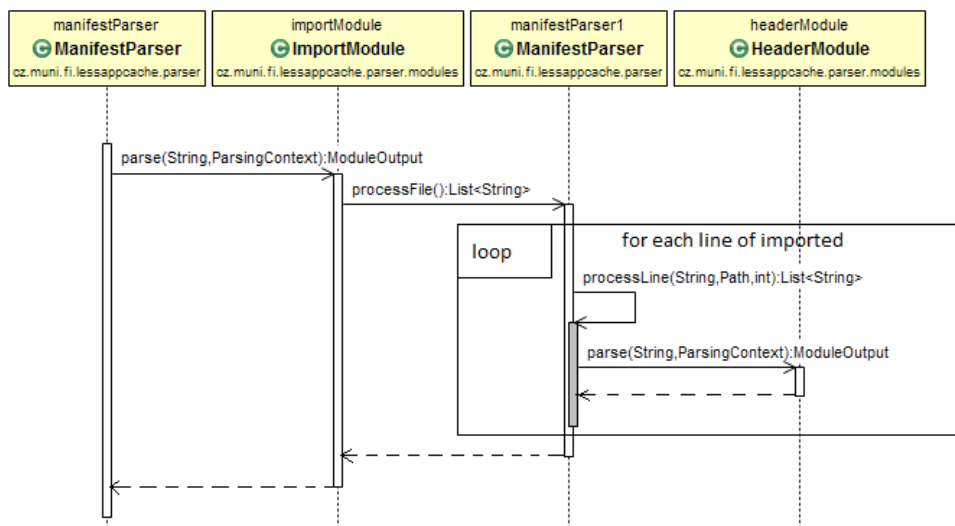


Figure 7.4: Simplified example of imported file stack

7.5 Filters

Functions of filter and available filters were briefly explained in [Section 6.1.3](#). This subchapter shows how the filters are implemented and defines options how to write and import own filters.

Filters are processed by `FilterModule`. This module processes every line that starts with `@` character. It splits the line by whitespaces to obtain filter name and arguments passed to the filter. Then it calls `FilterFactory` to obtain instance of requested filter and executes it with given arguments and context of the parser. `FilterFactory` ensures that only one instance of each filter exists to maintain state of it, in case the implementation of filter needs to preserve some behavior states, and to optimize performance.

`FilterClassLoader` class is used for real loading of the filters. It takes the declared name of `@any-filter-name` and translates it to the `AnyFilterName` and looks for this class in `cz.muni.fi`.

lesscache.filters package using reflection. This way filters of third parties can be easily added to the application by including their jar file with filters in correct package. If given filter could not be found, instantiated or accessed, application throws `FilterException`.

Important part of `FilterFactory`:

```
public static Filter getFilterInstance(String filterName)
    throws FilterException {
    if (!filters.containsKey(filterName)) {
        try {
            Class filter = loader.loadClass(filterName);
            filters.put(filterName, (Filter) filter.newInstance());
        } catch (ClassNotFoundException |
            InstantiationException |
            IllegalAccessException ex) {
            throw new FilterException("There was an error when " +
                "calling filter " + filterName, ex);
        }
    }
    return filters.get(filterName);
}
```

and `FilterClassLoader`:

```
@Override
public Class<?> loadClass(String name) throws ClassNotFoundException {
    return super.loadClass(resolveName(name));
}

private String resolveName(String name) {
    char[] delimiter = {'-'};
    return "cz.muni.fi.lessappcache.filters."+
        StringUtils.remove(WordUtils.capitalizeFully(name.substring(1)
            , delimiter ), "-")+ "Filter";
}
```

Figure 7.5 shows simplified class diagram of filters.

7.5.1 GlobFilter and RegexFilter implementation

Both `GlobFilter` and `RegexFilter` are used for listing files in given directory which matches a given pattern. `GlobFilter` uses glob syntax and `RegexFilter` uses regular expressions syntax for file matches.

They both extend `AbstractWalkDir` class providing the basic functionality. This abstract class also implements `Filter` interface. The `AbstractWalkDir` uses new API of `nio2` to list directory contents. New API defines an interface – `DirectoryStream`¹ which extends `Iterable`

1. <http://openjdk.java.net/projects/nio/javadoc/java/nio/file/DirectoryStream.html>

