BACHELORARBEIT

Titel

Evaluation of Responsive Images Solutions for the Web

Verfasser

Martin Plattner

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, August 2015

Studienkennzahl lt. Studienblatt: 033 526
Studienrichtung lt. Studienblatt: Wirtschaftsinformatik
Betreuer: Univ.-Prof. Mag. Dr. Hans-Georg Fill, Privatdoz.

Abstract

The success of smartphones and tablets has faced web developers with a big challenge. They had to adapt their workflow and create more flexible websites to provide a good user experience regardless of the device. Ethan Marcotte's Responsive Web Design approach solved the issue and fundamentally changed web design. However, Responsive Web Design does not sufficiently address images. The main issue is that the same image files are delivered to all devices. This introduces a big overhead when large, desktop-optimized images are served to small devices like smartphones. The result is an increased page loading time and thus, a poor user experience. Several approaches have been introduced to solve this issue. This work evaluates six responsive images approaches. The aim is to provide a guideline for selecting a responsive images solution. Software architects and developers should be supported in the decision making process when planning and implementing a web project. An evaluation framework was developed to ensure a formal and comprehensible evaluation. The framework consists of seven functional requirements, six non-functional requirements and three system prerequisites. The results clearly show that the native HTML5 responsive images solution outperforms the other solutions. However, the other solutions or a combination might be a sensible option for certain use cases. The results were used to select one solution and use it to create a prototype of a travel website using fullscreen images. This was done to provide a real-world example. The prototype confirmed the results of the evaluation. The native HTML5 responsive images solution was successfully implemented and it was possible to provide fallbacks for older and unsupported browsers.

Acknowledgments

The creation of this thesis took almost a year of work. Many people supported me during this stressful and intense time. I would like to thank the following persons who supported me. Without them, this work would not have been possible.

Huge thanks go to Dominik Bork, the supervisor of this work. Our regular meetings and phone conversations always gave me very valuable input. His expertise on how to write academic works was extremely helpful. Whenever I was stuck he made time and offered good advice which got me back on track. Overall, his motivating and appreciative nature made the collaboration a pleasure.

Big thanks also go to Wolfgang Klas for making time to discuss my thesis proposal and taking care of the formal aspects to make this work applicable for my Media Computer Science program.

Next, I want to thank my friends Lukas Lottersberger, Michael Oppermann, Oliver Spies and Patrick Riley for many helpful discussions. Their web development background and expertise allowed me to get very profound feedback.

I would also like to thank all the people in the community for answering my questions online. It is incredible how many extremely skilled people share their knowledge either by answering questions directly or publishing articles on their blogs.

Great thanks also go to my parents and sisters for their support while going through a hard time themselves. They have emotionally and financially supported me not only during my work on this thesis, but also all-along my studies in Vienna and Stockholm.

Finally, thanks go to my lovely girlfriend Lisa for her incredible support. She always encouraged me and had a lot of patience while I was busy working on this thesis.

Table of Contents

Index of Abbreviations

AJAX	Asynchronous JavaScript and XML
API	Application programming interface
CDN	Content Delivery Network
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DIP	Device Independent Pixel
DOM	Document Object Model
DPI	Dots per Inch
dppx	Device pixel per px *(Note: px refers to CSS pixel)*
DPR	Device Pixel Ratio
FUHD	Full Ultra High Definition
GIF	Graphics Interchange Format
GUI	Graphical User Interface
HiDPI	High Dots per Inch
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JP(E)G	Joint Photographic (Experts) Group
JS	JavaScript
Mbps	Megabit per second
MIME	Multipurpose Internet Mail Extensions
PHP	PHP: Hypertext Preprocessor
PNG	Portable Network Graphics
PoC	Proof of Concept
PPI	Pixels per Inch
RESS	Responsive Design with Server Side Components
RFC	Requests for Comments
RICG	Responsive Images Community Group
RWD	Responsive Web Design
SoC	Separation of Concerns
SVG	Scalable Vector Graphics
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
WURFL	Wireless Universal Resource FiLe
XML	Extensible Markup Language
XSS	Cross-Site-Scripting

Index of Tables

Index of Listings

Index of Figures

1 Introduction

The Web is the world's biggest information network, used by billions of people. For a long time, the environment of a user was pretty predictable and stable. The main device to access the Web was a desktop computer, having a big screen and usually a stable Internet connection. Web designers created websites with this environment in mind. They assumed a certain minimum resolution, eg. 1024×768 pixels, and optimized websites for such screens. This resulted in rather static and inflexible websites – which was sufficient, given that the environment was static as well. However, this has changed. The transition began with the introduction of the first Apple iPhone in 2007. It marked the beginning of the ongoing smartphone boom. New models were released in increasingly smaller intervals. The rise of smartphones was followed by the success of tablets. It started in 2010, again initiated by Apple with the release of the first iPad. The device landscape has been growing ever since. Wearables like watches and even glasses are on the verge of becoming the next hype. The new devices completely changed the user environment. Today, the web is less static and predictable than ever before. The once common desktop monitor can now be a watch, a smartphone, a tablet, or a TV screen – and this list is likely to grow. Instead of being bound to the office-like setup of desktop computers, users can now be online everywhere: on airplanes, while commuting to work on the metro, or at home on the couch. [1]

> *"The long and short of it is that we're designing for more devices, more input types, more resolutions than ever before. The web has moved beyond the desktop, and it's not turning back." [1, p. 8]*
>
> *– Ethan Marcotte (2010)*

This change presented web developers with a big challenge. Their old longstanding approaches were not sufficient anymore. They had no choice but to adapt their workflow. The aim was to create more flexible websites that work on different devices. They came up with several approaches. The most sustainable is *Responsive Web Design* (RWD), which was introduced by Ethan Marcotte in 2010. It marked the beginning of a new era of web design, the responsive web movement. Marcotte used already available components – namely Flexible Grid-based Layouts, CSS Media Queries and Flexible Media – and combined them in a new and innovative way. The technique allows to create flexible websites which adapt automatically to the device used to access them. [1]

1.1 Motivation

The motivation for reviewing responsive images approaches is diverse. At the most basic level, images are great for many reasons. They allow to very effectively convey information and knowledge. As early as 1911, journalist Arthur Brisbane recommended *"Use a picture. It's worth a thousand words."* [2]. Images also serve an aesthetic purpose and make designs look appealing [3, p. 40]. That said, it is important that images are presented as clearly and efficiently as possible on whatever device they are displayed on.

However, the motivation for responsive images is rather technical. The RWD approach addresses images using the fluid images technique. It causes images to adapt to the size of its parent element and be more flexible. However, the same image files are delivered to all devices. On the client, the images are (down-)scaled to fit the screen. As a result, high-resolution and desktop-optimized images are delivered to small devices like smartphones. This is far from ideal and introduces a lot of overhead as the large image is downscaled on the client anyway. This overhead significantly extends the loading time of websites and thus, negatively affects the user experience. As of July 2015, the total size of the average website is around 2.2 megabytes. More than 1.3 megabytes or 63% thereof are images [4]. A test by Tim Kadlec in 2013 showed that delivering device-optimized images can reduce the image size of by to 72% [5]. These savings can shorten the page load time by tens of seconds when being on a slow mobile Internet connection. Thus, what developers want is to deliver different image versions depending on the device of the user. This is just one of many use cases for responsive images. More use cases will be covered in Section 3. Web developers introduced several approaches to solve the issues. All of them have advantages and disadvantages. A native browser-implemented solution has been developed and released in 2014. It can be considered the most sophisticated available approach and is recommended for most use cases. However, its browser support was bad at the beginning. The situation changed for the better, but is still not ideal. Until there is widespread browser support for the native solution, other approaches have to be considered as well. This work reviews six responsive images approaches. The aim is to provide a guideline and support for decision making when planning web projects.

A personal motivation of the author for responsive images is the longtime idea to create a travel website using fullscreen images. Fullscreen images cover the entire screen of the device and the above mentioned overhead is most evident for such big images. Thus, an evaluation of the available approaches seemed sensible before starting with the implementation. The entire idea and its prototypical implementation is covered in detail in Section 4.

1.2 Research Approach

The first steps were comprised of getting an overview of the research area. Relevant technologies and background information were collected by reviewing literature. The findings are presented in Section 2. These steps were necessary to conduct the following evaluation of responsive images approaches. First of all, the author tried to select some solutions for evaluation out of the pool of available approaches. This was done according to predefined selection requirements. It was found to be helpful to know the target solutions before planning the evaluation. The actual evaluation was performed using an evaluation framework. The framework was developed to ensure a formal and comprehensible proceeding. First, use cases for responsive images were composed. These use cases were transformed into functional requirements. Additionally, non-functional requirements and system prerequisites were chosen for evaluation. The result was an evaluation framework with seven functional requirements, six non-functional requirements and three system prerequisites. All six selected solutions were evaluated in an order that helps understanding. The evaluation was performed by reviewing literature and doing sample implementations. The results were used to select an approach for the following prototypical implementation of a travel website. The aim of the proof of concept is to demonstrate the implementation of future-proof methods.

1.3 Outline of Contents

The following gives an overview of this work's contents and the required knowledge. The target audience of this work is people with a strong IT and web background. Section 2 explains most of the required knowledge. However, it is out of the scope of this work to go into great detail. Intermediate skills in web development, especially with HTML, CSS, and JS, are most helpful to fully follow the contents. Apart from the Introduction Section, this work has four more main sections.

Section 2, covers the required knowledge to understand this work. It tries to convey the big picture of the responsive movement. This includes its history, the underlying technologies and the motivation.

Section 3 comprises the main part of this work. It provides a guideline for responsive images and covers different approaches. It starts with the basics and the motivation and then documents the evaluation of six responsive images solutions. The results are presented at the end together with a following discussion.

Section 4 covers the prototypical implementation of a travel website using fullscreen images. First, the idea is outlined and three fullscreen images approaches are introduced. Following next, on approach is selected and combined with a responsive images approach to implement the prototype.

Section 5 summarizes the findings, proposes a guideline and future research areas.

2 Background

This section covers the required background knowledge to understand this work. The presented topics help to grasp the big picture of the responsive movement and how it changed the web. Section 2.1 outlines the development of such a big technology like the web. Section 2.2 covers the essential technologies to create websites while putting the focus on aspects needed for being responsive. Web developers should know the currently used devices and their characteristics, which are covered in Section 2.3. The very relevant Section 2.4 outlines the history, motivation and implementation of responsive websites using RWD. The Background Section ends with the Section 2.5, which covers web-relevant aspects of images and sets the stage for Section 3: Responsive Images.

2.1 Evolution of the Web

Responsive techniques often make use of very new features. Before using them it is important to check their support by major browsers. To do so, it is helpful to understand how the web platform as a whole is being developed and enhanced. The web is one of the biggest and most broadly used technologies. There are many stakeholders with different aims and needs. It is not an easy task to develop and enhance such an extensive system.

In 1989, Tim Berners-Lee proposed an information system, which later turned out the be a great success story known as the *World Wide Web* (WWW, or just web). A big problem at the beginning was that there were no standards of what web browsers should support. Browser vendors had free choice for their implementation. This led to a fierce competition between individual browsers vendors known as the time of browser wars. The result were many compatibility issues, with many features only being supported by one browser. [6]

Tim Berners-Lee encountered these issues by founding the *World Wide Web Consortium* (W3C) in 1994. However, it took many more years before the situation changed for the better. After a long time of arguments and disagreement about HTML's successor XHTML, various companies like Mozilla, Opera and later Apple teamed up and founded the *Web Hypertext Application Technology Working Group* (WHATWG). Its aim was to create specifications (specs) and further enhance the web as a whole, without breaking its backwards compatibility. Eventually the W3C and the WHATWG agreed to collaborate for a new version of HTML. The result was called HTML5, which together with CSS3 currently represent two of the main technologies to create websites. [6]

As of today, the W3C and the WHATWG are the main organizations to set web standards. They coexist side by side and complement each other. Their goals are similar and their specifications are mostly the same. The W3C specs are considered as more stable, while the WHATWG ones usually implement the latest technologies [7]. The specs are developed together with many stakeholders, eg. discussions and feedback from the community and companies. Browser vendors mostly try to implement the specs, but are not obligated to do so. Instead, they have the final say. If the specs introduce a new feature but browser vendors decide not to implement it, the feature might eventually be removed from the spec again. This also works the other way round. If spec writers are to slow to address an issue, one or more browser vendors might come up with their own solution. As soon as a number of web developers start using it, there is no going back without breaking one of the web's fundamental principles: compatibility. One example is the `<meta>` tag to configure the viewport of the browser. The viewport meta tag was introduced by Apple for the first iPhone in 2007 [8]. It is now a de-facto standard and a small but integral part of RWD.

The web has been evolving for more than 20 years now. When the web was introduced, its requirements were quite limited. Its main purpose was sharing academic knowledge in a very simple and modest way. The web was designed with these requirements in mind. Over the years, the web has matured and its requirements have steadily increased. Developers wanted to create feature-rich web applications and websites which are user-friendly. The standardization organizations tried to address these enhanced requirements by inventing new technologies. It is a difficult task to enhance a platform which was designed with very limited requirements. The open and decentralized development does not always result in the best possible solutions. Asked which technologies he would remove from the web platform, WHATWG specification editor Ian Hickson put it this way in 2013: *"HTML, JavaScript, DOM, [...] basically anything that anyone uses! The Web technology stack is a complete mess. The problem is: what would you replace it with? [...]"* [7]. This is a tough judgment, but illustrates the big tradeoff of having a web which is free, open and backwards-compatible, all while being developed by many different stakeholders. This procedure is the reason for not having one technology which does it all, but a widespread pool of technologies. [7]

2.2 Web Technologies

Now that we have some background information, we can introduce the three main client-sided technologies used to build websites:

1. *Hypertext Markup Language* (HTML) for the semantic markup of the page,
2. *Cascading Style Sheets* (CSS) for the styling and formatting of a page, and
3. *JavaScript* (JS) for dynamic interaction with the user and the browser.

Many books have been written about each of these technologies and it is out of the scope of this work to cover each technology in great detail. Therefore, the following only outlines a minimal foundation in order to understand the following sections. References which provide a full coverage are given in the respective sub-sections.

Web technologies can be classified as client- or server-side technologies. This classification is not always definitive, but if a technology does most of its purpose on the client, its usually client-sided and server-sided otherwise. All of the above-mentioned technologies are client-sided which means that they are processed on the client. Before the client can do so, it uses the *Hypertext Transfer Protocol* (HTTP) to request the HTML document of the webpage. If it contains any other resources like CSS, JS or image files, they are requested as well.

In contrast, relevant server-sided technologies to create website are mainly scripting languages. Common languages are PHP, Ruby and Python. They can be used to implement server-sided logic, for example to modify the HTML document before it is served. These programming languages are too complex to be covered in this work. PHP is used for some examples in the following sections and for the prototype of Section 4. Some basic knowledge of PHP is helpful but the code is mostly self-explanatory.

People who are new to web development might wonder why there is not just one technology or language that does it all. The idea of having not just one builds upon the *separation of concerns* (SoC) principle. It is a common design principle in computer science. In the context of the web, it is mainly about separating the structure of the document (HTML) from its formatting (CSS). This has several advantages, some of which are also relevant for the later explained responsiveness. Separating structure and style results in more efficient code and eases its maintenance. Consider that we want to change the styling of headings on our website. When using SoC, style changes only have to be applied to the CSS file, instead of having to update every HTML file that contains headings. It also helps accessibility and device compatibility. As the HTML document contains just plain markup without any styling, it is easy to exchange the stylesheet. That way, a website's look and feel can greatly adapt to the needs of the device or the user. This would be much harder if the actual content and its formatting are tightly coupled together in the same document using just one technology. [9] A website called the CSS Zen Garden [10] greatly illustrates the power of separation. On the website the user can choose from a variety of designs, which are all applied just by changing the CSS file without touching the HTML code.

After introducing the web and scratching the surface of its technologies, it is now time to put them into practice and see how they work.

2.2.1 Hypertext Markup Language

We start off with HTML because it has been the main language to write web pages since the early days of the web. The current version is called HTML5. It is defined in the WHATWG [11] and W3C [12] specifications, which are identical for most parts. The term HTML5 can be confusing. It is also a buzzword for modern web development. When used like that it comprises more than just the HTML5 language specs, but a set of modern web technologies like several JS *Application Programming Interfaces* (APIs). [13, p. 6]

HTML code can be seen as the skeleton of a webpage. It is used to structure and semantically outline a document and its contents. This is done by enclosing content within so called HTML tags or just tags in short. Most tags are comprised of the starting or opening tag, its content and the ending or closing tag. For tags without content the closing tag can be omitted, eg. for the line break tag (`<br>`). Both the opening and the closing tag always start and end with angle brackets (`<` and `>`). The closing tag always includes a forward slash (`/`) after the first bracket (`<`). The opening tag can contain one or more attributes to provide additional information for the element like an ID or the target of a link. Such attributes are used by some JS-based responsive images solutions to store the URL of various image versions. Tags can be nested to create complex hierarchical structures. The following list outlines some frequently used HTML elements. The list is by no means complete, but going into every detail of HTML is out of the scope of this work. For a full coverage of HTML please refer to [13].

1. `<img>`: The `img` tag represents an image, whose source is declared in the `src` attribute, eg. `<img src="logo.png">`.
2. `<a>`: The anchor tag represents a link to another resource. The target is referred in the `href` attribute, eg. `<a href="http://google.com/">Go to Google</a>`
3. `<div>`: The `div` tag is a general-purpose element with no semantic meaning. It is often used to apply CSS style rules to the contained content.
4. `<h1>`, `<h2>`, ..., `<h6>`: The `h1` to `h6` tags represent hierarchical headings.
5. `<!-- -->`: Text between `<!--` and `-->` outlines a comment by the developer and is ignored by the browser. It can be used to describe and comment HTML code.

Browsers have default CSS stylesheets which apply some basic styling to tags according to their meaning. For example, a link (`<a>`) is displayed in blue and underlined and a headlines (`<h1>`) are displayed with a big font size and bold per default. In order to stick to the SoC principle, HTML tags should be applied for their meaning and not for their (default) style. That said, a `<h1>` tag should be used to outline its content as a level 1 heading and not to make some arbitrary text appear in a big font size. This results in semantically correct documents, which are important to increase the accessibility. The following listing shows a simple HTML document to help understanding.

```
<!DOCTYPE html>                          <!-- Define that this is a html document.       -->
<html>                                   <!-- html section, contains entire document.    -->
  <head>                                 <!-- Head section, contains document metadata.  -->
    <meta charset="utf-8">               <!-- Set document character encoding to UTF8.   -->
    <title>Home | Example Website</title> <!-- The document's title.                     -->
    <link rel="stylesheet" href="style.css"><!-- A referenced CSS stylesheet.            -->
    <script src="script.js"></script>    <!-- A referenced JavaScript file.              -->
  </head>                                <!-- End of the head section.                   -->
  <body>                                 <!-- Body section, this is the actual content.  -->
    <header>                             <!-- header section                             -->
      <img src="logo.png" alt="The Logo"> <!-- The website's logo with alternative text  -->
      <nav id="main-navigation">         <!-- Beginning of the document's navigation.    -->
        <a href="/" class="active">Home</a><!-- A link to /, showing as “Home”.          -->
        <a href="about.html">About</a>   <!-- A link to about.html, showing as “About”.  -->
      </nav>                             <!-- End of the navigation section.             -->
    </header>                            <!-- End of the header section.                 -->
    <h1>Welcome to Example Website</h1>  <!-- A level 1 headline.                        -->
    <p>A lot ... of ... text</p>         <!-- A paragraph with the main content.         -->
    <footer>© Example Company 2015</footer><!-- Footer section with copyright message    -->
  </body>                                <!-- End of the body section.                   -->
</html>                                  <!-- End of the html section.                   -->
```

Listing 1: *An example of a simple HTML document*

Assuming that the referenced style.css file does not contain any style information, this HTML document looks pretty plain when viewed in a browser. This is because only the browser's default styles are applied to the elements. The CSS language can be used to add some more customized styling and formatting.

2.2.2 Cascading Style Sheets

The work on CSS began already in 1994 to encounter HTML's very limited abilities to style and format content. The first recommendation was CSS version 1 (CSS1). It was released in 1996, CSS2 followed shortly after in 1998. After being in development for many years, CSS2.1 was released in 2011. While CSS 1 and 2 were documented in one specification, their successor CSS3 was split up in many different modules. Some of these modules have already reached recommendation status, while others are still in development. Nevertheless, CSS3 is considered to be the current version as many features are supported by most major browsers. [14, p. 9]

The purpose of CSS is to add formatting, styling and a layout to XML and especially HTML documents. CSS is very powerful and allows web developers to customize the appearance of a webpage in great detail. The syntax of a CSS stylesheet is simple. It contains one or more style rules, which are comprised of at least one selector and the following declaration block. Before styles can be applied to HTML elements, the desired target elements have to be selected. As the name suggests, a CSS selector does exactly that. After elements have been selected, the declaration block denotes a list of declarations to be applied. A declaration consists of a property, followed by a colon (`:`), followed by the value to apply. Multiple declarations have to be separated from each other with a semicolon (`;`). Again, a simple example will help understanding. [14, p. 35]

```
h1                                         /* This selector selects all h1 elements     */
{                                          /* "{" opens the declaration block           */
  font-size: 12px;                         /* Sets the font-size to 12px (CSS pixels).  */
}                                          /* "}" closes the declaration block          */
```

Listing 2: *A CSS declaration to set the font-size of all h1 elements to 12px.*

Listing 3 uses CSS to add some formatting to the HTML file of the last section.

```
html, body {
  margin: 0;             /* set the outmost margin of the box to zero                    */
  padding: 0;            /* set the margin between the box and its border to zero.       */
  font-family: Arial;    /* set the font family to Arial.                                */
  font-size: 0.875em;    /* set the font size to 0.875em, which equals 14px (0.875 × 16). */
                            it is calculated using the default browser font size (16px).  */
}

nav a {
  display: inline-block;      /* display each link next to each other                    */
  padding: 8px;               /* add a padding of 8 pixels to the outside of container   */
  background-color: #AAAAAA;  /* set the background color to a light gray color          */
}
```

Listing 3: *More CSS declarations to add formatting to the HTML document of Listing 1.*

Two fundamental concepts of CSS are its cascading and its inheritance model. The cascade handles conflicts if a CSS property is set to different values by two declarations targeting the same HTML element. The opposite problem occurs when no value is set for a certain property. Then the inheritance model tries to determine the value by traversing up the document looking for the closest parent element with the respective property set. If a value can be determined, this value is inherited from the parent element to the child. [14, pp. 93, 103]

As with HTML, it exceeds the scope of this work to go into great detail about CSS. Therefore [14] is recommended for further reading. That said, the following only covers two more aspect which are important for creating responsive websites: CSS media queries, and CSS units.

CSS media queries are part of CSS3 and allow the developer to apply CSS declarations only if certain conditions are met. These conditions are expressed using CSS media queries. The details are covered in Section 2.4.2.

They are interpreted as a length and are used to set widths, heights, margins, font sizes and many other CSS properties. For a long time the most-used unit was "px" – pixel. It is an absolute unit which gives the designer a lot of control. As we will see in Section 2.4, the desire to control the design greatly limits the flexibility of a website. In contrast, relative units like *%*, *em* and *rem* enable the web designer to set lengths relative to other lengths. This allows very adaptive and flexible designs. The differences will be clearer after going into detail about the relevant units for this work.

1. **px**: As said before, pixels are an absolute unit. That means that one pixel is always one pixel, no matter how big the parent HTML element or the browser viewport is. It is worth mentioning that the number of pixels set for a length do *not* necessarily equal the number of physical pixels used to display the content. This might sound confusing at first, but there are good reasons for browsers to behave like that as we will soon see in Section 2.2.5. [15]
2. **%**: Percent are a relative unit and therefore dependent of another length. Due to CSS' inheritance model, many properties (eg. font-size) are inherited from HTML elements to their children. This does also apply when setting a length in %: the actual value gets calculated by using the properties' value of the parent element as a reference. A short example will help understanding: Given that a 500px width `<div>` tag contains another `<div>` tag which is set to a width of 10%, the resulting width of the inner `<div>` tag is 50px. If the parent element changes its size, the relative-sized child will change accordingly as well. This relative sizing method greatly increases the flexibility and is one of the main parts of RWD, which is covered in Section 2.4. [15]
3. **em**: The em unit is a relative unit, which is calculated in relation to the element's or parent element's font size. A value of 1em equals 100%, 1.5em equals 150% and so forth. The reference length of an em value depends on which CSS property it is used with. If an em value is used to set the font-size property of an element, the unit is relative to the font-size of its *parent* element. If any other property than font-size is being set, em are relative to the font-size of the *current* element itself. The CSS em unit is based on the em unit which is used in typography, but both follow a different definition. [15]
4. **rem**: The rem unit stands for *root em*. It is also a relative unit and closely related to em. The difference to the em unit is the used reference length. Length set in em are calculated relatively to the font-size of the current or parent element. In contrast, lengths set in rem are *always* calculated relatively to the font-size of the root element of the HTML document, which is the `<html>` element itself. Thus, a given root em value resolves the an equal length in the entire HTML document, no matter which element or property it being set on. The HTML element has a default font size of

16px in all major browsers. This value is changed if a user adjusts the default font size to his needs in the browser settings. [15]

Relative CSS units are an integral part of creating flexible and responsive websites. Percentages are often used to implement fluid layouts, which are covered in Section 2.4. The em and rem units help accessibility as they are relative to the (root) font-size and therefore adapt if a user changes the default font size of the browser.

Now we learned about HTML to structure a document and CSS to style and format it. We will now cover the last main client-sided technology to create websites.

2.2.3 JavaScript

JavaScript (JS, also ECMAScript) is a dynamic, object-oriented programming language. At the time of writing, JS is the most popular language on GitHub [16]. JS can be used for many purposes and environments but is most-known for being the web's client-sided scripting language. Using just HTML and CSS results in rather static websites. Changing the content or appearance of a page or certain elements usually requires to reload the entire page. The server can then return a modified HTML and/or CSS document. JS encounters this inflexibility and can be used to make websites dynamic (without reloading the page). It is very powerful and provides a wide range of features to interact with the user, the HTML document, and the browser. JS uses the *Document Object Model* (DOM) to represent HTML documents and their elements as objects to the developer. Using these objects and their properties, a developer can change many details of the element: its content, appearance and behavior, among many more. JS also allows to dynamically load and display new content by using *Asynchronous JavaScript and XML* (AJAX). Back in the web 2.0 era AJAX used to be a popular buzzword. Today it is an integral technique of modern web development. It enables the developer to perform HTTP requests without reloading the entire page and thus, adds a lot of flexibility. [17]

As with HTML and CSS, JS is not equally supported by all browsers. Some browser vendors have introduced their own functions and properties. This caused developers to write separate code for different browsers, which is inconvenient and hard to maintain. To encounter this issue and simply coding in general, JS frameworks can be used. They offer wrapper functions which are compatible across major browsers. Under the hood, they use native browser functions or emulate them with custom code. Additionally, they simplify coding by offering functions for many common tasks, eg. element selection, DOM manipulation or AJAX requests. A very popular and powerful framework is jQuery by Google. It is being used on more than 50% of all websites. For further reading about JS and jQuery, [17] respectively [18] are recommended. [19, p. 98]

As mentioned in Section 2.2.1, the HTML5 specs not only describe the HTML language but also several JS APIs. Some of them are relevant to the responsive web. The *Geolocation API* acts as an interface to the device's location services to determine the location of the user [20]. The *Battery Status API* provides information about the current battery level in percent and remaining time until its empty [21]. *Ambient Light Events* can be used to react to the brightness of the user's environment [22]. The *Screen Orientation API* offers functionality to retrieve the orientation of the device, eg. landscape or portrait [23]. All these APIs provide a lot of information about the user: location, battery level, network connection, brightness and screen orientation. This data can be used to enhance the user experience of a website by reacting to the user's environment. In the responsive images context, a developer might deliver smaller images when the battery is almost empty. Another JS API relevant to this work is the *Fullscreen API* [24]. It allows to display a website in the fullscreen mode of the browser without the address bar and other user interface elements. It is used to enhance the proof of concept in Section 4. [19, p. 107]

A downside of JS is that it can not be relied on as it is not available on every client. It can be disabled by the user for various reasons. The loading and execution of JS take time and computational power, which might not be available for every user. Other users might disable it because they do not want an enhanced and fancy user interface for usability or accessibility reasons. Another motive to disable JS is security. JS can be used to track users or exploit security flaws, eg. using *Cross-Site Scripting* (XSS). Beside being manually disabled be the user, JS might not be supported by the browser at all. This holds true for very old browsers or lightweight browsers for devices with little computational power, eg. feature phones. Due to this unreliability, websites should always be created to still work without JS. This can be achieved by applying the *progressive enhancement* strategy. Its basic idea is to provide a basic but functional version of the website and then progressively enhance it while considering the supported features of the browser, device and user. [25], [26]

2.2.4 Hypertext Transfer Protocol

We have now discussed the client-sided technologies to create websites. The files containing the HTML, CSS and JS code are stored on a web server and can be requested by a client. The *Hypertext Transfer Protocol* (HTTP) is used for the communication between the web server and the client. The browser requests a webpage by sending a HTTP request to the server. Every HTTP request begins with the request line, which includes the request method and the URL of the requested resource. It can be followed by various headers containing additional information. The server's reply is called a HTTP response. It is made up of a status line indicating the result of the request and can be followed by headers as well. The following listing shows a typical HTTP request. The arrows of the request (`->`) and the response (`<-`) are not part of the communication but for the visual separation.

```
-> GET /index.html HTTP/1.1
   Host: www.example.com
   [blank line at the end of the request]
<- HTTP/1.1 200 OK
   Date: Wed, 22 Jul 2015 16:44:34 GMT
   Content-Type: text/html; charset=UTF-8
   Content-Length: 153
   [blank line to separate headers and content]
   <html>
     <head>
       <title>A Webpage</title>
     </head>
     <body>
       This is a very simple HTML document.
     </body>
   </html>
```

Listing 4: *A HTTP request from the client and the HTTP response from the server.*

The following covers some aspects of HTTP which are relevant to being responsive. First, we will have a look at the User-Agent header. A user agent is software which acts on behalf of the user, eg. a web browser. Most browsers include this header in their requests to a server. This header's content is not standardized but usually contains the client's operating system, browser version and sometimes additional information. This information can be used to deliver appropriate images to the client. The concept of using the User-Agent header to learn more about a client is known as *user agent detection*. This approach is used by Solution 1 of the later on evaluated responsive images solutions. [27, Sec. 5.5.3]

Another useful feature of HTTP are cookies. Cookies allow the web developer to store small amounts of data within the user's browser. Cookies justifiably have a bad reputation because they can be used to track users across websites. However, they also represent an integral part for the functionality of most modern websites by defeating HTTP's statelessness. A browser can be instructed to store a cookie from the server-side using the HTTP header Set-Cookie or from the client-side using JS. After a cookie has been set, it is automatically included in every request to the same domain. In the responsive images context, cookies can be used to let the server know about certain device characteristics. This is done by storing device information in a cookie, which is then sent to the server with every subsequent request. This method is used in the responsive images Solution 2 in Section 3.4.3. [28]

Furthermore, HTTP supports a mechanism called content negotiation. It is used to deliver a different or modified version of a resource for the same URL. This is done using the Accept and Accept-Language headers. The client can supply these headers to advertise supported and preferred file formats or languages of the content. For example, Google's browser Chrome advertises its support for Google's new image format WebP by adding the Internet media type *image/webp* to its Accept header. An enhanced implementation of content negotiation is used in Solution 3 of the evaluated responsive images solutions in this work. [27, Ch. 5.3]

HTTP and most browsers support compressed transfers using the gzip algorithm. It is recommended to configure the web server to compress responses to speed up loading times. Another method to speed up websites is caching, which can be done locally by the browser or externally by proxy servers. The basics of both browser- and proxy-based caching are covered in [29]. Proxy servers are intermediary hosts between the web server and the client. They can cache HTTP responses and serve this cached response without consulting the web server again on subsequent requests. There are use cases where caching is not desirable, especially for personalized or sensitive content, eg. when doing online banking. The HTTP specification provides several headers which can be set by the web server to configure the behavior of intermediary proxy servers. These headers are also relevant when deploying the before-mentioned content negotiation mechanism. We learned that the web server can deliver different resources for the same URL. To make caching work, it is required to inform the proxy server about how the resources differ and when to serve which resource. This can be done using the Vary and the Key HTTP headers. The Vary header indicates which request headers were used to select a resource. For example, `Vary: Content-Encoding` tells the proxy server that different content is delivered based on the value of the request's Content-Encoding header. The Vary and Key HTTP headers are not applicable to all responsive images approaches and not all proxy servers support them. Thus, when using server-sided responsive images solutions as introduced in Section 3, intermediary caching by proxy servers might not work. It should be considered to add the `Cache-Control: private` header disable intermediary caching at all. This prevents proxy servers from delivering wrong resources. [27, Sec. 7.1.4], [29], [30, Sec. 13]

2.2.5 Browsers

A web browser, commonly only called browser, is the software to access web pages. Most common browsers have similar features and *graphical user interfaces* (GUI), with an address bar at the top, optionally some toolsbars and the content area (viewport) below. The most-used browsers are often referred to as major browsers, which are Chrome, Firefox, Internet Explorer, Opera and Safari. Many developers create websites with these browsers in mind. However, it should not be forgot that there are other types of browsers as well. Visually impaired people may use a voice browser or a braille display. Others may be limited to a very minimal browser due to slow hardware, eg. on old feature phones. Besides, web pages can also be accessed using a text-only command line browser or be processed by a computer (also called bot). For all users of these less common user agents it is important that the content is outlined and presented in an accessible way. This includes semantically correct HTML code, no dependency on JS and implementation of accessibility features. This work mainly focuses on the major browsers, but also addresses accessibility issues when appropriate.

The most-used browsers aggregated for all device categories are Chrome (45.1%), Safari (13.2%), Internet Explorer (10.8%), Firefox (10.0%), Android Browser (6.6%), UC Browser (5.6%), Opera (5.5%) and Others (3.2%) [31]. The UC Browser is not common in western countries, but is the biggest mobile browser in China. Market shares for different device types are given in the following section. The operating system and browser market share data in this work are based on page hits and were fetched from [31]. Great caution should be applied when comparing browser market share statistics from different sources. They can greatly differ due to different methods of measurement, eg. based on unique visitors vs. page hits, and possibly due to regional differences or target audiences with preferences for certain browsers. [32]

Browsers internally use a layout engine to parse and render web pages. Popular engines are Gecko, used by Firefox; Trident, used by Internet Explorer; WebKit, used by Safari; and Blink, a fork of WebKit used by Chrome and Opera. The used engine is relevant when it comes to feature support, as support is similar for browsers with the same engine (version). Compatibility and support of HTML and CSS features are a big issue for developers. Hardly any specification is fully implemented and works consistently across all devices or browsers. Instead, it is one of the key skills of a web developer to be up to date about compatibility issues and know how to implement workarounds. Especially Microsoft's Internet Explorer has for a long time caused issues due to many bugs and incompatibilities in older versions. That has changed for the better and since the release of version 10 it can be considered a modern browser. There are several services which help checking browser support for a certain feature [33], [34]. As this work covers a lot of new features, compatibility matters a lot. Knowing the evolution of the web, as described in Section 2.1, can be most helpful when evaluating the current support for a certain feature or predict its final implementation. [13, p. 27], [35, p. 7]

A relevant aspect of browsers when it comes to responsive websites is how a web page is loaded and rendered. When the user initiates the request of a web page, the browsers sends a HTTP GET request to the web server for the given URL. The server returns a HTML document which contains the requested page. The HTML code also contains references to other resources like CSS, JS and image files. The browser starts parsing the received HTML code to generate the DOM. Whenever it encounters a `<script>` tag, the parser by default halts to execute the script. Halting is necessary as the script could modify the following HTML code. If the script is referenced as a separate file, the browser has to retrieve it before it can be executed. Additionally, the browser tries to load all referenced CSS files before it executes any JS. This is done because a script could be dependent on style information. Building the DOM, fetching CSS files and executing JS code takes a reasonable amount of time. To increase the page loading speed, most browsers start a second parser called lookahead or pre parser together with the main parser. Its purpose is to parse the document without creating a DOM, but only to determine external resources. These resources can then be preloaded using another thread, while the main parser continues its work. As a result, resources may already be loaded and available by the time the main parser reaches the related HTML code. This mechanism is also referred to as speculative parsing, as the browser only guesses that the resource will be needed. It is good practice to put `<script>` tags just before the closing `</body>` tag. This prevents the main parser from halting to load and execute the JS code. Alternatively, the `async` or `defer` attributes can be added to `<script>` tags [19, p. 90]. They instruct the browser to load the JS file asynchronously without blocking or defer its loading and execution to after the DOM has been fully created. For an in-depth coverage of how browsers work internally, [36] is recommended. [37]

JS provides two types of events to indicate the loading state of the page. The DOMContentLoaded event is fired by the browser when the DOM has been fully loaded and created. It is good practice to wait for this event to fire before performing any DOM manipulations. The DOM being ready does not imply that all external resources have been retrieved. This is what the OnLoad event is for. It fires when the DOM is ready *and* all resources have been loaded. The OnLoad event can also used for individual `<img>` tags. The attached event handler gets executed when the image has been downloaded. [38]

Another essential concept to understand is the viewport of the browser. Generally speaking, the viewport is the area of the browser window which is actually used to display websites. Technically, it contains everything of a page, including the HTML element. Therefore it is called the *initial containing block* in the CSS specifications. Furthermore, it acts as a constraint for webpages by limiting the HTML element to the width and height of the viewport. This is necessary to provide an absolute reference for relative CSS units like percentages, which are always relative to its parent element. If a website does not fit within the viewport, the browser adds scrollbars so that the user can see all parts of the webpage by scrolling.