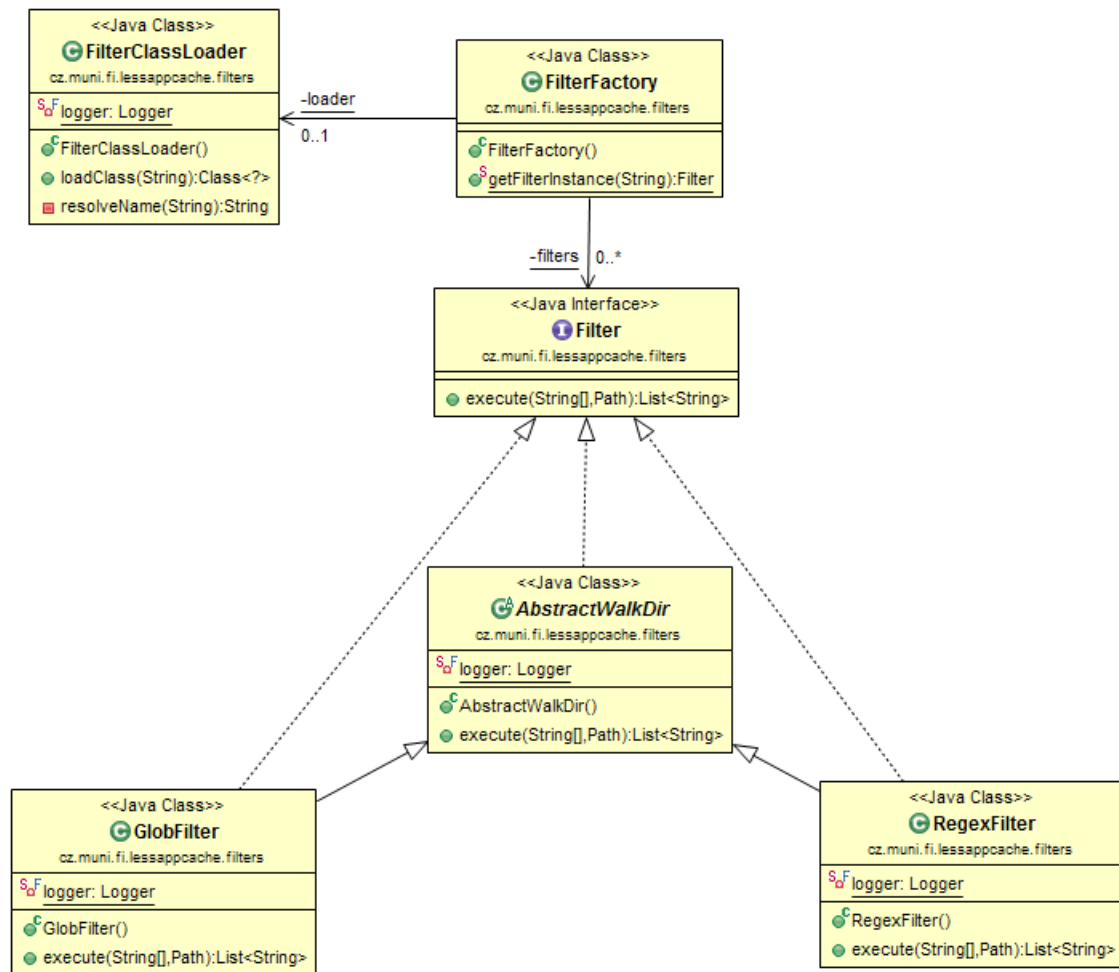


Figure 7.5: Class diagram of sample filter loading

interface, therefore items of directory can be walked in a for each loop. The `AbstractWalkDir` class returns every file in given directory that matches the pattern (regex or glob) that is not a directory. This way files of single directory can be obtained. Try-with-resource statement is used to safely close the accessed directory and `FilterExecutionException` is thrown when an I/O problem occurs. Next block of code demonstrates obtaining contents by New I/O API.

```
try (DirectoryStream<Path> stream = Files.newDirectoryStream(path,
    new DirectoryStream.Filter<Path>() {
        @Override
        public boolean accept(Path entry) {
            FileSystem fs = path.getFileSystem();
            final PathMatcher matcher = fs.getPathMatcher(args[2]);
            return matcher.matches(entry.getFileName());
        }
    }) {
```

```
))) {  
    for (Path entry : stream) {  
        if (!Files.isDirectory(entry, LinkOption.NOFOLLOW_LINKS)) {  
            result.add(pathRelative.resolve(entry.getFileName()).toString());  
        }  
    }  
}
```

7.5.2 RGlobFilter and RRegexFilter implementation

RGlobFilter and RRegexFilter are very similar to the previously mentioned filters but walks the directories recursively.

Also, they extend an abstract class that provides the functionality of walking the directory tree – AbstractWalkTree. New I/O API is as well used in this case. The API defines walkFileTree method on Files class. This method walks the tree of directories starting with given directory and invokes FileVisitor class on each file. The AbstractWalkTree class contains an inner Finder class extending a SimpleVisitor. The Finder class is comprised of methods defining behavior when the class visits a file or a directory. It applies a pattern matcher on every found file and ignores directory names. It visits every subdirectory ignoring symbolic links to prevent loops. Results are then relativized against the given directory and returned.