The described viewport behavior is how it works on desktop-like systems. However, on mobile devices the viewport concept is a bit more complex. When the first smartphones with full-fledged browsers were released, most websites were still optimized for desktop systems. Due to the small screen size of mobile devices, these websites would have been barely usable if they were rendered in the same way as on desktop systems. This is especially evident for fluid percentage-based layouts.

For example, a sidebar with a width of 30% looks fine on a big screen but squeezed on a smartphone screen in portrait mode. The phone vendor's goal was to provide a good browsing experience, which back then basically meant to be "*as much like desktop as possible*" [39]. They solved the issue by splitting the viewport concept into two viewports. [40]

The *layout viewport* acts as a constraint for the layout. It is given a (virtual) width which is similar to a desktop system, eg. 980 pixels on the Apple iPhone. Other vendors usually assigned widths between 800 and 1024 pixels, depending on the device and the browser. In consequence, the layout is rendered relatively to the size of the layout viewport and thus, similarly as on desktop systems. The width of the layout viewport can be controlled by the developer using a `<meta>` tag, which will be covered in Section 2.4. The second new viewport – the *visual viewport* – represents the part of the layout viewport which is visible to the user at a given moment. The visual viewport changes its size with zooming, while the layout viewport's size is fixed and not affected by zooming. [39], [41] George Cummins explains the concept using a real-world comparison.

> *"Imagine the layout viewport as being a large image which does not change size or shape. Now image you have a smaller frame through which you look at the large image. The small frame is surrounded by opaque material which obscures your view of all but a portion of the large image. The portion of the large image that you can see through the frame is the visual viewport. You can back away from the large image while holding your frame (zoom out) to see the entire image at once, or you can move closer (zoom in) to see only a portion. You can also change the orientation of the frame, but the size and shape of the large image (layout viewport) never changes." [42]*
>
> – *George Cummins (2011)*

With the introduction of *high dots per inch* (HiDPI) displays another issue arose. Those are screens with a relatively high pixel density of usually more than 240 *pixels per inch* (PPI). A higher pixel density requires (a) pixels to be smaller and/or (b) less space between individual pixels. That would result in websites looking tiny and illegible on these screens. CSS encounters this issue by defining CSS pixel (px) as an abstract unit, generally referred to as *device independent pixel* (DIP). The result is that one CSS pixel does not equal one device pixel, but for example 1.5 or 2 device pixels. The ratio of device pixel and CSS pixel is called the *device pixel ratio* (DPR). Please note that the CSS pixel unit is a linear measure of length, so a DPR of 2 results in 1 CSS pixel mapping to an area of 4 ($2{\times}2$) device pixels. [39]–[41], [43]

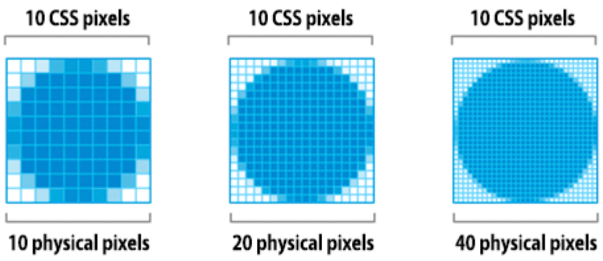

Figure 1: *Comparison of a DPR=1 screen (left) and HiDPI screens with a DPR of 2 respectively 4. [77]*

HiDPI screens caused the introduction of another, third viewport by Peter-Paul Koch – the *ideal viewport*. The ideal viewport represents the ideal viewport size to display a website on a device, considering the device's pixel density and the usual distance from the user to the device. It can be seen as a device-specific, DPR-corrected size of the physical resolution. For example, a tablet with a physical resolution of 1536×2048 and a DPR of 2 would have an ideal viewport size of 768×1024 (1536/2×2048/2). [44]

The DPR concept is also used for the zoom function on desktop browsers. Although the visual size of all elements on a page increase when zooming in, their widths in CSS pixel remain the same. What changes is only the browser's DPR and the viewport width. When zooming in, the DPR increases and the viewport consequently gets smaller. When zooming out, the DPR decreases and the viewport gets bigger. On mobile devices, the DPR does not change and the layout viewport has a fixed width, but the size of the visual viewport changes when the user zooms [45, Sec. 3.2], [46].

The DPR also affects the rendering of images, which is covered in Section 2.5.

2.3 Diversity of Devices

Within the last few years, the number of devices which are online has vastly increased. In year 2008, there were more Internet-connected devices than individuals who used them. Naturally, web developers and designers should have an overview of the devices on the market. There are several relevant *device characteristics* like screen size and resolution, hardware capabilities, operating system and the default browser, to name a few. Due to their increased popularity smartphones have been of special interest for the past few years. With almost 19.000 distinct Android devices in the wild and a plus of 6.900 new models in year 2014 alone, it is impossible to know all devices and their characteristics [47]. As many devices are very similar they can be categorized for simplicity. In the following, we will distinguish between five categories: *Desktops and Laptops*, *Mobile Phones*, *Tablets*, *TVs*, and *Others*. Due to the sheer number device variations and screen sizes, this categorization is not always definitive. For example, a smartphone with a 7 inch display could also qualify as a tablet – in that case some people use the term Phablet, a combination of phone and tablet. [19]

2.3.1 Desktops and Laptops

Desktop computers and laptops are still the most-used device to browse the web [48, p. 9]. They usually have big screens, powerful hardware and a fast Internet connection. Desktop systems are mostly used while seated, often at home or in an office. Laptops are compared more mobile, but still used in a similar context usually and thus, the following is valid for both desktops and laptops. Before the era of smartphones, websites were developed exclusively desktop-like systems. Assuming a minimal available resolution of 1024×768 pixel worked well when creating websites. Today, this resolution is still common for legacy screens, but most new monitors have higher resolutions of up to 2880×1800 pixel (Apple's MacBook Pro) or even more. [19, p. 2]

As of July 2015, the most common desktop operating systems are Microsoft Windows (87.2%), Apple OSX (8.6%) and Others (4.2%). Microsoft Windows XP still has a market share of 10.0% and is mentioned separately because when used with Microsoft Internet Explorer the highest available browser version is 8. Windows XP together with Internet Explorer is still a common setup, especially in business environments. The most-used desktop browsers are Google Chrome (55.4%), Microsoft Internet Explorer (18.9%), Mozilla Firefox (17.3%), Apple Safari (4.7%) and Opera (1.9%) as of July 2015. [31]

Common screen resolutions are 1366×768 (30.9%), 1920×1080 (12.8%), 1024×768 (8.2%), 1280×800 (6.4%), 1440×900 (6.3%), 1600×900 (6%), 1280×1024 (5.9%), 1280×1024 (5.9%) and Others (23.5%) [31]. Great caution should be applied when examining resolution statistics. The `screen.width` JS property which is used by most analytics services, including the one used for this work, is not reliable. It behaves correctly and returns the number of horizontal physical device pixels in most desktop browsers. However, in mobile and tablet browsers `screen.width` often returns the size of the ideal viewport. As described in Section 2.2.5, the ideal viewport is a device-specific and DPR-corrected size of the screen. It does not necessarily equal the number of physical device pixels. [49], [50]

Especially in home environments, devices like tablets or smartphones became a serious alternative to desktop computers within the last years. Since two years the market for personal computers has declined, 5.2% alone in the first quarter of 2015 compared to last year [51]. Despite this tendency, desktop computers will of course still be around in the future [52]. Users prefer to use desktop systems for productive and task-oriented work or actions where a big screen and close control is beneficial, for example text-intensive tasks or image editing. The feeling of security is also higher when using a desktop system, for example when doing online banking. [19, p. 3]

2.3.2 Mobile Phones

The next category we look at are mobile phones, or just mobile in short. In general, older feature phones and smartphones are likewise counted to this category. This work mainly focuses on smartphones as the most of the presented techniques and approaches require a reasonably modern browser and hardware. That is usually not the case for feature phones.

The smartphone market has been doing very well since the beginning of its boom about 8 years ago. In 2013 there were 1.3 billion smartphones in use and this number is expected to double by 2018 [53]. This trend led to many new models being introduced with continuously improved hardware and software. Many users change their devices very frequently in cycles of 12 to 18 months. Modern smartphones often have a computational power similar to budget desktop computers. In [54], Jonathan Stark also outlines benefits that go beyond that:

> *"[...] smartphones are actually more powerful than desktops in many ways. They are highly personal, always on, always with us, usually connected and directly addressable. Plus, they are crawling with powerful sensors that can detect location, movement, acceleration, orientation, proximity, environmental conditions and more." [54]*
>
> *– Jonathan Stark (2012)*

All of these capabilities should be considered when planning and creating a website. The increased mobility of these devices also greatly influenced the user's behavior. The user is able to fetch information, get directions, do online shopping and much more – all while being on-the-go. But according to a research by Google, smartphones are also used at home 60% of the time [48]. Often for *simultaneous screening*, a usage pattern which describes the usage of two screens at the same time, for example watching TV while using the smartphone.

As of July 2015 the most common mobile operating systems on phones were Android (64.1%), iOS (20.4%), Nokia's Series 40 (3.7%), Windows (2.3%), Blackberry (1.2%) and Others (8.3%). The most-used browsers are Chrome (33,8%), Safari (19.0%), Android Browser (15.2%), UC Browser (14.93%), Opera (11.63%), IEMobile (2.1%), BlackBerry (0.9%), Nokia (0.7%), Firefox (0.35%) and Others (2.5%).

The UC Browser is not common in western countries, but is the biggest mobile browser in China. [31]

Common screen resolutions are 360×640 (19.9%), 480×800 (10.2%), 320×568 (9.4%), 320×480 (6.0%), 720×1280 (5.3%), 375×667 (4.9%), 320×534 (3.9%) and Others (40.6%) [31]. The large *Others* value depicts the great fragmentation of screen resolutions in the mobile phones device category. Smartphones are often shipped with HiDPI displays, which are displays with a pixel density of usually more than 240PPI. These HiDPI displays caused the `screen.width` property to be changed, which makes the above resolution statistics unreliable. [49], [50]

While feature phones are not within the scope of this work, they should generally be considered when talking about later-covered responsiveness and user experience. Especially in Africa, but all over the world there are countries where feature phones are still the main devices to access the Internet [55]. Developers should be aware of that and make their websites as accessible as possible. [19, p. 3]

2.3.3 Tablets

The tablet market has developed in a similar way as the mobile market. While the concept of tablets has been around for many years, they only got popular with the introduction of the first iPad by Apple in 2010. Since then many different models have been introduced. Most tablets are shipped with a similar hardware than smartphones but with considerably bigger screens of usually 6 to 10 inch. That said, tables are not as mobile as smartphones but still portable. This is also reflected in usage statistics with tablets being used at home 79% of the time compared with only 60% for smartphones [48]. The same research shows that the main motivators for using tablets are entertainment and browsing, often in a relaxed and time-unbound setting. [19, p. 5]

The operating systems and browsers are similar to those of smartphones, but their market shares vary. For operating systems, Apple leads with iOS (66.4%) followed by Google's Android (29.7%), Linux (2.6%) and Microsoft Windows (1%). The browser landscape matches the OS distribution: Safari (59.6%), Chrome (17.0%), Android Browser (16.2%), Amazon's Silk Browser (2.6%), UC Browser (1.17%) and Others (3.5%). The market share data is from July 2015. [31]

Common tablet resolutions are 768×1024 (59.7%), 800×1280 (9.3%), 600×1024 (7.4%), and Others (23.7%). [31] The screen resolution statistics are subject to the same inaccuracy as mobile screen resolutions due to the unreliable `screen.width` behavior. Especially the high 768×1024 value, which correlates with the iOS market share, might not be accurate. The Apple iPad version 3 and after are shipped with a HiDPI display having a resolution of 1536×2048 and a DPR of 2. The `screen.width` property returns the size of the ideal viewport, which is still 768×1024 due to the DPR of 2. [49], [50]

2.3.4 TVs

Televisions are perhaps not expected in a list of web-enabled devices. Despite that, many current TV models can be connected to the Internet and come with an integrated browser. In 2010, 24% of all US households had at least one Internet-connected TV. This number doubled to 49% in 2014 and further growth is expected [56]. These so called smart TVs are shipped with powerful hardware and the browsers have a good support for modern features [57]. The common screen sizes range from 30 to 60 inches with resolutions of 1920×1080 (Full HD) or up to 3840×2160 pixel for new Ultra HD displays. The next generation, *Full Ultra HD* (FUHD) televisions with a resolution of 7680×4320 pixel are already planned. It should also be mentioned that even older TVs without web support can be connected to the Internet using devices like Apple TV or Google Chromecast. [58]

No statistics of operating system or browser market shares and the used resolutions are available at the time of writing.

A big issue with smart TVs is the input method. Using an ordinary TV remote control to navigate through a website or enter text does not provide a good user experience. Manufacturers came up with different solutions: special keyboards, gesture or voice control and more. A practical and universal approach is to pair a smartphone or tablet with the TV and use it as an input device. [19, p. 5], [59]

Furthermore, privacy issues may arise when browsing with a smart TV. Those devices are usually very big and often set up in a shared room, eg. the living room. Thus, the displayed content may be visible to more people than just the intended consumer. This should be considered when creating websites or applications. In general, other style guidelines apply when designing and developing for such big screens compared to desktops or mobile. Jason Grigsby gave an excellent talk [59] on developing for TVs and Opera provides a good pool of resources as well [60]. [19, p. 5]

2.3.5 Others

We have now covered four common device types. More types exist but are not frequently enough used to categorize them. Most currently popular gaming consoles are shipped with an integrated web browser. Sony's Playstation 4 and Nitendo's Wii U come with custom browsers based on the WebKit engine. Microsoft's XBox One runs Internet Explorer 10. All three browsers can be considered (semi-)modern and have decent feature support. Most older consoles have poor to no support for the web. The shortcomings of TVs apply for gaming consoles as well. [19, p. 6], [61]

Another relevant device type are e-book readers. Their computational power is often very limited and they usually come with monochrome displays which have very low refresh rates. Due to this combination it is a not very user-friendly way to browse the web. [19, p. 6]

The last covered device type are so called wearables – devices which can be worn on the body, eg. watches or glasses. Especially smart watches have gained popularity within the last two years [62]. The can be paired with a smartphone to allow easy access to common functions like reminders, text messages and others. A known glasses project is Google Glass.

The list of covered device types and categories is by no means complete. The intention was to show that there are many different devices which need to be considered when planning and implementing a web project. The boom of smartphones back in year 2007 encouraged developers to come up with new approaches to target different devices. That marked the birth of a new movement in the community which is today known as the responsive web.

2.4 Responsive Web

Firstly, it is important to know that there are several meanings of being responsive or responsiveness. The web performance community also uses the term responsive to refer to speed. While speed and performance is also an important aspect in this work's context, the terms are used in another context. The term *Responsive Web Design* (RWD) was coined in 2010 by Ethan Marcotte and is a well-defined technique [63]. It is not that simple with the terms *responsiveness, responsive website* and *being responsive.* All three have refer to the same concept, but there are ongoing discussions in the community about what they actually comprise [64]–[66]. All terms will be explained and discussed in detail later on. For now, we generalize to help understanding. Being responsive means to react or answer to something. In the context of this work the reaction is about responding to the user's device and needs on a website. Most people who own a smartphone have probably experienced responsiveness already when browsing the web. Some websites look different on a smartphone compared to a desktop system. The most obvious difference on small devices is usually the layout. Multiple-column layouts may collapse to a layout with less or even just one column. Other parts of the website like the navigation may be initially hidden and only shown after clicking a “Show navigation” button. This behavior on a website is usually described as being responsive. This visual aspect is the most obvious, but not only part of responsiveness. Instead, responsiveness refers to the big picture and concept of developing for multiple device types and providing a good user experience. RWD is one concrete technique to implement responsive websites.

2.4.1 Evolution of Design

Before going into detail about the definitions, it is helpful to understand the history of print design and the transition to web design. When designing for print, there are certain physical boundaries. For a painter it is the size of the canvas and for the layout designer of a news paper it is the size of the page, just to give two examples. Within these boundaries, the designer is in control. It is fixed that the resulting print is the only way the user can consume the content. In contrast to the web, the consumer of a printed document can not change the size of the page, the font size, colors or zoom the entire document. Print designers learned to deal with these limits and design with them in mind. This approach totally makes sense for print media. [1]

In comparison, the web is very different to print. As explained by John Allsopp, when new mediums arise, the new one usually borrows from the existing one [67]. Allsopp takes the example of early television, which was kind of "*instructive viewing*" or "*radio with pictures*" *[67]*. It did not take advantage of the new medium's possibilities. The same holds true for the transition from the print medium to the web medium. When the web emerged, designers tried to apply the print workflow to it. The new boundary was not the canvas or the size of the printed page, but the size of the browser viewport. Beside that, the consumer has taken over control from the designer. The consumer can easily adjust the font size, colors and other styles to fit her preferences and needs. Furthermore, in the web the viewport size can easily be changed by resizing the browser window or zooming the content. [67]

Before the era of smartphones, most of the print workflow was not good, but sufficient to serve the average user. The variety of displays and resolutions was not very diverse. Most users had at least a certain value of pixels in width. In year 2004 the most common resolution was 1024×768. It then became common practice for designers to assume 1024×768 pixel as the minimum available resolution when creating a website [68]. Designers had this minimum width in mind and used the absolute CSS unit pixels to create websites. Back then, the result worked well for most users. It was the time when notes like "Optimized for a resolution of 1024×768" were common. However, there are various downsides of this approach. As soon as one or more constraints change, for example the user resizes the browser window, the layout could break. In this context breaking means that horizontal scrolling is needed in order to see all of the content. This is considered a bad usability and thus, a poor user experience. The resulting websites were not flexible at all, while the web inherently is. The change of the device landscape due to smartphones forced designers to adapt to this flexibility of the web. The idea was not new. The before-mentioned article [67] already praised the web's flexibility in comparison to traditional print in year 2000. It is an excellent and strongly recommended article which outlines the idea behind responsiveness very well – more than ten years before it was broadly put to practice by web designers:

> *"The control which designers know in the print medium, and often desire in the web medium, is simply a function of the limitation of the printed page. We should embrace the fact that the web doesn't have the same constraints, and design for this flexibility. [...] Everything I've said so far could be summarized as: make pages which are adaptable. Make pages which are accessible, regardless of the browser, platform or screen that your reader chooses or must use to access your pages. This means pages which are legible regardless of screen resolution or size, or number of colors (and remember too that pages may be printed, or read aloud by reading software, or read using braille browsers)." [67]*
>
> *– John Allsopp (2000)*

When this article was written in year 2000 it was a very forward-thinking approach. However, technology lagged behind. Not many methods to make websites more flexible were available and the existing ones lacked browser support. Instead, web designers were struggling to work around various browser incompatibilities at that time and were not concerned about making websites flexible. There was simply no need for flexible websites, as the user environment was stable and predictable.

This was slowly changing to the better, but only the introduction of smartphones caused an industry-wide rethink. Especially the release of the first Apple iPhone in 2007 was an influential event. The assumption that most users have screens with at least 1024 pixel in width was no longer true. New devices emerged over time and as we have seen in the last section, designers have to deal with a greater fragmentation of screen sizes and resolutions than ever before. Displays and resolutions are steadily getting smaller *and* bigger – speaking mostly from wearables or smartphones on one end and desktop monitors or TVs on the other. Since a few years, designers have had no choice but to adapt to building more flexible websites as the market is demanding them. [1]

Various approaches were invented to achieve the goal of creating websites which are optimized for different devices. One approach is to create a separate website for mobile devices. Those separate sites are referred to as *m-dot* sites, because often they are reachable at `http://m.<site>.<tld>`, eg. `http://m.facebook.com`. Another approach is to create a native smartphone app. These alternatives are discussed later on in Section 3.5.2.

2.4.2 Responsive Web Design

After some years of many different approaches, in year 2010 Ethan Marcotte invented a technique he named *Responsive Web Design* (RWD). In a popular *A List Apart* article he defined it as the combination of three components: flexible, grid-based layouts, CSS media queries and flexible images [63]. Marcotte's book on the topic was released about a year later [1]. All three components will be used for the proof of concept in Section 4.

1. **Flexible, grid-based Layouts:** The foundation of a RWD website is its flexible layout. It is implemented by using the CSS percentage unit to set the widths of the layout. This simple method causes a big difference to using the absolute pixel (px) unit. Due to the nature of the relative percentage unit, the layout will automatically adapt to the size of the viewport. This method alone is already very powerful, but usually not enough. Presentation issues may arise especially on very narrow viewports. For example, a 3-column layout with a lot of text will likely look squeezed on a mobile device when viewed in portrait mode. Additionally, the layout adapts but the other properties like the font size or margins remain unchanged. Usually some more adjustments are needed to make websites work well on small and big screens. This is what CSS media queries are used for. [1, p. 17]
2. **CSS Media Queries** are part of CSS3 and allow the developer to apply CSS declarations only if certain conditions are met. These conditions are expressed using CSS media queries. A media query can consist of media types and media features. Available media types are screen, handheld, tv, print, braille, speech, among others. The handheld and tv media types could prove very useful for responsive design. However, the only applicable media type for computer displays is screen. This is due the very inconsistent implementation of the handheld and tv media types in all major browsers [69]. The print media type is used to set styles for printing a page and screen is for all types of computer screens. Media features allow to refine media queries and check for more specific conditions. The most relevant media feature for RWD is *width*, especially its variants *min-width* and *max-width*. They enable the developer to apply separate CSS declarations targeting only certain viewport widths. When gradually reducing the size of the viewport, a design usually needs adaption when falling below certain widths. While usually not as severe, this also holds true when the width of the viewport is increased. The developer can use media queries to adjust the design accordingly. The widths at which a design changes its appearance are called breakpoints. [70]

For example, the following media query could be used to target screens below 320 pixel in width. It changes the layout from a 3-column layout to a single-column layout. Web developers would speak of a breakpoint at 320 pixels.

```
@media screen and (max-width: 320px) {
  .column {
    width: 100%; /* Change width from 33.3% to 100% (3 → 1 column layout) */
    float: none; /* Show columns one below another instead of side by side. */
  }
}
```

Listing 5: *A CSS media query to change from a 3- to a 1-column layout.*

3. **Flexible images and media:** This method is also known as the fluid images technique, but can applied to other media like videos as well. It consists of a very short piece of CSS: `img { max-width: 100%; }`. This code causes images to be automatically scaled to fit into their container element. Section 3 covers flexible images in detail and explains why this approach alone is usually not enough to create well-working and efficient websites. [1, p. 45]

All three components of RWD use client-sided technologies. Sometimes it is sensible to use responsive approaches on the server as well, which is then referred to as *Responsive Design with Server Side components* (RESS). However, RESS is not a part of the RWD approach per definition. A server-sided example is user agent detection, which is used in a solution of Section 3.

To make RWD work, the viewport of the browser has to be set up correctly. In the Section 2.2.5 we learned that mobile devices render sites using a (virtual) layout viewport of usually around 980 (again, virtual) pixels in width. This is done to make desktop-optimized websites render correctly on mobile devices. A website built with RWD does not need this wide viewport because they are optimized for small screens already. Instead, the layout viewport should be as wide as the ideal viewport. This can be configured using a special `<meta>` tag introduced by Apple for the first iPhone [8]. The tag of the following listing causes the device's layout viewport to be set to the size of the ideal viewport. For example, the layout viewport might be set from 980 pixels in width to 360 pixels in fully zoomed-out presentation. As a result, the initial page representation after loading the page shows 360 CSS pixels in width and percentage CSS units are calculated relatively to this width. [44]

```
<meta name="viewport" content="width=device-width, initial-scale=1">
```

Listing 6: *The viewport `<meta>` tag to set the layout viewport of (mobile) browsers.*

RWD got one of the biggest movements in web development since the late 90s under way [71, p. xi]. Due to its clear definition, web designers and developers managed to change their workflow. By using the three proposed technologies it was possible to create more flexible websites which adapt to the size of the device they are displayed on. RWD and *being responsive* has become a big buzzword in the web community since then. Many clients demand a responsive website, sometimes without even knowing what it means. Many people with different backgrounds talking about the same thing can easily cause misunderstandings. While RWD is well-defined, *responsiveness* is not. At first, responsiveness was used as a synonym for RWD. But soon after the introduction of RWD, designers and developers realized that often more than the three components of Marcotte's approach are needed [72], [73]. For a good user experience, more than responding to the size of the device is necessary.

2.4.3 Meaning of Responsiveness

As mentioned at the beginning of the Section 2.4, there are ongoing discussions in the community about what is needed to be responsive [64]–[66]. Jason Grigsby argues that a website can be responsive even if it does not use all three ingredients of RWD. What matters is if it works well for the user: "*I take comfort ignoring the definitions and instead asking these questions about a design and its implementation: Does it adapt to screen size? Does it take advantage of device capabilities? Is it accessible anywhere? Does it work well?"* [64]. In [72], Tim Kadlec suggests that a new term like *future-friendly* may be needed to describe the big picture of modern web design. He also proposes that changing the meaning of the already so common term RWD will cause even more ambiguity and misunderstandings. In [66] Jeffrey Zeldman reasons that a very clear and easy to follow concept like RWD was needed back in 2010. It caused designers to actually use it and change their workflow. This would perhaps not have been the case if it was a vague description of the big picture. However, he agrees that it is time to rethink responsiveness and that it comprises of more than just flexible layouts, flexible images and media queries. Scott Jehl picks up these discussions in his book *Responsible Responsive Design* [74]. Adding *responsibility* to RWD is a well-conceived idea. Every web designer and web author carries great responsibility as their way of creating a website is crucial to how their content can be consumed. Jehl suggests to combine RWD with the following keywords to deliver a "*broadly accessible, delightful and sustainable web*": usability, accessibility, sustainability and performance [74]. Matthew Wilcox describes responsiveness as trying to "*present information as clearly as possible*" and "*be as efficient as possible for the user*" [75]. In summary, the meaning of responsiveness seems to evolve towards an overall concept for a good user experience. This work tries to apply a reasonable intersection of the current understanding of various authors.

2.5 Images on Websites

Before we cover responsive images in the next section it is important to learn more about images on the web in general. In web design and also other design disciplines, images have either a contentual or a presentational purpose. Contentual images are of informative value to the user and contribute to the document's content by carrying this information. For example, a picture or a figure which further describe the content. Presentational images only serve a decorative purpose and do not contain valuable information for the document's context. Typical examples are small icons for buttons or background images, eg. a colored pattern that stretches across the entire webpage. [76]

Having covered these two types of images, we can introduce the two main methods to display images on websites: the HTML `<img>` tag and CSS background images. Earlier we learned that HTML is used to semantically outline the structure and content of a document. Thus, the `<img>` tag is intended for contentual images and will be covered first. The following listing shows a typical `<img>` tag.

```
<img src="family.jpg" width="600" height="350" alt="A father playing with children at home.">
```

Listing 7: *An example of the HTML `<img>` tag.*

The src attribute provides the image source and is the only required attribute. It is usually a relative or absolute URL to an image file. Using the data URI scheme it is also possible to provide a Base64 encoded image directly within the src attribute [74, p. 129]. The result is that no extra HTTP request is needed to fetch the image file, as it is already embedded in the HTML document itself. This feature is rarely used but can be a sensible alternative to improve the performance, especially for small images. The data URI scheme can be used for JS-based responsive images solutions to deliver valid HTML documents. An issue for responsive images is that the `<img>` tag only supports only src attribute. The width and height attributes contain the image's intrinsic size in CSS pixel. This can help the browser to render the page layout before the image has been loaded completely. It should be considered that there are users which might not be able to see images, for example blind people. The alt attribute is for them. It stands for alternative text and should provide a textual description of the image. Beside the attributes used in the above example, the sizes and srcset attributes are relevant to responsive images. These two attributes were recently added to the specs as part of the native responsive images solution, which is covered in the next section.

The second main method for implementing images on websites are CSS background images. This approach is intended for presentational or decorative images. As the name suggests, images implemented this way are displayed in the background of the actual content. There are several CSS properties to adjust the appearance of background images. The properties can be applied to any HTML element. The image resource is set by the background-image property. As with `<img>`'s src tag, the value can be an URL or an inline image encoded as Base64. The background-repeat property determines if the background image is repeatedly displayed if it fits into the target element more than once. More properties are covered in the evaluation of the CSS Background Images solution in Section 3.4.5. [76]

In Section 2.2.5 we learned that CSS handles HiDPI screens by defining a DPR to map one CSS pixels to a different number of device pixels. This method affects the presentation of images. The browser renders an image not necessarily with the image's intrinsic width, but calculates the actual width in device pixel using <intrinsic image width> × DPR. For example, an image width an intrinsic width of 200 pixel would be scaled and displayed with 400 device pixels on a screen with a DPR of 2. This behavior is mostly desired if the provided content is *not* optimized for DPRs other than 1. Without this scaling process the image would be displayed much smaller and perhaps be illegible. However, if we *do* supply optimized images, they should always be displayed with a DPR of 1. This ensures that we get the desired result of sharp images, where 1 image pixel maps to 1 device pixel. In order to consider this issue, the client needs to know the DPR of the images returned by the server. Responsive images approaches should offer a method to provide the DPR with the delivered content. If this is not possible, the issue can be solved by explicitly assigning width and height values using CSS.

Image files are stored and organized using an image format. They usually apply some kind of compression to decrease the amount of storage needed to save the file. Compression algorithms can be classified as lossless or lossy. While a lossless algorithm operates without decreasing the quality of the source material, lossy compression results in even smaller files but at the expense of quality. Most image formats can also be categorized as raster or vector formats. [77]

Raster formats store information for every pixel in the graphic and thus, the file size increases with the size of the image. The most common raster image formats for the web are the *Joint Photographic Experts Group* (JPEG or JPG) format, the *Portable Network Graphics* (PNG) format and the lesser-used *Graphics Interchange Format* (GIF). All of them are supported by all major browsers. The JPG format is used for 45% of all images on the web as of July 2015 [4]. It is the main format for lossy photography compression. It does that very efficiently and as the proof of concept in Section 4 uses photographs, it is the main image format used in this work. However, all presented solutions also work with PNG and GIF images. PNG supports a separate (fourth) color channel for storing transparency, also known as alpha transparency. It is a frequently used feature on websites. The PNG format can very efficiently store images with big uniformly colored areas. These features made PNG a very popular web image format as well, being used for 30% of all images [4]. The GIF format is popular for its ability to store animations. It is used for 23% of all images on the web [4]. Due its limitation to only 256 different colors, animations should be the only use case to deploy the GIF format. There are some other promising image formats up and coming, but none of them has substantially gained popularity so far. For example, Google's WebP format is reported to be more efficient as JPG but lacks browser support [78, p. 238]. [77]

Vector formats use mathematical expressions to only store information about individual elements of a graphic, eg. lines or circles. Vector images can be scaled without losing quality. In theory, this feature makes them an ideal format for responsive images. However, due to the complexity of photographs they can not be efficiently represented by mathematical expressions. Thus, vector formats are only suitable for simple graphics like icons or logos in practice. The W3C invented the *Scalable Vector Graphics* (SVG) format, which is the only widely supported raster format for the web. As this work's focus is put on the use of photographs, SVG is not further discussed in the following sections. For content other than photographs, SVG and other vector formats should definitely be considered as an alternative to raster formats though. [19, p. 126], [74, p. 140]

3 Responsive Images

This section evaluates six responsive images solutions. It starts with Section 3.1, which covers the basics and the motivation. Section 3.2 outlines the conceptual method for making images responsive and outlines the differences of server- and client-sided approaches. The evaluation was performed according to an Evaluation Framework, which is documented in Section 3.3. Following next, Section 3.4 covers the evaluation itself and describes the conceptual implementation of each solution. An overview of the findings can be found in Section 3.5. This last section also interprets the results, gives recommendations, suggests alternatives and outlines limitations of the research approach.

3.1 Introduction

In Section 2 we learned a lot about the history and motivation for the responsive movement. It started as an attempt to encounter the problems that occurred with the boom of smartphones, which started in 2007. Eventually Marcotte came up with RWD [63], which happened to be adopted by developers and has been the core part of responsive techniques since. RWD addressed images by a technique called fluid images. It consists of basically one line of CSS which causes images to be flexible:

```
img { max-width: 100%; }
```

Listing 8: *CSS declaration of the fluid images technique.*

This CSS code causes images not to be displayed with their intrinsic width and height, but to adapt to the width of their parent element. That way, the client automatically scales images to fit the parent element while preserving their aspect ratio. Images adapt to the flow and size of other elements, which gives this technique its name fluid images. [1, p. 48]

However, this adaption is just one aspect of truly responsive images. Let us remember [75]'s definition of responsiveness: "*present information as clearly as possible*" and "*be as efficient as possible for the user*". Now consider these two practical questions:

1. Is information presented as clearly as possible by using the same image with the same field of view for all screen sizes?
2. Is it as efficient as possible to serve the same image file to a 1920×1080 pixels desktop monitor as to a 480×320 pixels smartphone?

The answer to both questions is *no* for most use cases. Firstly, answer one is explained. As we learned before, by using the fluid images technique images are automatically scaled. It is important to understand that the actual image file does not change. That means the image content does not change either, it is just scaled. The issue here is most evident for big images with a small area of interest. Such an image does look fine on a big screen. On a small screen though, due to scaling the already small area of interest gets even smaller and perhaps illegible. As we will learn the next section, the use case of serving different image content depending on variables like screen size is referred to as *art direction.* [79]

The second answer can easily be explained using some example data. Let us stick to the resolutions from the question and assume that the image is display with the width of the viewport, thus 100% of the `<html>` element. A typical 1920×1080 pixels photograph is around 2 megabytes in size. This same image is delivered to the smartphone, where it gets down-scaled to 480 pixels in width by the client. If we would deliver a down-scaled image from the beginning, we could easily save 1.8 megabytes – a huge saving of 89%. This issue is referred to as *resolution-based image selection.* [5], [79]

Motivation

Now that we have a basic understanding of what the responsive images idea is about, we can further outline the motivations for a solution. As of July 2015, the total size of an average website is 2.2 megabytes [4]. This is a lot of data – even with a very fast Internet connection it usually takes at least one or two seconds to download. When using a mobile device while having a poor network coverage it can easily take tens of seconds or even longer to load. Speed and performance are one of the key factors of a good user experience. Research shows that users are not willing to wait for longer than 10 seconds and might immediately leave the website [80]. Developers have addressed a lot of issues to decrease the total weight of a webpage. They minify CSS and JS files, enable HTTP compression, optimize caching and use content delivery networks (CDNs) to spread their files on server all over the world to reduce loading times [74, p. 109]. All of these methods have great benefits and should be used, but another major aspect is often missed: images. Sheer 63% of the total size of the average website are images. In absolute numbers that is 1.3 megabytes of images for the average website [4]. For years, these numbers have been growing and that trend will probably go on due to continuously increasing screen resolutions. An important factor for this trend are HiDPI screens. A display with a DPR of 2 or 3 results in 4 respectively 9 times as many pixels as for a DPR=1 display.

The general issue of image size could be approached by inventing new image formats with better compression algorithms. However, this is a very complex topic and it takes years to establish a new image format on the web. Furthermore, it is just one part of the problem.

The main issue lies in the great fragmentation of screen resolutions, especially since smartphones got so popular. Even with better compression, it is far from ideal to serve the same image files to every device. The `<img>` tag was originally limited to just one resource for the src attribute though. Designers were faced with the choice of "*whether to make their pages fuzzy for some or slow for all – most designers choose the latter, sending images meant to fill the largest, highest-resolution screens to everybody.*" [81]. A lot of data – and thus, time – is wasted by serving large images to small screens. The community came up with many approaches to tackle this problem. Some of these approaches are reviewed in Section 3.4. All of them have some drawbacks which is why a native solution supported by the browser was demanded. The *Responsive Images Community Group* (RICG) was formed in 2012 by the W3C to address this issue. The development of a proposal was accompanied by many discussions, suggestions and disagreements. Eventually the RICG and other stakeholders agreed on a solution and it was added to the WHATWG specs in August 2014. The W3C specs followed in March 2015. That did not immediately solve the problem though. As we learned in the Section 2.1, something being added to the specs does not necessarily mean browser vendors implement it.