7.6 Execution

Current application includes class Main which is declared in maven assembly plugin. This way deployed jar can be executed. The main method must be launched with at least one parameter and at most two. The mandatory argument is a path to file to be processed. Files are always processed against actual working directory. The second parameter can be a path to the root of the web server. The processing tool then checks the resources defined with absolute path against this root path. This argument is optional. In case the argument is not present, absolute paths are not checked.

```
java -jar lesscache.jar file.lesscache /path/to/server/root > file.appcache
```

The tool can be also used as a library for other front-ends. It could be a part of GUI editor for manifest file which automatically processes resources and checks whether they exist. When more arguments and flags are needed, Apache Commons CLI or similar can be used to be more specific. Next versions of the tool can provide flags to set how the application should log errors or print output files, or to specify checking of existence of files, possibly format of version comment and more.

Chapter 8

Demonstration

Use of language and framework will be demonstrated on a real life web application Celebrio which is suitable to contain offline mode. In this chapter, Celebrio application will be introduced and transformation process of two modules to ensure capability of running offline will be described. The end of this chapter sums up benefits and results of use of the extended language and processing tool.

8.1 Celebrio

Celebrio is a web application developed by Celebrio Software, s.r.o.¹, simulating interface of an operating system. The main target is to provide a system with simple interface and without need to force users to make decisions on tasks they know nothing about. The system is designed for elderly users, users without any computer knowledge and for users with various handicaps. It is suitable to be used on computers with touch displays and also on classic computers and laptops with mouse.

The web system provides several applications to facilitate basic computer tasks like communication, obtaining information and entertainment. These applications are: News, Mail, Talker, People, Gallery Weather, Books, Games and Internet. Each application focuses on one task allowing users use their computer as simple as possible. The applications provide unified user interface to deliver complex experience.

Celebrio is accessible right from the browser (currently supported are Internet Explorer 9+, Mozilla Firefox, Google Chrome and Safari). This way users can reach their data on any device with internet connection without need to install anything.

Dedicated applications are available for Microsoft Windows and Android. Installed application is able to add closer integration with the underlying operating system and add features that are not possible in a common web browser. Windows application is available for Windows XP, Windows Vista, Windows 7 and Windows 8 (x86).

Project is completely based on web technologies like HTML5, JavaScript or CSS3 and native applications use web browser rendering cores to display the contents. Server side of the project is developed in PHP 5.3 and runs in cloud on Microsoft Azure platform.

Celebrio currently needs persistent connection in order to run. Most applications are connection dependent like weather or news. Problem is that the application does not work

1. Reader can try the system for free on <http://www.celebriosoftware.com>

at all when the internet is not reachable. Users could read news saved while browsing, look at older photos, read books or play games while offline. This is why the described language and framework was integrated in the application.

8.2 Demonstration application

Subset of Celebrio application was created to demonstrate capabilities of framework. This demonstration application is available on an attached CD and is also running on <http://demo-lesscache.rhcloud.com> so readers can easily test the capabilities. The application is in czech language as the company now offers content mainly from czech providers (news).

8.2.1 Description of the application

The demonstration application consists of three modules – the main module simulating the desktop of Celebrio application and two application modules simulating modules News and Games of original application.

Main screen is divided into three areas which are consistent through whole application. Header contains title of the screen and may contain a scroller for horizontal layouts when the content is wider than width of the screen. Footer part can contain two buttons – lower left is the back button which navigates browser one level up and lower right button leads right to the main menu so users do not have to click through menus when they want to switch application. The main part between header and footer varies through applications and provides the main functionality. The content is scrollable, most of menus can be dragged horizontally, some content (for example articles in application News) can be dragged vertically.

The main module is the main menu of the application and in demonstration application comprises of two items as the application includes just two modules.

The game module main page includes links to three games: Sudoku, Chess and Solitaire. Games are free flash applications created by third parties. The game is loaded after click on the menu item and help is loaded as well. The game module is available offline after user visits any page of game module and all games are available.

The news module main page menu comprises of categories. Each category contains articles from media portals. The demonstration application includes sources from Czech television² and Idnes³. The original application includes more sources but for demonstration purposes only two were chosen. Each article contains save button which user can use to pick articles he wants to read later even when he does not have internet connection. These articles can be found in category Saved (“Uložené”). Application is capable of running offline once the application is visited. When offline, anytime users tries to reach a web page in news module which is not available, browser automatically redirects to the category Saved and