Opera and Chrome were one of the first browsers to support native responsive images in October 2014. This was only made possible by a crowd funding project initiated by RICG member Yoav Weiss [82]. Many people and companies contributed, what clearly showed the actual need for a solution. The situation is getting better and the native responsive images solution should be supported by all major browsers in foreseeable future. Mozilla only recently shipped version 38 of its Firefox browser with support in May 2015. Apple has partly implemented the specs for the Safari browser. Microsoft has implemented the srcset attribute to its preview version of Internet Explorer and the `<picture>` element development status is "*under consideration*" [83]. This sounds promising, but does not solve the issue for older browsers which are still in use. Until there is widespread support and old browser versions vanished, developers have to consider all available options and choose the most-appropriate solution for every project. The following tries to provide a guideline and decision support by reviewing and comparing current approaches. [84]

3.2 Technical Foundations

Before we set up an evaluation framework, it is helpful to know how possible solutions work from a technical perspective. Available solutions can be categorized as server- or client-sided, based on whether the image URL or resource is selected and set on the server- or client-side. Both categories have similar characteristics and their pros and cons, which will be discussed later in this section. Before doing so, let us first cover the concept of possible solutions in general. Most of the responsive images use cases require to choose an image depending on characteristics of the user's device, eg. the screen resolution. The basic workflow of a possible solution is as follows:

1. Retrieve the current value of a device characteristic of interest.
2. Use some logic on the value to determine which image fits best.
3. Instruct the user agent to download only this image and display it.

These steps should be understood as a rather conceptual description of a solution. Not all solutions require the developer to implement every step. Some steps might be handled by the solution itself or multiple steps can be processed together.

Step 1 deserves a closer look as the quality of a solution is greatly dependent on the available device information. First of all, device characteristics are inherently only known by the device itself. The browser running on the device has access to the operating system and thus, to a lot of information like screen resolution, connection speed or available input methods. Some of this information is made available to the developer through JS and CSS media queries. As we learned earlier, it can not be relied on JS being available. CSS media queries only work with CSS background images (or JS), which are not suitable for all use cases. Assuming we can neither use JS nor CSS media queries, we are left with server-side technologies. However, on the server-side it is far more difficult to obtain information about the device. Server and client communicate via the HTTP protocol. Per default the only useful transmitted information by HTTP requests is the User-Agent header. This string can be used to perform a user agent detection, which is used by Solution 1. A more detailed discussion about the differences of server- and client-sided solutions is presented in the following.

3.2.1 Server-Side Solutions

In general, all server-side measures to enhance the responsiveness of a website are summarized as RESS. The classification as a server-side responsive images solution in this work means that the image URL or resource is determined and set on the server. This is done by a server-sided scripting language like PHP, Python and others. The script determines the best image and then directly inserts the image URL into the HTML or CSS code before it is sent to the user. In consequence, the client knows the correct image source immediately after fetching the HTML or CSS code. This allows server-side approaches to be very fast, as the browser's lookahead parser can initiate the image download at a very early stage. [37]

An alternative to setting the image URL directly in the HTML or CSS code is to leave the URL the same, but serve different images. This can be done by configuring the web server to forward all image requests to a custom script. This script selects and delivers the best available image or creates a fitting version exclusively for the particular request. An advantage of that option is the easy integration into existing applications. The HTML markup for images can remain unchanged and the best image is delivered for the existing image URLs. A problem with serving multiple image versions for the same URL is caching. Proxy servers have to be informed about when to use which image version. This is not always possible and can prevent responses from being cacheable. Details will be covered later in the evaluation.

A common issue of server-side approaches is the gathering of device information. The used information gathering technique is the main difference between the first three evaluated solutions. Solution 1 deploys user agent detection as a very simple method to learn more about the user's device. Solution 2 utilizes JS to retrieve device information and saves this information to a cookie. The cookie data is sent to the web server with every HTTP request and can be used by a script to select an image. Solution 3 is based on a rather new approach called HTTP Client Hints. It extends the HTTP protocol by various headers which the client can set to transmit device information to the server.

Another drawback of server-side approaches is their inability to dynamically react to changes of the user environment. Possible changes include resizing the viewport, changing the device's orientation or using the zoom function, among others. As the image source is set on the server-side, changes can only be made on a new page load or a refresh. This prevents images from being changed immediately after changes of the user environment. The issue can partially be overcome by using JS to react to environment changes. Solution 3 supports dynamic changes, but it is up to the browser when to reissue image requests on changes.

3.2.2 Client-Side Solutions

The classification as a client-side solution in this work means that the image URL or resource is determined and set on the client. This can either be done by the developer using CSS or JS, or natively by the browser itself. The covered client-sided solutions are far more different among each other than the server-sided solutions. Therefore, not as many common characteristics can be found.

In general, the device information gathering process is easier because the code runs on the client device itself. All presented client-sided solutions are quite flexible. They allow for dynamic changes without a reload of the whole page. Especially if JS is available, very customized and powerful implementations are possible. A downside of JS approaches is that they can slow down the page loading time. JS code per default only gets executed when the HTML has been fully loaded and the DOM has been created. This prevents the lookahead parser from pre-loading images. [37]

Client-sided solutions commonly use different URLs for different images and thus, allow the use of CDNs and images to be cached by proxy servers.

3.3 Evaluation Framework

This section outlines the used approach of the following evaluation. It documents the creation of an evaluation framework. The framework's purpose is to perform the evaluation in a formal, understandable, and comprehensible manner.

3.3.1 Selection of Solutions

One of the first steps was to select solutions to be evaluated. There are various responsive images approaches available at the time of writing. Many people created customized solutions to suit their specific requirements. Most of these approaches have an overlying conceptual approach in common. It is not within the scope of this work to evaluate every available solution. This was encountered by selecting a subset of concepts for evaluation. First, it was tried to collect a pool of available concepts. Following next, some of these concepts were selected for evaluation. The selection process was done by considering the following requirements:

1. The solution should be commonly used at the moment.
2. Browser support should be good, or expected to be within the near future.

The first requirement was assessed by reviewing opinions of the developer community. Many posts, comments and articles were considered to come to a decision [85]–[88]. The second requirement could mostly also be evaluated together with the research for requirement one. Browser support of at the time unsupported solutions or features was verified by checking browser vendors' websites and bug tracker platforms [89]–[91]. This approach led to a selection of six solutions, which are presented in the next section. The HTTP Client Hints solution is not commonly used at the moment due to a lack of browser support. It was selected anyway as support can be expected within the near future. Not selected solutions include the CSS image-set method [92] and the CSS content property approach [93]. They are both not commonly used. The image-set method only allows a DPR-based selection at the moment [94, Sec. 2.3]. The content property approach has several drawbacks. Both approaches are further discussed as alternatives in Section 3.5.2.

3.3.2 Evaluation Criteria

A formal process was needed in order to conduct a comprehensible evaluation and to maximize objectivity. This was encountered by creating an evaluation framework to follow. This resulting framework consists of various criteria which are rated.

The evaluation was conducted by only one person, the author of this work. Therefore, a personal bias was inevitably present. To minimize this influence, it was decided to use a qualitative evaluation model rather than a quantitative model. This means rating the criteria by choosing from a pool of predefined alternatives instead of assigning a numerical value within a certain range. Furthermore, for quantitative approaches it is often sensible to assigns weights to the criteria. These weighting values are usually very different and dependent on the requirements of the particular business scenario. Thus, the qualitative approach used in this work does not aim to find the best of all evaluated solutions. Instead, the intention is to compare different solutions and provide data to support the decision making process.

The evaluation criteria were established based on common use cases and requirements of responsive images. It was reasonable to categorize all criteria as either functional requirements, non-functional requirements, or system prerequisites. Even though system prerequisites are part of the non-functional requirements, they were chosen to represent a separate category. The reason was to provide a more detailed distinction and will be explained later in this section.

Functional Requirements: Use Cases

The functional requirements category consists of seven use cases. All use cases are based on a document by the RICG [79]. The RICG was tasked to develop a standardized solution for responsive images. They created a list of use cases and requirements as a foundation for their work. As their goal was to create a future-proof standard, their list is very comprehensive. Not every use case is of equal relevance to the developer community. Most available responsive images solutions try to solve use cases one and two, which can consequently be considered the most demanded use cases. These two use cases – resolution-based selection and art direction – have already been introduced in the example at the beginning of Section 3.
Based on document of the RICG, the following describes the functional requirements for the evaluation in form of use cases. The possible rating options were chosen to be *no*, *poor*, *medium*, *wide*, and *full* support.

1. **Resolution-based selection:** This use case refers to choosing an image file depending on the screen size of the client. Web developers want to provide images at multiple resolutions so that every device can request and display the most appropriate size. This avoids wasting bandwidth and time on loading image files which are (much) bigger than the client's screen.
2. **Viewport-based selection:** Viewport-based selection takes into account that images are often not displayed with the full width of the viewport. A characteristic of responsive designs is that the width of images can change depending on the active CSS media query breakpoint. Therefore, it should be possible to select images not only based on the screen or viewport size, but also take the current breakpoint and its specified image width into account. [79]
3. **DPR-based selection:** Many smartphones and increasingly other devices as well are shipped with HiDPI displays. Those are screens with a relatively high pixel density of usually more than 240 PPI. HiDPI displays have changed the way websites and images are rendered. Details are covered in Sections 2.2.5 and 2.5. The consequence is that HiDPI displays should be served with higher resolution images as traditional (DPR=1) screens. Otherwise, the images are upscaled by the DPR and do not take advantage of the high resolution of HiDPI screens. Thus, a solution should allow to consider the DPR when selecting an image. The solution should also provide a way to inform the client about the DPR for which the returned images are optimized. This is needed to render the images with the correct size if no explicit display size was assigned to the image using CSS. [39], [41], [79]
4. **Art Direction**: Responsive designs usually target many different devices with a broad range of screen sizes. By using the fluid image technique, images automatically adapt to fit the layout. The scaling process inevitably causes information to be lost. At worst, small objects in the image might not be identifiable anymore after the scaling process. Especially if an important object of the image is affected, the image could lose its informative purpose and become useless. Therefore, in addition to selecting images by

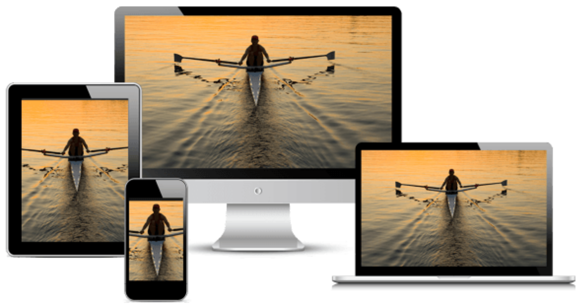

Figure 2: *The art direction use case: A different (cropped) image is used for smaller screens. [148]*

resolution, viewport and DPR, developers also want to serve *content-wise* different images for each of the other use cases. This could mean a cropped version of the main image or a different image altogether. This process of author-based selection is refereed to as art direction. [79]

5. **Match media types and features**: This use case describes more refined image selection based on device capabilities. For example, printers commonly work at resolutions of 600dpi or more – a lot higher than usual screens. In order to print images sharp and clear, the browser should be able to request images with a higher resolution for printing. Printers can be targeted using the media type print. Other devices might require an even more detailed selection using media features. Many e-book readers are shipped with monochrome displays. After the conversion from color to monochrome, many colors are not distinguishable anymore on these screens. This can be an issue when colors hold important information, eg. in diagrams. Authors want to provide an alternative image that is optimized for these devices. This can ideally be solved using CSS media types and features. Additionally, the upcoming CSS4 specs will implement new media features which could open up new possibilities for this use case. [79]
6. **Image format-based selection**: At the moment the commonly used and supported image formats in the web are JPG, PNG and GIF. However, new formats with more features and better compression algorithms are emerging. Some formats are very effective for particular image content, eg. SVG for simple vector graphics. Developers want to choose the most suitable image format for every image. However, currently there is no reliable method to check if a format is supported by the client's browser. This leaves developers with using either JPG, PNG or GIF. A solution should provide support to explicitly specify the image format of a resource and let the browser chose one. [79]
7. **User agent-based decisions**: The are certain selection conditions that the developer can not detect. For example the user's Internet connection: How fast is it? Is it stable or a mobile network with a weak coverage? Is it an expensive roaming connection? Other possible conditions would be the processor speed or current battery level of the device. If the computing power is very limited or the battery is almost empty, it might sensible to deliver smaller images. To encounter this use case, the browser of the user should have the possibility to select the best image based on the current conditions and user-defined settings. [79]

Non-Functional Requirements

A solution should support all six use cases, which comprise the functional requirements. Beside these, there are some more general, non-functional requirements [75], [79]. The possible rating options were chosen to be *low*, *medium,* and *high.*

1. **Browser Support:** A solution should be supported by all major browsers. If it is not supported by default, it should be possible to implement a workaround for the given browser, eg. using a polyfill [74, p. 79]. Furthermore, a solution should be backwards compatible. It must not break existing websites or code and should be compatible to older browsers. If it is not compatible with older browsers by default, it should be easy to implement a workaround. [75], [79]
2. **Performance:** A solution should be fast. Responsive image sources should ideally be present already in the HTML document so that the browser's lookahead parser can initiate preloading of images. Images should be responsive already at the first pageload. This means no time-consuming actions like extra HTTP requests should be needed before the solution works. Furthermore, the HTTP responses generated by the solution should be cacheable by proxy servers and CDNs, even when using the same URL for different versions of a resource. [79]
3. **Reliability:** A solution should produce reliable and accurate results on all devices if the browser is supported and the system prerequisites are fulfilled. This requirement is greatly dependent on the quality of the available device information. [75]
4. **Integrability:** A solution should be easily integrable into an existing system or application. This is especially relevant for legacy websites which should be made responsive without changing the code or markup.
5. **Flexibility:** A solution should not only statically fulfill the use cases. Instead, the webpage should also be updated on dynamic changes of the user environment, eg. the viewport size or the screen orientation. The page needs to be updated immediately without requiring a page reload. For example, when a user turns the device from landscape to portrait mode, another image resource might be needed – a common art direction use case.
6. **Accessibility:** A solution should allow content to be presented in an accessible way. This includes compatibility with HTML's accessibility features, especially providing an alternative text for images using the alt attribute. A solution should also allow the developer to write semantically correct and valid HTML code. Images should be outlined as such, ideally using the `<img>` tag. Furthermore, the solution should not use meaningless

placeholder images which are later replaced by the real image using JS. In other words, a browser which does only load the HTML document and with JS disabled should be able to render the page correctly including the images. [79]

System Prerequisites

The system prerequisites requirement was chosen to be split into three individual criteria. The presented solutions either require cookies, JS, or a server-sided scripting language; or a combination of such. A solution does ideally not have any of the above system prerequisites. Cookies and JS are user-defined settings and are not within the developer's reach of influence. In contrast, server-sided scripting languages run on the server of the website and can usually be installed or changed by the system administrator. The availability of these system prerequisites may vary depending on the target audience and the used hosting platform. A single systems prerequisites criterion would not allow a thorough comparison of the evaluated solutions. Therefore a separated inspection of cookies, JS, and server-sided logic was found to be a reasonable approach. The possible rating options were chosen to be *needed* and *not needed*.

1. **Cookies:** Does the solution need HTTP cookies on the client?
2. **JavaScript:** Does the solution need client-sided JS to work?
3. **Server-side logic:** Does the solution need server-side logic to work? This mainly includes the requirement of server-sided scripting languages like PHP, Python, or Ruby; but also any server-sided configuration, eg. for the web server.

3.3.3 Resulting Evaluation Framework

The below tables show the resulting evaluation framework. It consists of seven functional requirements with each five rating options, six non-functional requirements with each three rating options, and three system prerequisites with each two rating options.

Functional Requirements	Rating Options
1. Resolution-based selection 2. Viewport-based selection 3. DPR-based selection 4. Art Direction 5. Match media types and features 6. Image format-based selection 7. User agent-based decisions	1. No support 2. Poor support 3. Medium support 4. Wide support 5. Full support

Table 1: *Overview of functional requirements*

Non-Functional Requirements	Rating Options
1. Browser support 2. Performance 3. Reliability 4. Integrability 5. Flexibility 6. Accessibility	1. Low 2. Medium 3. High

Table 2: *Overview of the non-functional requirements*

System Prerequisites	Rating Options
1. Cookies 2. JavaScript 3. Server-side logic	1. Needed (Yes) 2. Not needed (No)

Table 3: *Overview of the system prerequisites*

3.4 Evaluation of Solutions

As we learned in Section 3.1, a native solution for responsive images is already available. Once it is widely supported by all major browsers and older browsers are less relevant, it would be the best solution for most use cases. Until that, other approaches have to be considered as well. This section evaluates the rather new native solution of HTML5 as well as five alternative methods according to the framework defined in Section 3.3. This is done alongside with explaining the implementation principle using sample code. It is out of the scope of this work to describe the implementation in great detail, but the concept is outlined for common use cases.

3.4.1 Overview

The following table shows the solutions that are evaluated in this work. They can be categorized as server- or client-sided, based on whether the image URL or resource is selected and set on the server- or client-side. Both categories have similar characteristics and their pros and cons, which are discussed in Section 3.2.

Server-Side Solutions	**Client-Side Solutions**
1. User Agent Detection	4. CSS Background Images
2. Cookies	5. JavaScript
3. HTTP Client Hints	6. Native HTML5

Table 4: *Overview of the selected solutions.*

First of all, we look at approaches without using the new native `<img>` srcset attribute and `<picture>` element of Solution 6. As we saw earlier, the `<img>` tag has supported only one src attribute before responsive images were added to the specs. This only value of the src attribute is set in the HTML code of the document. There are two common methods to serve different src values to the client. We can change the attribute on the client using JS or we can change the attribute dynamically on the server before sending out the HTML code. The latter can be done via server-side scripting languages like PHP or Python and is discussed first.

3.4.2 Solution 1: User Agent Detection

As we have seen before, Solutions 1, 2, and 3 mainly differ by the used method to gather device information. The first solution utilizes an approach called user agent detection. It is a method to draw conclusions about the user's device and browser by examining the HTTP User-Agent header. The approach was invented to encounter the issue that the server does by default not know the client's capabilities.