2. <http://www.ceskatelevize.cz/rss/>

3. <http://rss.idnes.cz>

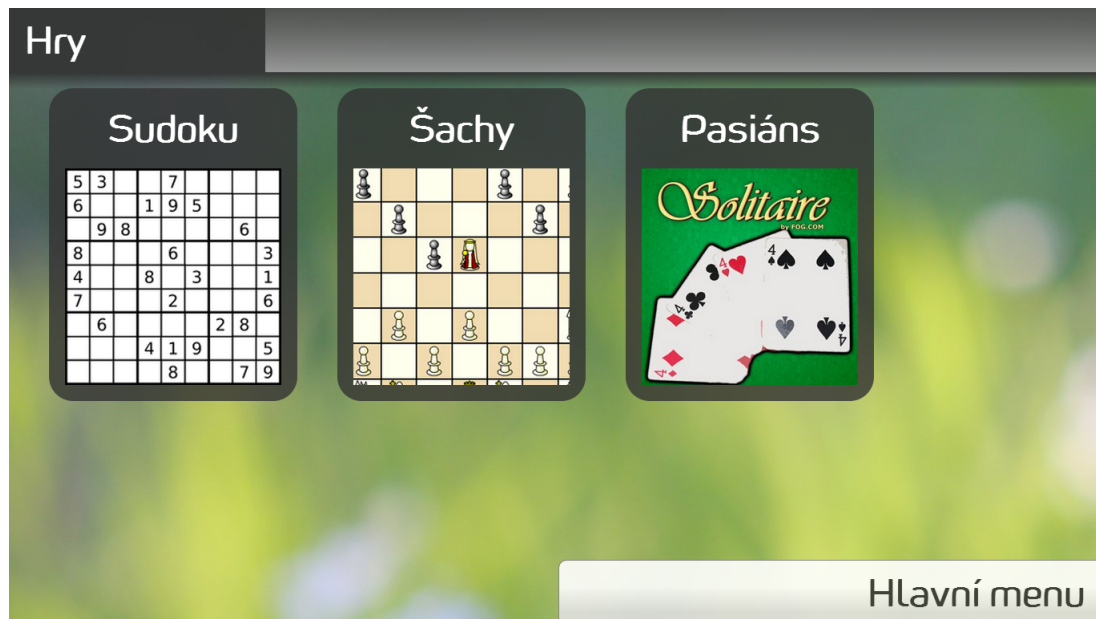


Figure 8.1: Screenshot of news application

displays a message which notifies user that he is offline and only saved articles are available. The pictures are not saved with the article as they would needlessly burden user's storages.

8.2.2 Offline Web Application

There are four Lesscache application manifests which are translated by the framework processing tool to regular application manifest files.

The `system.lesscache` file lists all resources which are needed by whole application like background image, fonts, common styles, common scripts and libraries. Every other lesscache file imports this file so the shared resources are accessible by every module.

The `main.lesscache` is for the main menu. It imports the system application cache manifest and adds icons which are visible in the menu. It also adds the main fallback – any not caught resources by fallbacks in other modules redirects to the main menu. The `prefer-online` setting is also introduced to check whether the master entities are new on the web server.

The `games.lesscache` makes possible that the game module can be accessed offline. It also imports the system manifest to enable basic functions of the Celebrio application and adds application-specific resources – subpages with the games and help, styles, images and flash files. Once visited, whole application is reachable offline.

The `news.lesscache` is responsible for defining resources for the news application. It imports the system manifest as the other modules and adds its own styles and icons. Furthermore, it defines the fallback routing any non-cached resources of the news application

to the web page which displays the saved articles:

```
/news /news/offline
```

As stated before it is not possible to create “offline” buttons to define which resources should be offline by JavaScript API. The offline mechanism used in news module circumvents this disadvantage: The button, which makes the article available offline, simply reloads the web page with article but the server adds the `manifest` attribute in the `html` element. This way the web page with the article is added as master entry and therefore stored in the browser. The Web Storage API is used for remembering which links were generated with the attribute linking the manifest and creates the list of articles in the Saved category.

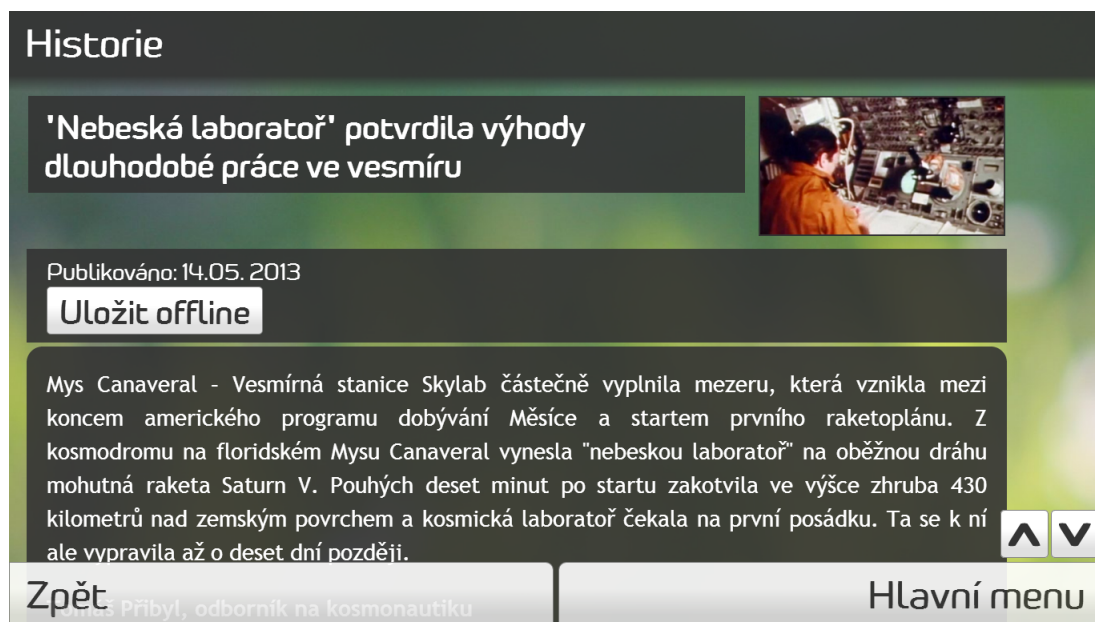


Figure 8.2: Screenshot of news application with Save button

8.3 Benefits of framework

The framework brings several benefits when creating application manifest files. They were described in the [Chapter 6, “Proposed solution”](#). This section summarizes the advantages the language and processing tool delivers.