Screen resolution, connection speed, available input methods – none of these information are sent via a HTTP request. HTTP Client Hints are a HTTP extension to change that and are used by Solution 3. JS could help out here but let us assume JS is not available. In this case the only useful available information is the HTTP User-Agent header. It is an optional header that can be added to HTTP requests by the browser. The header contains information about the client. Usually it includes the browser name and version, and the device's operating system. Some mobile browsers also include the model name of the device. The following table shows some typical user agent strings [95]:

Platform	**User Agent String**
Apple iPhone with iOS 5.1	`Mozilla/5.0 (iPhone; CPU iPhone OS 5_1 like Mac OS X) AppleWebKit/534.46 (KHTML, like Gecko) Version/5.1 Mobile/9B179 Safari/7534.48.3`
Windows 7 64bit with Chrome	`Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.81 Safari/537.36`
Windows 7 with Internet Explorer 11	`Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko`
Motorola Moto G	`Mozilla/5.0 (Linux; U; Android 4.3; xx-xx; XT1032 Build/14.10.0Q3.X-28) AppleWebKit/534.30 (KHTML, like Gecko) Version/4.0 Mobile Safari/534.30`

Table 5: *Examples of HTTP User-Agent headers*

It can be seen that the contained information greatly varies between browsers and platforms. A very simple user agent detection approach would be to only detect the device category. Mobile devices mostly include the string "mobile" in their user agent string. We can use this fact to determine if we are dealing with a mobile device. This can be helpful, but is not enough due to the great fragmentation of mobile devices. More sophisticated approaches use the user agent string to query a so called device database. These databases contain user agent strings for many (mobile) devices and their characteristics. Ideally, the query returns the exact device model and all its capabilities. A popular database is called WURFL, maintained by ScientaMobile [96].

User agent detection is the subject of many controversial discussions. It can be helpful for the developer but has some drawbacks and is not reliable for various reasons. There is no specification what a user agent string has to include. It is remarkable that all of the above user agent strings begin with "Mozilla/5.0", even when not running a browser from Mozilla. This is one of many remains from the browser wars era. Back then, the Mozilla browser supported more features than other browsers. Some web servers were configured to send an enhanced version of the website to the modern Mozilla browser, based on the user agent string. Eventually other browses caught up and implemented the new features as well. Their vendors also wanted the enhanced websites for their users and started spoofing the user agent string by adding “Mozilla”. This and some similar events led to the user agent string becoming *“[...] a complete mess, and near useless, and everyone pretended to be everyone else, and confusion abounded”* [97]. User agent spoofing is also implemented as a “feature” in some mobile browsers. These browsers offer a setting like “Show full website” to display the desktop version rather than the mobile version of a website. This is partly done by spoofing the user agent string to match a desktop system. [74, p. 48]

Additionally, many user agent strings do not contain any information that can be used to determine the screen resolution. Only some mobile devices include an exact model tag that can be queried against a device database. The databases can be outdated and might not be free of charge. As a result of these issues, user agent detection often leaves developers with only knowing the device category, if at all. This is not sufficient because of the great fragmentation of devices and resolutions. [74, p. 48]

That said, server-side user agent detection can sometimes be a helpful addition for being responsive but should never be the sole solution. It can be used as a fallback when other solutions fail, eg. when JS is not available while using Solution 2. [98]

Implementation

As the last paragraphs suggested, parsing a user agent string is a hard task. It is recommended to utilize ready-to-use implementations. The following shows two sample implementations. Earlier we learned that server-sided approaches are capable of serving different images for the same URL. To do so, we need to configure the web server to not directly serve image requests but forward them to a predefined script. For the popular Apache web server, the mod_rewrite module can be configured to do so with the following code. [99]

```
<IfModule mod_rewrite.c>
  RewriteEngine On                                  # activate the RewriteEngine
  RewriteRule \.(?:jpe?g|gif|png)$ image_handler.php # rewrite rule, based on file extensions
</IfModule>
```

Listing 9: *Apache configuration to forward image requests to a custom script.*

The above listing configures the Apache web server to forward all requests ending with .jp(e)g, .gif, or .png to a script called image_handler.php instead of serving the requested file directly. Once the image request forwarding is set up, we can start with the implementation of the image handler script. First, we look at a very simple mobile device category detection to demonstrate the approach. It is inspired by Matthew Wilcox's Adaptive Images approach, which is also used as a sample implementation for Solution 2 [99].

```
<?php
  // check if user agent string contains “mobile”
  if (stripos($_SERVER['USER_AGENT_STRING'], 'mobile') !== false) {
    $imageDirectory = '/images/mobile/'; // mobile device
  }
  else {
    $imageDirectory = '/images/desktop/'; // no mobile device, assume desktop system
  }

  // get the requested filename and its path
  $requestedFile = basename(parse_url(urldecode($_SERVER['REQUEST_URI']), PHP_URL_PATH));
  $filePath = $imageDirectory . $requestedFile;

  // set the Content-Type HTTP header to image/png, image/gif or image/jpeg
  $extension = strtolower(pathinfo($filePath, PATHINFO_EXTENSION));
  if (in_array($extension, array('png', 'gif', 'jpg, 'jpeg'))) {
    header("Content-Type: image/" . ($extension == 'jpg' ? 'jpeg' : $extension));
  }

  // set HTTP headers to prevent caching by proxy servers, but allow browser caching
  $browserCacheTime = 86400 * 7; // one week
  header("Cache-Control: private, max-age=" . $browserCacheTime);
  header('Expires: ' . gmdate('D, d M Y H:i:s', time() + $browserCacheTime) . ' GMT');
  header('Content-Length: ' . filesize($filePath));

  // send out image
  readfile($filePath);
  exit();
?>
```

Listing 10: *Using user agent detection to serve different images based on the device category.*

More sophisticated implementations can detect more than just the device category. They also not only consider the user agent string, but other headers which can be attributed to certain browsers or platforms. One of these implementations is the PHP library *Mobile Detect*, which supports a wide range of detections [100]. The following listing demonstrates its use to target different devices and browers.

```
<?php
  require_once 'Mobile_Detect.php';
  $MobileDetect = new Mobile_Detect;

  if ($MobileDetect->isMobile())    { /* code for mobile devices        */ }
  if ($MobileDetect->isTablet())    { /* code for tablet devices        */ }
  if ($MobileDetect->isAndroidOS()) { /* code for Android devices       */ }
  if ($MobileDetect->is('Chrome'))  { /* code for Google Chrome browser */ }

  /* image handling code should be inserted here */
?>
```

Listing 11: *Using the Mobile Detect PHP library for device and browser detection.*

Evaluation

The evaluation was performed according to the evaluation framework that was defined in Section 3.3. The comment column explains the assigned rating.

Functional Req.	**Support**	**Comment**
Resolution-based selection	Poor	Mobile devices which send a model tag can be handled decently but for most desktop systems there is no information about the screen size and thus, no delivery of responsive images possible.
Viewport-based selection	No	The size of the viewport and especially the display width of the images can not be known on the server-side by only using user agent detection.
DPR-based selection	Poor	Mobile devices which send a model tag can be handled decently but for most desktop systems there is no information about the screen size and thus, no selection based on the DPR is possible. No method to inform the device about the content's DPR. Thus, the display size of the image has to be specified within HTML or CSS.
Art Direction	Poor	Only very simple art direction based on the screen resolution, if available. No support for CSS media queries and no dynamic changes without a reload of the page.
Match media types and features	No	–
Image format-based selection	Medium	The browser version in the user agent string can be used to determine if it supports a specific image format. Additionally, the HTTP Accept header sent by the client could provide information about supported image formats.
User agent-based decisions	No	There is no standardized way to instructing the server to deliver a certain version (eg. small, to save bandwidth) of images.

Table 6: *Evaluation results of the functional requirements of Solution 1.*

Non-Func. Req.	Rating	Comment
Browser support	High	User agent detection only makes use of the user agent string, which is by default sent by every major browser. Browser support can be considered very high, not taking into account browsers which send a user agent string without containing any valuable information (see Reliability).
Performance	Medium	As for all server-sided approaches, it is possible to (a) use the same URL for different image versions in a HTML or CSS document or (b) directly insert the URL of the most-appropriate image into the HTML or CSS on the server-side. Both results in the image URLs being available directly within the HTML code. Thus, the browser's lookahead parser can initiate image requests very early. Responses can not be cached by proxies or CDNs.
Reliability	Low	The impact of this result can not be stressed too much. The reliability of user agent detection for responsive images is very low. Mobile devices which send a model tag can be handled decently but for most desktop systems there is no information about the screen size and thus, no delivery of responsive images possible.
Integrability	High	User agent detection can easily be integrated into an existing application without changing the markup. The server can handle the image selection and the generation of different image versions.
Flexibility	Low	Reacting to dynamic changes of the user environment by re-issuing image requests is not possible. JS would be needed to add that functionality.
Accessibility	High	The `<img>` tag and its alt attribute to provide an alternative text can be used to correctly outline the content as an image. The image source can be placed into the HTML document directly on the server, resulting in semantically correct HTML code.

Table 7: *Evaluation results of the non-functional requirements of Solution 1.*

System Prereq.	Needed	Comment
JavaScript	No	JS is not needed for server-sided user agent detection, but it is common practice to extend the implementation using JS.
Server-side Logic	Yes	Server-sided logic is needed to handle the entire process of making images responsive. Thus, a server-sided scripting language and a configurable web server is required.
Cookies	No	–

Table 8: *Evaluation results of the system prerequisites of Solution 1.*

3.4.3 Solution 2: Cookies

It was mentioned earlier that Solution 1, two and three mainly differ in their method of getting device information. Instead of using the user agent string, this solution utilizes client-sided JS to retrieve device capabilities. The device information have to be transferred from the client to the server for the selection of the best image. An obvious approach might be to send a simple HTTP GET request to the server and include the device information as a parameter. If we think of our requirements, that is not a good idea though. We want to deliver responsive images as fast as possible, ideally already for the first page load. By the time the JS code has been executed and the GET request has been sent, the browser will most likely have loaded (possibly not best-fitting) images already. This solution solves the issue by saving the device information to a cookie using JS. The cookie with the device information is automatically included to every request to the server. The image handler script on the server can use the information to select and deliver the best image. The JS code to set the cookie is inserted very early in the HTML document to ensure its execution before the browser initiates any image requests. [99]

A downside of cookie-based solutions is intermediary caching. When different images are served for the same URL, proxy servers have to be informed how the images vary and when to deliver which image. In theory, this can be done using the HTTP Vary header. This header contains the name of another HTTP header which was considered to select a resource. In our case, the correct Vary header would be `Vary: Cookie`. However, most websites create more cookies than just our device information cookie. The result are practically endless combinations of Cookie header values. As a result, proxy severs can not make practicable use of a Vary headers set to Cookie. The same issue applies to the previous solution, as there are thousands of different used user agent strings.

Another issue of this solution is the limitation to one domain. This is due to the same-origin policy concept of web applications, which also applies to cookies. It states that a cookie for a certain domain can only be set by JS code or HTTP responses of the same domain. Furthermore, a cookie set for a certain domain will *only* be included to requests to the same domain. For the current solution, the JS code to set the cookie with the device information has to be inserted directly within the HTML code of the website. Thus, the cookie will only be included to requests to the same domain. This requires that all images which are meant to be responsive have to be delivered by a server that is on the same domain. This issue could prevent the solution from being used with 3rd party CDNs that run on a different domain. [101]

Implementation

A commonly used implementation of a cookie-based responsive images solution is Adaptive Images by Matthew Wilcox [99]. It acts as a foundation for the following explanations. As for Solution 1, it is required to configure the web server to forward image requests to a custom script. This script comprises the main part of Adaptive Images and is written in PHP. Before this script can work as intended, the cookie with device information has to be set. JS can be used to do so using the `document.cookie` property of the DOM:

```
document.cookie='resolution='+Math.max(screen.width,screen.height)+'; path=/';
```

Listing 12: *JS code to store a cookie named resolution and the value of the screen size.*

The above JS code sets a cookie named resolution to the screen width or height of the client, whichever value is higher. The code can be wrapped by a `<script>` tag to be inserted in a HTML document. It should be inserted very early in the `<head>` tag of the document to ensure early execution. The used JS property is `screen.width` respectively `screen.height`. Both properties are standardized, but browser implementations vary. Most current mobile browsers return the resolution of the ideal viewport in CSS pixels. Older mobile browsers and desktop browsers return the physical screen resolution in device pixel. The inconsistent implementation of `screen.width` and `screen.height` can produce unexpected or wrong results in some browsers. Detailed compatibility tables for many browsers can be found at [50], [102]. [49]

The above JS code only stores the screen size in the cookie. Other characteristics can be added if needed. The Adaptive Images script supports image selection based on the DPR. The following listing shows how to store not only the screen size but also the DPR in the cookie.

```
document.cookie = 'resolution=' + Math.max(screen.width,screen.height) +
  ("devicePixelRatio" in window ? "," + devicePixelRatio : ",1") + '; path=/';
```

Listing 13: *The above JS snippet, but enhanced by storing the client's DPR as well.*

So far we have configured the web server to forward image requests and set the cookie on the client using JS. It is now time to take a look at the Adaptive Images image handling script. It allows the developer to specify the used CSS breakpoints within the image handler. The script then takes the cookie-stored screen size of the user and selects the closest breakpoint dimension. This dimension is determined as the ideal image width. The script is capable of creating scaled versions of the requested image to match the breakpoint widths. All created versions are cached on the server to prevent recurring rescaling on subsequent requests. The selected or created image is then delivered to the client. Adaptive Images sets the HTTP Cache-Control header to private. This is to prevent intermediary proxy servers from caching the response for the earlier mentioned reasons. [99]

It can happen that the cookie is not available to the script. This might either be due to the user having cookies disabled in the browser or due to a race-condition when setting the cookie. The latter occurs when the lookahead parser is not waiting until the JS code gets executed to set the cookie, which is the case at least in Internet Explorer 9 [103]. Adaptive Images handles missing cookies by implementing a simple user agent detection to determine if it is dealing with a mobile device. If so, it chooses the lowest resolution image version automatically. This is not ideal, as mobile devices do not necessarily have a low screen resolution. However, the server somehow has to select an image and it can yield better results than always sending out the biggest image. Adaptive Image's workaround also gives an idea of how different solutions can be combined to complement each other or serve as a fallback.

The Adaptive Images script requires JS to be enabled, but it also provides a fallback for users without JS. CSS media queries are used to trick the browser into requesting a fictitious background image. The background image URL contains the upper limit of the media query width range as a GET parameter. The following listing shows how to implement the fallback on the client-side.

```
<style>
  @media only screen and (max-device-width: 479px) {
    html {
      background-image:url(cookie.php?maxwidth=479);
  }}
  @media only screen and (min-device-width: 480px) and (max-device-width: 640px) {
    html { background-image:url(cookie.php?maxwidth=640);
  }}

  /* more media queries can be added here to increase precision. */

</style>
```

Listing 14: *Using CSS media queries to initiate requests to transmit the screen width and set a cookie.*

The requested cookie.php script creates the cookie from the server-side without needing JS. However, this workaround harms the performance, as the CSS parsing and sending of the image request takes too much time for the cookie to be set before the first image requests. In consequence, responsive images can only be delivered for subsequent requests after the first page load. The evaluation was performed assuming that JS is required and available, but this non-JS workaround is shown anyway for the sake of completeness. [99]

Evaluation

The evaluation was performed according to the evaluation framework that was defined in Section 3.3. The comment column explains the assigned rating.

Functional Req.	Support	Comment
Resolution-based selection	Wide	Supported, but reacting to dynamic changes of the user environment is not possible without using JS events to reissue image requests.
Viewport-based selection	No	The display width of the images can not be known on the server-side by only using cookies.
DPR-based selection	Medium	Supported, but reacting to dynamic changes of the user environment is not possible without using JS events to reissue image requests. No method to inform the device about the content's DPR Thus, the display size of the image has to be specified within the HTML or CSS code.
Art Direction	Medium	Art Direction based on all JS-detectable device characteristics is possible. Media queries are supported using the JS function matchMedia. However, no dynamic reaction to changes is supported.
Match media types and features	Medium	The JS function matchMedia can be used to evaluate a CSS media queries. The result can be stored in the cookie to deliver images based on it. No native selection by the browser possible, eg. when printing a document.
Image format-based selection	Medium	JS can be used to determine support for image formats. Additionally, the HTTP Accept header sent by the client could provide information about supported image formats.
User agent-based decisions	No	There is no standardized way of instructing the server to deliver a certain version (eg. small, to save bandwidth) of images.

Table 9: *Evaluation results of the functional requirements of Solution 2.*

Non-Func. Req.	**Rating**	**Comment**
Browser support	High	As long as the client-sided system prerequisites are met (JS and cookies), browser support is very good.
Performance	Medium	As for all server-sided approaches, it is possible to (a) use the same URL for different image versions in a HTML or CSS document or (b) directly insert the URL of the most-appropriate image into the HTML or CSS on the server-side. Both results in the image URLs being available directly within the HTML code. Thus, the browser's lookahead parser can initiate image requests very early. Responses can not be cached by proxies or CDNs. A race-condition might prevent the first image requests from being processed correctly and can slow down the page load time. Responses can not be cached by proxies or CDNs.
Reliability	Medium	The `screen.width` property of the DOM is implemented differently by browsers. The solutions mostly produces decent but not ideal results. This can negatively affect the correct selection of images.
Integrability	High	This solution can easily be integrated into an existing application without changing the markup. The server can handle the image selection and the generation of different image versions.
Flexibility	Low	Reacting to dynamic changes of the user environment by re-issuing image requests is possible using JS events, but cumbersome to implement. The available device characteristics are limited to what can be retrieved using JS.
Accessibility	High	The `<img>` tag and its alt attribute to provide an alternative text can be used to correctly outline the content as an image. The image source can be placed into the HTML document directly on the server, resulting in semantically correct HTML code.

Table 10: *Evaluation results of the non-functional requirements of Solution 2.*

System Prereq.	**Needed**	**Comment**
JavaScript	Yes	JS, Server-Side Logic and cookies are each a key component of this solution. JS is used to retrieve the device characteristics and to save them in a cookie.
Server-Side Logic	Yes	Server-sided logic is needed to handle the entire process of making images responsive. Thus, a server-sided scripting language and a configurable web server is required.
Cookies	Yes	Cookies are needed to store the device characteristics.

Table 11: *Evaluation results of the system prerequisites of Solution 2.*

3.4.4 Solution 3: HTTP Client Hints

Solution 1 and 2 utilized rather hacky approaches provide let the server with device characteristics. Solution 3 makes use of a more sophisticated and standardized approach called HTTP Client Hints. It extends the HTTP protocol by a set of headers which hold information about the device. It will eventually be standardized as a *Request for Comments* (RFC) document and is currently in draft status [104]. The following headers are currently available to be added by the client [105]:

1. **DPR (device pixel ratio):** This header contains the current DPR of the client. Every screen has a default, fixed DPR. However, on desktop systems the value changes when the user zooms the content. Enlarging the content (zooming in) increases the DPR while zooming out decreases the DPR.
2. **Width:** This header advertises the width in CSS pixels at which the resource will be displayed. The browser can derive the width from the `<img>`'s width or sizes attribute. CSS declarations are not planned to be taken into account as stylesheets are usually not yet available when the lookahead parser encounters the image. If the display width of the resource can not be determined, the browser should use the viewport width.
3. **Viewport-Width:** This header indicates the current viewport width of the client in CSS pixels. On desktop systems, this value changes when the user resizes the browser window or zooms the content. The change due to zooming is opposed to the DPR: zooming in decreases the viewport width while zooming out increases the viewport width.
4. **Downlink:** This header contains the maximum speed of the client's network connection in *megabits per second* (Mbps). The possible values are derived from W3C's Network Information API, which defines a table for many different connection types like GSM, UMTS, WiFi, Ethernet, and more. It should be noted that the indicated speed always refers to the maximum possible speed of the underlying technology. The actual current speed might be (much) slower. Thus, the benefits of this header are limited and its final implementation is still unclear.
5. **Quality (planned):** This header contains a quality keyword to let the client indicate which resource quality it prefers. It is a form of content negotiation and has diverse use cases. The user (agent) may prefer resources of "low" quality because of a slow network connection or a mobile plan with limited bandwidth. This header was in the draft document of the specification but has been removed again due to its missing definition of possible quality keywords and their meaning. It is planned to be added again after a revision.

These headers allow the client to inform the server about the client's characteristics relevant to responsive images. It is up to the user agent which headers are included in a request. The current draft suggests to only include headers on an opt-in basis. This means the client should only send the headers after the server has advertised support for them. The server does that by adding the CH-Accept header containing the supported client hints to its responses. The opt-in mechanism is used to avoid the overhead of sending client hints when the server does not support or use them anyway. [104]

An advantage of all server-sided approaches is their easy integration into existing applications. This is especially true for HTTP Client Hints. No changes to the markup or the application logic are required. A solution can be implemented directly into web servers by 3rd party plugins or modules. The same applies to intermediary proxy servers. Such plugins could handle the entire process – image selection, optionally scaling, and delivery – without making any changes to the underlying application. This makes HTTP Client Hints very interesting for legacy applications and websites. Currently, no such plugins are yet available though. As an alternative, a custom image handling script as used for solutions one and two can be used as well. This again requires to configure the web server to forward image requests to a custom script.

The other covered server-sided solutions suffered from caching issues when delivering different images for the same URL. This issue can be overcome with HTTP Client Hints. If the server uses one or more hints to select a resource, the server has to include the Vary header in its response. The Vary header indicates which request headers were considered to select the resource. For example, if the server selects a resource based on the Width and DPR headers, it must include a `Vary: Width, DPR` header in its response. The optional Key header can be used to provide more detailed information. Proxy servers use the Vary and Key headers to learn about the resource characteristics and when to send which resource. [106]

A big disadvantage of HTTP Client Hints is its currently poor support by browsers and web servers. At the moment, no major browser supports them by default. Google's Chrome browser has limited support in its developer version Chrome Canary. Some community members, especially RICG member Yoav Weiss, put a lot of effort in shipping an implementation to the Blink and WebKit engine. However, there are ongoing discussions on the various engine and browser bug trackers about how (and if) HTTP Client Hints should be implemented [107]–[110]. Some argue that the draft spec is not ready for implementation yet and that there are too many open issues. However, an implementation in Blink looks promising [111]. It would provide support for Chrome and Opera. The implementation status for Mozilla's Gecko engine used by Firefox is *"unconfirmed"* [112]. Microsoft's current implementation status for Internet Explorer is *"under consideration"* [113].

Implementation

The HTTP Client Hints specification is still in draft status. At the time of writing, there are no available implementations for web servers or applications. Implementations are likely to be released after the first browser(s) support HTTP Client Hints. An implementation could work very similarly to those of solutions one and two. The main difference with HTTP Client Hints is that the device characteristics are transmitted via HTTP headers instead of cookies or the user agent string. The following listing shows how to access the respective HTTP headers using PHP.

```
<?php
  $width            = $_SERVER['HTTP_WIDTH'];
  $devicePixelRatio = $_SERVER['HTTP_DPR'];
  $viewportWidth    = $_SERVER['HTTP_VIEWPORT_WIDTH'];

  /* image handling code should be inserted here */
?>
```

Listing 15: *PHP Code to retrieve HTTP headers sent by the client.*

HTTP Client Hints is the only server-sided solution that includes a method to inform the client for which DPR the returned content is optimized. The client needs this information to render the image with the correct size, if no explicit width and height was assigned to the image using CSS. The HTTP Client Hints specs provide the Content-DPR header to do that. The following two listings of a HTTP conversation will help understanding. First, the client requests an image called image.jpg and adds the DPR and Width client hints.

```
GET /image.jpg HTTP/1.1
User-Agent: Some Browser
Accept: image/webp, image/jpg
DPR: 2.0
Width: 160
```

Listing 16: *A sample HTTP request for an image with enabled HTTP Client Hints.*

The client advertises the server that it wants to display the image with 160 CSS pixels wide and a DPR of 2. Thus, the ideal image would have an intrinsic size of 320 (160×2) pixels. If the server has several image versions available, it should select the one which closest matches the client hints. Alternatively, the server can generate an image specifically for the client by scaling a high resolution version to fit the client's preferences. The server then responds with the image and adds the Content-DPR header to indicate that the client hints have been considered. The following listing shows the response to the above image request. The Content-DPR header informs the client that the resource is optimized for a DPR of 2. The result is that the client does not scale the image anymore and uses a DPR of 1 to render it. The listing also shows how to use the Vary header to allow intermediary caching. The server sets the Vary header to “DPR, Width” to indicate that the response was chosen based on the headers DPR and Width. [104]

```
HTTP/1.1 200 OK
Server: Apache
Content-Type: image/jpg
Content-Length: 75645
Vary: DPR, Width
Content-DPR: 2.0
[blank line to separate headers and content]
[image data follows here]
```

Listing 17: *A sample response to Listing 16, with the Content-DPR and Vary headers set.*

Other than that, an image handler script could work very similarly to Adaptive Images [99], a popular sample implementation of Solution 2.

Evaluation

The evaluation was performed according to the evaluation framework that was defined in Section 3.3. The comment column explains the assigned rating.

Functional Req.	Support	Comment
Resolution-based selection	Wide	Supported, but reacting to dynamic changes of the user environment may be limited. It is up to the user agent to re-request an image when the environment changes.
Viewport-based selection	Medium	Supported, if the sizes or width attribute of `<img>` tags is set so that the browser can determine the display width of the image. Reacting to dynamic changes of the user environment may be limited. It is up to the user agent to re-request an image when the environment changes.
DPR-based selection	Wide	Supported, but reacting to dynamic changes of the user environment may be limited. It is up to the user agent to re-request an image when the environment changes.
Art Direction	Medium	Art direction based on the available client hints (Width, DPR) is possible, but not more. HTTP Client Hints are designed to be combined with the native `<picture>` element if art direction is needed.
Match media types and features	No	The specification does not provide headers to inform the server about the client's media type or features.
Image format-based selection	Medium	The solution is not able to explicitly indicate the content type of images, but the HTTP Accept header can be used for content negotiation.
User agent-based decisions	Medium	A header to indicate a preferred resource quality is planned in the specification. However, the server decides which resource is delivered and the user agent can not choose.

Table 12: *Evaluation results of the functional requirements of Solution 3.*

Non-Func. Req.	Rating	Comment
Browser support	Low	At the time of writing no major browser or web server supports HTTP Client Hints. Support is expected for Blink (Chrome and Opera), Internet Explorer's status is „under consideration” and Firefox's status is „unconfirmed“.
Performance	High	As for all server-sided approaches, it is possible to (a) use the same URLs for images in a HTML document or (b) directly insert the URL of the most-appropriate image on the server. Both results in the image URLs being available directly within the HTML code. Thus, the browser's lookahead parser can initiate image requests very early. Responses can be cached by proxies and CDNs.
Reliability	High	HTTP Client Hints produce reliable and accurate results if it is supported by both the browser and the web server.
Integrability	High	HTTP Client Hints can easily be integrated into an existing application without changing the markup. The server can handle the image selection and the generation of different image versions.
Flexibility	Medium	Re-requesting an image when the user environment changes is up to the user agent according to the draft specification. Another specification that documents implementation details of browsers is planned.
Accessibility	High	The `<img>` tag and its alt attribute for an alternative text can be used to correctly outline the content as an image. The image source can be placed into the HTML document directly on the server, resulting in semantically correct documents.

Table 13: *Evaluation results of the non-functional requirements of Solution 3.*

System Prereq.	Needed	Comment
JavaScript	No	JS is not needed for HTTP Client Hints, but it could be used to extend to solution.
Server-Side Logic	Yes	Server-sided configuration (and a scripting language) is needed as image selection and delivery is handled by the web server or a custom script within the application.
Cookies	No	–

Table 14: *Evaluation results of the system prerequisites of Solution 3.*

3.4.5 Solution 4: CSS Background Images

After covering three server-sided solutions it is time for the client-sided solutions. The first solution we look at are CSS Background Images. It is covered first because it is based on CSS media queries, which are needed again for Solution 5 and 6. The switch from the server to the client-side is not the only change. CSS Background Images is also the only evaluated solution which does not use HTML's `<img>` tag to display images. Instead, the background image feature of CSS is used, which is a fundamentally different way to display images on websites. As the name suggests, images implemented this way are displayed in the background of the actual content. There are several CSS properties to adjust the appearance of background images. CSS background images do not require a special HTML tag and can be applied to any HTML element. Using CSS background images instead of foreground images based on the `<img>` tag goes along with several advantages and disadvantages.

The main advantage is that media queries can be used to conditionally set background images. CSS media queries are covered in Section 2.4.2 and in the following Implementation Section. They allow to select images based on the viewport width, DPR, screen type, and much more.

The relevant CSS background image properties and media queries are very well supported by all major browsers [114]. Only Internet Explorer 8 and below does not support media queries. However, support for most media features can be added using the Media.match polyfill [115]. Older Safari and Android browser versions and Internet Explorer 9 to 11 do not support nested media queries [114]. Nested media queries are not required for any of the presented methods, but they allow for simpler and easier to read CSS code.

The CSS Background Images solution provides a good performance. The CSS declarations can be included directly within the HTML document. In consequence, the declarations are available as soon as the HTML file has been loaded. If the separation of markup and style is important, a separate CSS file can be used. This introduces some delay because the file has to be requested before any images can be fetched and displayed. All media queries for one image should be written to be mutually exclusive. This prevents the browser from downloading more than one image version. [116]

A downside of CSS background images is that less accessibility features are available. There is no equivalent to `<img>`'s alt attribute to provide an alternative text for visually impaired users or when the image request fails. Furthermore, CSS background images are not outlined as an image and can be applied to any HTML element. The result is a lack of semantics. While this is less of an issue for decorative images, contentual images should always be outlined as images to provide semantically correct code.

Another drawback of CSS background images is their cumbersome implementation. The required mutually exclusive CSS media queries are rather complex to write and hard to read. Media queries can not be applied directly to an element within the HTML code as inline CSS. Instead, they have to be applied using a `<style>` tag or a separate CSS file. This maintainability difficulties are especially relevant on websites with many images. Furthermore, it is inconvenient to add media query-based CSS declarations queries using JS. This is needed when responsive images are dynamically added to a page without performing a reload of the page.

Implementation

CSS background images can be applied to any HTML element. The following listing shows a simple declaration for a background-image which is automatically scaled (and cropped) to fill the entire target element.

```
#some-element {
  background-image: url('my-image.jpg');
  background-repeat: none;
  background-size: cover;
  background-position: bottom right; /* let us assume the bottom right part of the image
                                        contains important information and should not be
                                        cropped by the background-size: cover; declaration */
}
```

Listing 18: *A simple example of a CSS background image.*

The following shows a list of CSS properties that are relevant to background images [14, p. 240].

1. **background-image:** This property sets the image source. The source can be provided as an URL or data URI scheme. Both methods require the value to be wrapped with the CSS url() function.
2. **background-repeat:** This property specifies if the image should be horizontally or/and vertically repeatedly displayed if the HTML element is bigger than the image. Possible values are *no-repeat*, *repeat-x*, *repeat-y*, and *repeat*, which is the default.
3. **background-size:** This property allows to set the display size of the background image in CSS pixels. It accepts absolute and relative units for the horizontal and vertical axis. This property is needed to correctly display DPR-optimized images by specifying the desired dimensions in CSS pixels. This prevents the browser from upscaling the image by the DPR.

   Furthermore, the keywords *cover* and *contain* can be used to automatically adjust the image to fill (cover) or fit within (contain) the parent element by cropping and rescaling it.

4. **background-position:** This property allows to set the position of the image relatively to the containing element. Two space-separated values or one for both the horizontal and vertical position are accepted. Possible values can be relative and absolute vales as well as the keywords *top*, *left*, *right*, *bottom*. This property is usually not needed for responsive images but is listed here for completeness. It is commonly used for a technique called CSS sprites and together with the background-size property to align fullscreen images.

For a full coverage of all properties and their syntax related to background images, [14, p. 240] is recommended. CSS background image declarations can easily be wrapped within media queries to deliver responsive images. The following list gives an overview of the relevant media features which can be used in media queries [70].

1. **(min-/max-)width:** The width media feature is evaluated against the width of the viewport in CSS pixels. It is almost always used with the min- or max- prefix to target devices below or over a certain viewport width. By combining a min-width and max-width media feature with the `and` keyword, a range of widths can be targeted.

   The CSS specs also define a device-width media feature which should be evaluated against the screen width in theory. However, in practice it is sometimes the actual screen width in device pixels and sometimes the width of the ideal viewport. Thus, it is recommended to use the width media feature. [70]
2. **(min-/max-)resolution:** The resolution media feature allows to target certain DPRs. The *dots per 'px'* (dppx) unit should be used to avoid having to convert the DPR to the dpi unit. [70]
3. **orientation:** The orientation media feature can be used to check if a device is in landscape or portrait mode. If the height resolution is greater or equal than the width, the device is in portrait mode, and in landscape otherwise. [70]
4. **monochrome:** This media feature can be used to target monochrome screens as they are for example commonly used for e-book readers. They can be evaluated with an optional number, which indicates the number of bits used for each pixel. [70]

Responsive images can easily be implemented using these media features. It is important that the queries are mutually exclusive, which means that only one media query evaluates to true at a given moment [116]. This is to prevent the browser from loading more than one image. The following listing shows an example of mutually exclusive media queries to deliver responsive images for different viewport widths.

```
#an-image {
  /* general declarations valid for all viewport sizes go here */
}

/* target screens up to 320px in width */
@media all and (max-width: 320px) {
  #an-image { background-image: url('image-320px.jpg'); }
}

/* target screens of 321 up to 640px in width */
@media all and (min-width: 321px) and (max-width: 640px) {
  #an-image { background-image: url('image-640px.jpg'); }
}

/* target screens of 641 up to 1200px in width */
@media all and (min-width: 641px) and (max-width: 1200px) {
  #an-image { background-image: url('image-1200px.jpg'); }
}

/* target screens larger than 1200px in width */
@media all and (min-width: 1201px) {
  #an-image { background-image: url('image-1920px.jpg'); }
}
```

Listing 19: *Media queries to implement resolution-based selection with CSS background images.*

Implementing a DPR-based selection is more complicated. The WebKit engine was one of the first engines which implemented a media feature for the pixel density of a screen. They called the media feature `device-pixel-ratio` and it had to be prefixed with "`-webkit-`". Eventually the CSS specs caught up and standardized the `resolution` media feature. For compatibility with Safari and older browsers, it is recommended to provide all (prefixed) versions in the CSS declarations, as shown in the following listing. [117]

```
@media
only screen and (-webkit-min-device-pixel-ratio: 2),
only screen and (   min--moz-device-pixel-ratio: 2),
only screen and (     -o-min-device-pixel-ratio: 2/1),
only screen and (        min-device-pixel-ratio: 2),
only screen and (                min-resolution: 192dpi),
only screen and (                min-resolution: 2dppx) {
  /* Declarations for DPR=2 screens go here */
}
```

Listing 20: *Target DPR=2 screens using prefixed media feature versions for different browsers.*

The recommended, most convenient and future-proof media feature is resolution together with the dppx unit, which uses the same metric as the DPR values we have seen in this work. However, the dppx unit is not supported by Internet Explorer 11 and below [118]. The alternative is to use the dpi unit, where 96 dpi equal a DPR of 1, 192 dpi a DPR of 2 and so on.

Another issue with DPR-based media queries is writing mutually exclusive queries. Using the dppx unit, floating point numbers are needed to correctly cover the desired ranges. Browsers use different precisions when working with fractions in CSS [119]. This could possibly lead to unexpected behavior and should be tested accordingly. The following listing shows DPR media queries to implement a DPR-based selection using CSS. The prefixed media feature versions have been omitted for better readability.

```
#an-image {
  background-size: 800px 600px;
}

@media only screen and (min-resolution: 1.0dppx) and (max-resolution: 1.25dppx) {
  #an-image {
    background-image: url( 'image-1x.jpg' );
  }
}
@media only screen and (min-resolution: 1.26dppx) and (max-resolution: 1.999dppx) {
  #an-image {
    background-image: url( 'image-1.5x.jpg' );
  }
}
@media only screen and (min-resolution: 2dppx) {
  #an-image {
    background-image: url( 'image-2x.jpg' );
  }
}
```

Listing 21: *Using media queries to implement DPR-based selection with CSS background images.*

The *match media type and features* use case is implemented the same way as the above listings. The monochrome media feature can be used to target devices with monochrome screen, eg. e-book readers.

```
@media all and (monochrome) and (orientation: landscape) {
  #an-image {
    background-image: url( 'image-monochrome-landscape.jpg' );
  }
}
```

Listing 22: *Using media queries to target monochrome screens in landscape mode.*

Evaluation

The evaluation was performed according to the evaluation framework that was defined in Section 3.3. The comment column explains the assigned rating.

Functional Req.	**Support**	**Comment**
Resolution-based selection	Full	Fully supported using media queries.
Viewport-based selection	Full	Fully supported using media queries.
DPR-based selection	Full	Fully supported using the resolution media feature of media queries.
Art Direction	Full	Fully supported using media queries.
Match media types and features	Full	Fully supported using media queries.
Image format-based selection	Medium	The solution is not able to explicitly indicate the content type of images, but the HTTP Accept header can be used for content negotiation.
User agent-based decisions	No	There is no standardized way to instructing the server to deliver a certain versions (eg. small, to save bandwidth) of images.