8.3.1 Application modules

When user visits the main page, only the main page itself is available offline. The standard approach would be to create one application cache manifest for whole application and whole web site would be capable of running offline but in case there were tens of applications available, the web browser would need to make them available offline all at once. This

could take too much time and store too large amount of data so the browsers could refuse the request. Also in case the applications would be created by third parties, it would be impossible to maintain one manifest file for all applications. The applications themselves have to be responsible for the process of making them work offline.

When user visits the news or games application, they begin to be available offline and they do not influence the other one at all. If user does not use games application and never visits it, the application is not available offline and does not store any data on user's computer. By this style of creating manifests, complex and vast applications can be created and only modules used by the particular user are accessible offline.

8.3.2 Reduced number of lines

The filters that facilitate adding resource based on pattern matches reduce number of lines developers need to write to cache all needed resources. It also makes harder creating typing errors as the filters find automatically the matching files and add them to the manifest. Advantage of modules also reduces number of lines as repeating resources need to be stated just once.

Table 8.3.2 shows the difference between number of non-empty lines in the lesscache file which needs to be composed by the developer and generated application cache manifest which represents the file which would be needed to be written without use of the Lesscache framework. Numbers in the table represent count of non-empty lines.

File	Lesscache	Appcache	Reduction
System	10	46	36
Main	7	54	47
Games	12	63	51
News	12	66	54
TOTAL	41	229	188

Table 8.1: Total number of lines in the application manifest files

This table presents, that in this particular application (and style of writing manifest files like modules) only less than 18 % lines had to be written by the developer. The reduction of the code is therefore more than 82 %. However, in case the application cache manifest files would not be developed in the module style and every line would be put in one file, the reduction would not be so significant. Nevertheless, the lesscache manifest files could be even shorter by defining more general patterns in the filters.

Example of differences between lesscache and appcache files is stated in **Appendix B**.

8.3.3 Automatization

The processing tool can check syntax and mistyped characters. If any problem happens inside of the framework (invalid filter name, invalid number of arguments, incorrect argument

and so on), the processing tool logs the error. When developer states a resource which is not found on the file system, the application raises a warning.

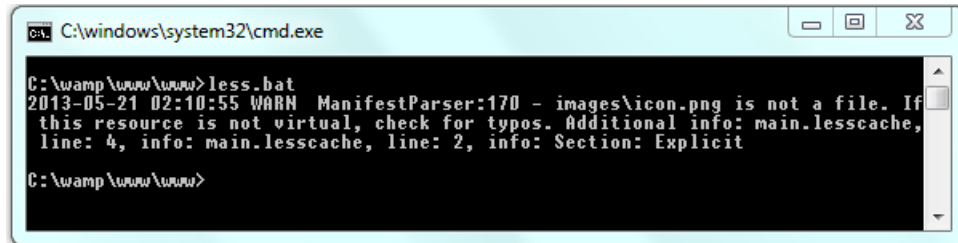


Figure 8.3: Command line warning

The processing tool can be easily launched automatically. Scripts can be run in the deployment process so developers do not have to update manifest files with new resources in the files as the filters automatically recreates manifests.

```
@ECHO OFF
set lesspath=C:\absolute\path\to\lesscache.jar
set original=%CD%
FOR /R . %%G IN (*.lesscache) DO (
    cd %%~dpG
    java -jar "%lesspath%" %%~nxG > %%~nG.appcache
)
cd %original%
```

Example 8.3.1: Windows batch file executing lesscache.jar on each lesscache file

Web browsers parse the manifest file when the contents of it has changed but the change often happens only inside of the resources (for example a few lines were added in the stylesheet or used picture was replaced). The manifest itself has not changed but new resources should be reloaded. This is why the processing tool adds the version comment. The tool checks every resource it processes and in the end it adds a line containing information about file that was last modified. This way the manifest file changes against its predecessor only when the resources have really changed and redownloading is necessary. In case no file has changed since the last deploy, the generated manifest file will be exactly same as last time.

```
# Version: Last modified file
# styles\less\webTop.less, Date: 05/03/2013 12:23:20
```

Example 8.3.2: Version comment

Chapter 9

Conclusion

This thesis embraced the topic of developing web offline applications. The technologies Web Storage, IndexedDB, File API were introduced briefly and Application cache technology was described thoroughly with all its advantages and disadvantages this approach contains.

Framework, containing language extending original application manifest language and processing tool parsing it, was described introducing its advantages. Application cache manifests can now use modules to make offline just parts of the web pages, processing tool automatically checks existence of stated resources to prevent browser failures. Resources can be added to the file by making use of filters which enable listing whole directories at once. This feature lowers risk of making a typo and simplifies work for developers as they do not have to create long lists manually. Framework also automatically creates version comments which can control redownloading resources when needed and only when needed.

Web pages turned from static documents to complex applications providing functions formerly available only in desktop applications. Therefore, creating web applications at least partially available even without internet connection is more important than before.

Demonstration application showed the results of use of processing tool on a real application Celebrio with all benefits it brought. Next step will be bringing these functions to a production version of application and make offline more modules than news and games applications.

The application is open sourced and available under Apache License, Version 2.0 available on <<https://github.com/kuncajs/Less-Appcache>> Further development of the processing tool can be made by anyone and architecture allows easy extensibility of the parser. Unofficial extensions can be added to the tool by including their own jar packages on classpath. First of all, integration tests should be created to verify stability on various platforms as only unit tests are currently implemented. More complicated CLI can be developed to allow developers to define various levels of logging, format of version comments or strictness of resource checker.

Another requirements will be identified when more users will start working with the framework and discover drawbacks or unencountered challenges.

The core goal of the thesis was reached and demonstrated, but more applications using the parsing tool as library can be developed. GUI editor facilitating easy creation of manifest file cooperating with the framework might be useful. Also more complex framework enabling complete automatization of making web applications offline and automatical synchronization would be a good topic for dissertation thesis.

Bibliography

- [1] Bidelman, E.: *A Beginners Guide to Using the Application Cache*, 5/27/2011 [retrieved 2/3/2013], from <<http://www.html5rocks.com/en/tutorials/appcache/beginner/>> . 4.3.2
- [2] Christian, M. and Lubbers, P.: *Appcache Facts [online]*, 11/20/2012 [retrieved 4/23/2013], from <<http://caniuse.com/>> . 4.2.4, 4.3.1
- [3] Berjon, R. and Leithead, T. and Navara, E. and O'Connor, E. and Pfeiffer, S.: *HTML5, W3C Candidate Recommendation* 12/17/2012 [retrieved 3/30/2013], W3C, from <<http://www.w3.org/TR/2012/CR-html5-20121217/>> . 1
- [4] Deveria, A.: *Offline web applications [online]*, 11/12/2012 [retrieved 3/11/2013], from <<http://caniuse.com/>> . 4.1, 9
- [5] Russell, A. and Wilkins, G. and Davis, D. and Nesbitt, M.: *The Bayeux Specification*, 2007 [retrieved 5/1/2013], The Dojo Foundation, from <<http://svn.cometd.org/trunk/bayeux/bayeux.html>> . 2.3
- [6] Bert, B.: *CSS current work*, 5/3/2013 [retrieved 5/9/2013], W3C, from <<http://www.w3.org/Style/CSS/current-work>> . 2.2
- [7] Kinlan, P.: *Working Off the Grid with HTML5 Offline*, 6/1/2011 [retrieved 4/1/2013], HTML5 Rocks, from <<http://www.html5rocks.com/en/mobile/workingoffthegrid/>> . 3.4
- [8] Archibald, J.: *Application Cache is a Douchebag [online]*, 5/8/2012 [retrieved 3/30/2013], from <<http://alistapart.com/article/application-cache-is-a-douchebag>> . 5.1
- [9] Hickson, I.: *HTML 5 Living Standard*, 5/5/2012 [retrieved 5/17/2013], WHATWG, from <<http://www.whatwg.org/specs/web-apps/current-work/multipage/dnd.html>> (Work in Progress) . 2.3
- [10] Casario, M.: *HTML5 solutions: essential techniques for HTML5 developers*, New York, NY: Friends of ED, 2011, ISBN 978-143-0233-879. 2.1
- [11] The Exceptional Performance team: *Best Practices for Speeding Up Your Web Site [online]*, 2012 [retrieved 2/5/2013], from <<http://developer.yahoo.com/performance/rules.html>> . 5.1
- [12] Ranganathan, A. and Sicking, J.: *File API*, W3C Working Draft 10/25/2012 [retrieved 2/2/2013], W3C, from <<http://www.w3.org/TR/FileAPI/>> . 3.3
- [13] Lubbers, P. and Albers, B. and Salim, F.: *HTML5: programujeme moderní webové aplikace*, Brno: Computer Press, 2011, ISBN 978-80-251-3539-6. 1, 4.2.2

-
- [14] Metha, N. and Sicking, J. and Graff, E. and Popescu, A. and Orlow, J. and Bell, J.: *Indexed Database API*, W3C Last Call Working Draft 16/5/2013 [retrieved 5/20/2013], W3C, from <<http://www.w3.org/TR/IndexedDB/>> . 3.2
- [15] Friesen, J. and Husted, T.: *Beginning Java SE 6 platform: from novice to professional*, Berkeley, CA: Apress, 2007, ISBN 978-159-0598-306. 7.3
- [16] Reese, R. and Reese, J.: *Java 7 new features cookbook*, Olton Birmingham: Packt Pub., 2012, ISBN 978-1849685627. 7.1.1
- [17] Gosling, J.: *The Java language specification. Java SE 7 edition*, Sebastopol, CA: O'Reilly Media, 2012, ISBN 01-332-6022-4. 7.1.1
- [18] Flanagan, D.: *JavaScript: the definitive guide. 6th ed*, Sebastopol, CA: O'Reilly, 2011, ISBN 05-968-0552-7. 2.3
- [19] Massol, V. and Husted, T.: *Maven: the definitive guide*, Sebastopol: O'Reilly, 2008, ISBN 978-0-596-51733-5. 7.1.3
- [20] MacDonald, M.: *HTML5: the missing manual*, Sebastopol, CA: O'Reilly Media, 2011, ISBN 978-144-9302-399. 2.1, 4.1, 4.2.1, 4.3.1, 5.4
- [21] Mozilla Developer Network: *IndexedDB*, 2/28/2013 [retrieved 3/11/2013], MDN, from <<https://developer.mozilla.org/en-US/docs/IndexedDB>> . 3.2
- [22] Mozilla Developer Network: *window.navigator.onLine*, 4/28/2013 [retrieved 4/30/2013], MDN, from <<https://developer.mozilla.org/en-US/docs/Web/API/window.navigator.onLine>> . 3.4
- [23] Hickson, I.: *HTML 5 Living Standard*, 5/5/2012 [retrieved 5/17/2013], WHATWG, from <<http://www.whatwg.org/specs/web-apps/current-work/multipage/offline.html>> (Work in Progress) . 1, 4.1, 4.3.1, 4.3.2, 5.5
- [24] Bloom, J.: *Problems with Application Cache [online]*, 6/24/2012 [retrieved 4/2/2013], from <<http://blog.jamesdbloom.com/ProblemsWithApplicationCache.html>> . 5.2, 5.1, 5.5, 9
- [25] Konda, M.: *A look at Java 7's new features [online]*, 9/2/2012 [retrieved 1/12/2013], from <<http://radar.oreilly.com/2011/09/java7-features.html>> . 7.1.1
- [26] The Java Tutorials: *File Operations [online]*, 4/12/2012 [retrieved 5/1/2013], from <<http://docs.oracle.com/javase/tutorial/essential/io/fileOps.html>> . 7.1.2
- [27] The Java Tutorials: *The Path Class [online]*, 4/12/2012 [retrieved 5/1/2013], from <<http://docs.oracle.com/javase/tutorial/essential/io/pathClass.html>> . 7.1.2

-
- [28] The Java Tutorials: *Path Operations [online]*, 4/12/2012 [retrieved 5/2/2013], from <http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html> . 7.1.2
- [29] Pilgrim, M.: *HTML5: up and running*, Sebastopol, CA: O'Reilly Media, 2010, ISBN 978-059-6806-026. 2.1, 2.3, 3.1, 4.2.3
- [30] Berners-Lee, T. and Masinter, L.: *Uniform Resource Locators (URL)*, 12/1994 [retrieved 2/23/2013], from <http://www.ietf.org/rfc/rfc1738.txt> . 2.3
- [31] Hickson, I.: *Web SQL Database*, 11/18/2013 [retrieved 4/25/2013], W3C, from <http://www.w3.org/TR/webdatabase/> . 3.2
- [32] Hickson, I.: *Web Storage*, 4/9/2013 [retrieved 4/22/2013], W3C, from <http://www.w3.org/TR/webstorage/> . 3.1

List of Figures

2	Introduction to HTML5	
2.1	The W3C HTML5 logo	3
4	Application cache	
4.1	UML state diagram of events in appcache	18
5	Drawbacks	
5.1	Diagram of fetching data by application cache [24]	23
5.2	Printed screen of console log in Google Chrome	25
7	Processing tool	
7.1	Package diagram	35
7.2	Class diagram of core of the application	37
7.3	Sequence diagram of parsing a manifest file	38
7.4	Simplified example of imported file stack	42
7.5	Class diagram of sample filter loading	44
8	Demonstration	
8.1	Screenshot of news application	48
8.2	Screenshot of news application with Save button	49
8.3	Command line warning	51

List of Tables

4 Application cache

4.1 Browser support for Application cache technology [4] 13

8 Demonstration

8.1 Total number of lines in the application manifest files 50

Appendix A

Attached CD

Attached CD contains:

- framework source codes,
- framework documentation,
- compiled executable jar package,
- demonstration application source codes,
- this thesis in PDF, XML and TEX format,

Appendix B

Example of lesscache and generated appcache file

system.lesscache:

```
CACHE MANIFEST
images/wall.jpg
images/favicon.ico
images/vertical333333alpha08to00.png
@glob images/arrows *.png
@glob images/scrollbar *.png

@r-glob scripts *.js

styles/fonts.css
@r-glob styles/less *.less
@r-glob styles/webfonts *.*
```

system.appcache:

```
CACHE MANIFEST
# Version: Last modified file
# styles\less\webTop.less, Date: 05/20/2013 22:01:23
# Imported file: system.lesscache
images\wall.jpg
images\favicon.ico
images\vertical333333alpha08to00.png
images\arrows\down.png
images\arrows\left.png
images\arrows\right.png
images\arrows\up.png
images\scrollbar\Scrollbar.png
images\scrollbar\ScrollbarBackground.png
scripts\jquery\jquery-1.9.1.min.js
scripts\jquery\jquery-ui.js
scripts\less.js
scripts\loader.js
scripts\scroll\celebrioIscrollPlus.js
scripts\scroll\iscroll-lite.js
styles\fonts.css
styles\less\app_main.less
styles\less\designPrimitives\content_wrapper.p.less
```

B. EXAMPLE OF LESSCACHE AND GENERATED APPCACHE FILE

```
styles\less\designPrimitives\flash_message.p.less
styles\less\designPrimitives\menu_item.p.less
styles\less\designPrimitives\menu_item_people.p.less
styles\less\designPrimitives\n_line_title.p.less
styles\less\designPrimitives\n_line_title_rounded.p.less
styles\less\designPrimitives\one_line_title.p.less
styles\less\designPrimitives\one_line_title_rounded.p.less
styles\less\designPrimitives\pop_up.p.less
styles\less\designPrimitives\slider.p.less
styles\less\designPrimitives\table_header_title_image.p.less
styles\less\designPrimitives\table_two_column_form.p.less
styles\less\designPrimitives\unauthorized.p.less
styles\less\designPrimitives\unauthorized_notification.p.less
styles\less\system.less
styles\less\webTop.less
styles\webfonts\eot\style_198602.eot
styles\webfonts\eot\style_198603.eot
styles\webfonts\eot\style_198611.eot
styles\webfonts\ttf\style_198602.ttf
styles\webfonts\ttf\style_198603.ttf
styles\webfonts\ttf\style_198611.ttf
styles\webfonts\woff\style_198602.woff
styles\webfonts\woff\style_198603.woff
styles\webfonts\woff\style_198611.woff
# End of imported file: system.lesscache
```

Appendix C

JavaScript APIs Specification

```
interface ApplicationCache : EventTarget {

    // update status
    const unsigned short UNCACHED = 0;
    const unsigned short IDLE = 1;
    const unsigned short CHECKING = 2;
    const unsigned short DOWNLOADING = 3;
    const unsigned short UPDATEREADY = 4;
    const unsigned short OBSOLETE = 5;
    readonly attribute unsigned short status;

    // updates
    void update();
    void abort();
    void swapCache();

    // events
    attribute EventHandler onchecking;
    attribute EventHandler onerror;
    attribute EventHandler onnoupdate;
    attribute EventHandler ondownloading;
    attribute EventHandler onprogress;
    attribute EventHandler onupdateready;
    attribute EventHandler oncached;
    attribute EventHandler onobsolete;
};

interface NavigatorOnLine {
    readonly attribute boolean onLine;
};

interface Storage {
    readonly attribute unsigned long length;
    DOMString? key(unsigned long index);
    getter DOMString getItem(DOMString key);
    setter creator void setItem(DOMString key, DOMString value);
    deleter void removeItem(DOMString key);
    void clear();
};
```

```
interface IDBRequest : EventTarget {
    readonly attribute any result;
    readonly attribute DOMError error;
    readonly attribute (IDBObjectStore|IDBIndex|IDBCursor)? source;
    readonly attribute IDBTransaction transaction;
    readonly attribute IDBRequestReadyState readyState;
    attribute EventHandler onsuccess;
    attribute EventHandler onerror;
};

[Constructor]
interface FileReader: EventTarget {

    // async read methods
    void readAsArrayBuffer(Blob blob);
    void readAsText(Blob blob, optional DOMString encoding);
    void readAsDataURL(Blob blob);

    void abort();

    // states
    const unsigned short EMPTY = 0;
    const unsigned short LOADING = 1;
    const unsigned short DONE = 2;

    readonly attribute unsigned short readyState;

    // File or Blob data
    readonly attribute (DOMString or ArrayBuffer)? result;
    readonly attribute DOMError error;

    // event handler attributes
    [TreatNonCallableAsNull] attribute Function? onloadstart;
    [TreatNonCallableAsNull] attribute Function? onprogress;
    [TreatNonCallableAsNull] attribute Function? onload;
    [TreatNonCallableAsNull] attribute Function? onabort;
    [TreatNonCallableAsNull] attribute Function? onerror;
    [TreatNonCallableAsNull] attribute Function? onloadend;
};
```