Table 15: *Evaluation results of the functional requirements of Solution 4.*

Non-Func. Req.	Rating	Comment
Browser support	High	All major browsers support CSS background images and CSS media queries.
Performance	Medium	The CSS code can be integrated directly within the HTML document using the `<style>` tag. This is the fastest option, as the CSS declarations are available immediately when the HTML code has been received. For easier maintenance on image-heavy websites, a separate CSS file is recommended. However, this adds some delay for fetching the CSS file. Mutually exclusive media queries should be used to prevent the browser from downloading more than one image version.
Reliability	Medium	CSS media queries are widely supported, but are inconsistently implemented across browsers. The solution mostly produces decent but not always ideal and correct results.
Integrability	Low	Writing mutually exclusive media queries is cumbersome and results in bloated code. For websites with only a few images this is not an issue, but for image-heavy websites this makes the integration difficult and hard to maintain.
Flexibility	Medium	Reacting to dynamic changes of the user environment by re-issuing image requests is possible using CSS media queries.
Accessibility	Low	This solution does not use the `<img>` element. Thus, the alt attribute to provide an alternative text is not available.

Table 16: *Evaluation results of the non-functional requirements of Solution 4.*

System Prereq.	Needed	Comment
JavaScript	No	–
Server-Side Logic	No	–
Cookies	No	–

Table 17: *Evaluation results of the system prerequisites of Solution 4.*

3.4.6 Solution 5: JavaScript

The second evaluated client-sided approach is JavaScript. Its name is less meaningful compared to the other solutions. This is because JS solutions can be very diverse and many implementations are possible. The general idea is to use JS to set or change the src attribute of `<img>` tags.

The use of JS goes along with advantages as well as disadvantages. A big advantage of JS solutions is that they are very flexible. JS can detect a wide range of device characteristics which can be used for the image selection process. This includes screen size, viewport size, DPR, size of certain HTML elements on the page, browser features and many more. There are also ways to measure the connection speed of the client, although the test takes time to run and the results may not be accurate [120].

Another advantage of JS solutions is their high browser support. As long as JS is available and enabled in the client's browser, it can be supported. JS features are not consistently supported by all browsers and the specific implementations vary. However, usually it is possible to write functions which are compatible across all browsers. A JS framework like jQuery might be a sensible option to help with that. An issue are inconsistently implemented properties like `screen.width`. On most desktop systems it returns the width of the screen in device pixels, but the width of the ideal viewport on most mobile devices [49], [50].

Probably the biggest disadvantage of JS solutions is their poor performance. The browser's lookahead parser can not be used to optimize the loading process. As we learned in Section 2.2.5, JS gets executed immediately when it occurs in the code. The browser's main parser halts until the JS code has executed. This halting of the entire rendering process significantly increases the loading time. To prevent the parser from halting, the `<script>` tag(s) should be put at the end of the document. Another option is to add the async or defer attribute to the `<script>` tag to delay the loading and execution of such. However, both results in the JS code only being executed after the DOM has been fully created and is ready, which is indicated by the DOMContentLoaded event. This behavior prevents the lookahead parser from preloading any responsive images. [37]

Many JS implementations go along with a lack of semantics in the HTML code. The reason was partly explained in the last paragraph. The src attribute is set using JS and usually only after the DOM has been fully created. This may result in semantically incorrect documents when JS is not available. The possible options to solve this issue are addressed in the Implementation Section.

The dependency on JS is a general downside of JS-based approaches. Users without JS should be provided with a fallback so that they can see images as well. Possible workarounds are covered in the following Implementation Section.

Implementation

As mentioned earlier, a JS-based solution sets or changes the src attribute of `<img>` tags using JS. There are several ways to do that, some of which are shown in the following listing.

```
// set an attribute using plain JS
document.getElementById('an-image').setAttribute('src', 'http://example.com/image.jpg');

// using a more sophisticated selector engine with CSS selectors support
document.querySelector('#an-image').setAttribute('src', 'http://example.com/image.jpg');

// using jQuery to do the same
$('#an-image').attr('src', 'http://example.com/image.jpg');
```

Listing 23: *Two JS-only and one jQuery method to set the src attribute of an image.*

An open issue of the above listings is how to supply the image URLs for the different image versions. A common solution is to use data attributes to provide URLs or filenames. For example, attributes named data-1x and data-2x could be provided to offer images for HiDPI screens. The following listing shows example code which changes the src attribute of an image based on the DPR using jQuery.

```
<img src="default-image-1x.jpg" data-2x="image-2x.jpg" alt="Alternative text for the image.">
<script>
  // wait for the DOM to be ready (DOMContentLoaded event)
  $(document).ready(function() {
    // check if the device pixel ratio (DPR) is at least 1.5
    if (window.devicePixelRatio >= 1.5) {
      // select images with the data-2x attribute and change the src attribute to
      // the value of the data-2x attribute
      $('img[data-2x]).attr('src', $(this).attr('data-2x'));
    }
  });
</script>
```

Listing 24: *Using data attributes to store different DPR-based image versions and apply them using jQuery.*

The same approach can be applied for an image selection based on the viewport width. The data attributes could be named data-400 and data-800 to provide 400 and 800 pixel wide image versions. Getting the correct viewport width in a cross-browser compatible way is a difficult to impossible task. The best available property is `screen.width`, but it is inconsistently implemented as we have seen in the evaluation of Solution 2.

The following shows a list of properties related to the viewport. All except the DPR have an equivalent for the height.

1. **screen.width:** This property should return the screen resolution in CSS pixels according to the specification [45, Sec. 5.3]. However, most desktop browsers return the value in physical pixels. At the time of writing, the desktop versions of Firefox and Internet Explorer adjust the value according to the DPR when using the zoom function. On mobile, some devices return the screen resolution in device pixels, but most devices return the size of the ideal viewport, which is considered the correct behavior in the community. [49], [50], [102]
2. **window.innerWidth:** On desktop systems, this property returns the width of the viewport in CSS pixels. Firefox and Internet Explorer are counting scrollbars to the viewport size. On mobile, it returns the width of the visual viewport in CSS pixels. Thus, it is adjusted when using the pinch zoom function. [102]
3. **document.documentElement.clientWidth:** On desktop, this property returns the width of the viewport in CSS pixel, but without scrollbars. On mobile, it returns the width of the layout viewport in CSS pixels. [102]
4. **jQuery(window).width():** This value is retrieved using a jQuery function rather than a native JS DOM property. On desktop, it is the viewport width in CSS pixels. On mobile, it is the layout viewport in CSS pixel.
5. **window.devicePixelRatio:** On desktop, this property returns the current DPR, considering the zoom level. On mobile, it returns a fixed value which indicates the DPR of the screen. [102]

The above list indicates that the support for properties is inconsistent and it is hard to retrieve the intended value across all devices and device types. The `screen.width` property usually provides a decent approximation of the value we are interested in. When the viewport `<meta>` tag of Section 2.4 is used, the `document.-documentElement.clientWidth` property can serve as an alternative to retrieve the ideal viewport. However, a lot of testing is recommended to find the correct value for given target devices.

The following listing can be used to display the above properties for testing purposes. The values are updated every 250ms in case they change, eg. when using the zoom function.

```
<script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>

<p>screen.width: <span></span></p>
<p>window.innerWidth : <span></span></p>
<p>document.documentElement.clientWidth (layout viewport): <span></span></p>
<p>window.devicePixelRatio: <span></span></p>
<p>$(window).width(): <span></span></p>

<script>
  setInterval(function(){
    $('span:eq(0)').html(screen.width);
    $('span:eq(1)').html(window.innerWidth);
    $('span:eq(2)').html(document.documentElement.clientWidth);
    $('span:eq(3)').html(window.devicePixelRatio);
    $('span:eq(4)').html($(window).width());
  }, 250);
</script>
```

Listing 25: *Testing different JS properties related to the viewport and the device's screen.*

Let us now cover the issue of how to initially deliver the `<img>` tags in the HTML document. There are several options, all of which have their pros and cons. One option is to serve `<img>` tags with the src attribute set to a default image. Doing so, even users with no JS will see an image. It might not be the best image for them, but at least there is an image. This option also allows the lookahead parser to preload the default images, which is basically a good thing. However, the preloading might introduce some overhead. When the page is loaded, first the HTML code is parsed and at the same time the lookahead parser will request the default image(s) that are set in the src attribute. Then the JS code runs to determine the best fitting image. If the image chosen by JS image is not the same as the preloaded default image, the new image has to be requested. This results in some overhead as the preloaded image is not being used and up to two image requests are needed for every responsive image on the page.

This overhead can be avoided by omitting the default image in the src attribute. However, serving an empty src attribute results in invalid HTML code. This can cause problems and unexpected behavior in some browsers [121], [122]. Furthermore, valid code is important for search engines and accessibility. The code can be made valid by using the data URI scheme, which was introduced in Section 2.5. It allows to provide a Base64 encoded image directly within the src attribute. By providing a transparent 1×1 pixel GIF image, the code gets valid and there is no overhead caused by additional image requests. The following listing shows such an inline image [123]:

```
<img src="data:image/gif;base64,R0lGODlhAQABAAAAADs=">
```

Listing 26: *HTML `<img>` tag with an embedded transparent 1×1 pixel GIF image using data URI scheme.*

However, when using this option there is no default image anymore. This means the lookahead parser can not retrieve the image, which might be the best image already. Furthermore, non-JS users will only see the 1×1 pixel GIF. That said, all options are a compromise between user support and performance.

Now let us go back to solving the given use cases. As we have seen for the last evaluated solution, CSS media queries offer a powerful way to select images. JS offers its native matchMedia function to evaluate media queries. It allows to implement resolution- and viewport-based selection, art direction, and allows to match media types and features. Event handlers can be bound to individual media queries. JS does automatically fire the event when the evaluation of the media query changes from true to false or reverse. The following listing shows an example of the matchMedia function and media query event handlers. [74, p. 80]

```
var mediaQuery = window.matchMedia("(orientation: landscape)");
if (mediaQuery.matches) {
  /* this code is executed if the above media query evaluates to true */
}

/* bind a function to the media query that is executed on every evaluation change */
mediaQuery.addListener(function(mQ) {
  if (mQ.matches) {
    console.log("viewport is now in landscape mode.");
  }
  else {
    console.log("viewport is now in portrait mode.");
  }
});
```

Listing 27: *Evaluating media queries and binding a function to evaluation changes using JS.*

Reacting to dynamic changes of the user environment can also be handled using other JS events. For example, the `resize` event fires when the user resizes the browser window. The following listing shows how to listen for events using JS. [74, p. 80]

```
// Listen for orientation changes
window.addEventListener("resize", function() {
  /* this code is executed every time the size of the viewport changes */
});
```

Listing 28: *Binding a function to the resize event to react to changes of the viewport size.*

Evaluation

The evaluation was performed according to the evaluation framework that was defined in Section 3.3. The comment column explains the assigned rating.

Functional Req.	**Support**	**Comment**
Resolution-based selection	Wide	Supported using CSS media queries together with the JS function matchMedia.
Viewport-based selection	Wide	Supported using CSS media queries together with the JS function matchMedia. The display width of images is known as the JS code runs only when the page has been rendered.
DPR-based selection	Wide	The `window.devicePixelRatio` property can be used to implement a DPR-based selection. However, the property behaves different on desktop and mobile browsers. While it is fixed and device-specific on mobile, desktop systems adjust it when using the zoom function.
Art Direction	Wide	Supported using CSS media queries together with the JS function matchMedia.
Match media types and features	Medium	The JS function matchMedia can be used to evaluate a CSS media query. No native selection by the browser is possible, eg. when printing a document.
Image format-based selection	Medium	JS can be used to determine support for image formats. Additionally, the HTTP Accept header sent by the client could provide information about supported image formats.
User agent-based decisions	No	There is no standardized way of instructing the server to deliver a certain version (eg. small, to save bandwidth) of images.

Table 18: *Evaluation results of the functional requirements of Solution 5.*

Non-Func. Req.	Rating	Comment
Browser support	High	As long as the client-sided system prerequisites are met (JS), browser support is very good.
Performance	Low	Depending on the specific implementation, the browser's lookahead parser can either not be used at all, or the risk of requesting two images per displayed image is introduced.
Reliability	Medium	The relevant JS properties and CSS media queries are not consistently implemented across browsers. The solutions mostly produces decent but not always ideal and correct results.
Integrability	Medium	The markup of existing pages has to be changed and the JS code has to be integrated into the application.
Flexibility	Medium	Reacting to dynamic changes of the user environment by re-issuing image requests is possible using JS events, but cumbersome to implement. The available device characteristics are limited to what can be retrieved using JS.
Accessibility	Medium	The `<img>` tag and its alt attribute to provide an alternative text can be used to correctly outline the content as an image. However, depending on the specific implementation the HTML document may not be valid and semantically correct, eg. when using a 1×1 GIF placeholder image.

Table 19: *Evaluation results of the non-functional requirements of Solution 5.*

System Prereq.	Needed	Comment
JavaScript	Yes	JS is needed to handle the entire process of making images responsive.
Server-Side Logic	No	–
Cookies	No	–

Table 20: *Evaluation results of the system prerequisites of Solution 5.*

3.4.7 Solution 6: Native HTML5

The last evaluated approach is the rather new native HTML5 responsive images solution. It was developed by the RICG and was added to the WHATWG specs in August 2014. As it runs natively within the browser and was developed exclusively for providing responsive images, it outperforms most other available solutions. The solution operates on the client-side. All information about the different available image versions is outlined in the HTML code. The main components are the newly added `<picture>` element and the attributes srcset and sizes. The browser is provided with a set of available image versions. Every image can be explicitly assigned with its intrinsic width or the DPR it is optimized for, the image format, and the approximate display width. The client has free choice which image it selects and requests. This allows the browser to consider user preferences, eg. downloading small images when having a slow Internet connection. Media queries be used to force the use of a particular image to implement the art direction use case. That said, the solution supports all functional requirements of the evaluation framework. [84]

Besides, it offers several other advantages. Its performance is very good. The URLs of all image versions are available directly within the HTML document on the first page load. Thus, the browser's lookahead parser can preload the selected images.

The solution does not need any of the evaluated system prerequisites and works without JS, cookies, and server-sided logic.

The only drawback of this solution is browser support. At the time of writing, Chrome, Firefox, and Opera fully support the specification. Safari only supports the *DPR-based selection* use case using the srcset attribute with the pixel density descriptor. Internet Explorer does not support any of the specs, but partial support has been announced for Microsoft new browser Edge. [124], [125]

However, support for most unsupported browsers can be added with the picturefill polyfill [123]. It adds support by emulating the native implementation using JS. The dependency on JS results in only the image's alternative text from the alt attribute being shown when the user has JS disabled. The polyfill introduces some other minor drawbacks, which are covered in [123].

Implementation

The simplest way to deliver responsive images with this solution is to use the new srcset and sizes attributes. These attributes can be added to the existing `<img>` tag to maintain backwards compatibility. The srcset attribute allows to provide a comma-separated list of image URLs. Every URL is followed by a space character, a number, and either a pixel density descriptor (x) *or* a width descriptor (w). The pixel density descriptor can be a floating point number and represents the DPR the image is optimized for. The width descriptor indicates the intrinsic width of the image. A simple example for the DPR-based selection use case will help understanding. [81]

```
<img src="default_image_500px.jpg"
     srcset="image_750px.jpg 1.5x, image_1000px.jpg 2x"
     alt="Image of a cat.">
```

Listing 29: *Using the srcset attribute to provide different image versions based on the DPR.*

The above listing provides the client with three image versions. The src attribute contains the default image. It is displayed by legacy browsers without support for the srcset attribute. Supported browsers consider it for their selection and implicitly assign it a DPR of 1. Two more images optimized for a DPR of 1.5 respectively 2.0 are provided using the srcset attribute. Using pixel density descriptors the client can only perform a DPR-based selection. If a resolution- or viewport-based selection is needed, width descriptors have to be used instead.

When the srcset attribute contains at least one width descriptor, the sizes attribute has to be set as well. The sizes attribute provides the browser with hints of how big the image will be rendered. This information is necessary for the client to select an image with a width descriptor close to the actual display width. Otherwise, the client would have to assume that the image is displayed with the width of the viewport, which is often wrong and results in the selection of a too large image.

```
<img sizes="(max-width: 30em) 100vw, (max-width: 50em) 50vw, calc(33vw - 60px)"
     srcset="image-200.jpg   200w,
             image-400.jpg   400w,
             image-800.jpg   800w,
             image-1200.jpg 1200w
     src="image-400.jpg"
     alt="Image of a cat.">
```

Listing 30: *Implementing viewport-based selection using `<img>` with srcset and sizes attributes.*

The above listing provides the client with four different image versions. Again, the image in the src attribute acts as a fallback for older browsers. The srcset attribute provides four images with an intrinsic width of 200, 400, 800, and 1200 pixel. For the browser to select the best-fitting image, it has to know at which width the image will be displayed. CSS declarations are not yet loaded at this early stage of parsing and can not be used. The sizes attribute helps out and allows to provide the needed information in a comma-separated list. The sizes attribute of the above example contains two image sizes which are used when the attached media conditions evaluate to true. The last expression is used if no media condition evaluates to true. In detail, the above sizes attribute is processed as follows: If the `max-width: 30em` media condition evaluates to true, an image width of 100vw is assumed and the parsing of the sizing attribute has finished. The vw CSS unit stands for *viewport width* and is a percentage unit relative to the viewport width. For example, 50vw means 50% the width of the viewport. If the first media condition evaluates to false, the next condition is checked and so on. If no condition evaluates to true, the last item of the list is used, which can be a primitive CSS length or a calc() expression [15, Sec. 8.1]. The above expression means the image is displayed with a width of "33vw minus 60px". This example could be for an image which is displayed with 33vw and has a padding of 15px assigned on each side. The `<img>` tag in combination with the srcset and sizes attributes is sufficient for many use cases. However, to implement art direction or new image formats, the `<picture>` element is needed. [81]

The `<picture>` element acts as a wrapper for one or more `<source>` tags. Each `<source>` tag can have a srcset and sizes attribute, which behave the same as for the `<img>` tag. The difference of the `<source>` tag is that a media attribute can be added to enable art direction. All `<source>` elements of a `<picture>` tag are processed consecutively by evaluating the media attribute. The first matching `<source>` element is selected and its srcset attribute is used to select a resource. [81]

```
<picture>
  <source media="(min-width: 40em)" srcset="large.jpg">
  <source media="(min-width: 30em)" srcset="medium.jpg">
  <img src="small.jpg" alt="A cat.">
</picture>
```

Listing 31: *Implementing art direction using the `<picture>` and `<source>` tags.*

The above listing utilizes the `<picture>` and `<source>` tag to implement art direction. The large.jpg image is used if the viewport has a minimum width of 40em, the medium.jpg image is used if the viewport has a minimum width of 30em and the small.jpg image is used in all other cases and for older browsers without `<picture>` support.

The example can easily be enhanced to support viewport- or DPR-based selection as well. The srcset attribute of every `<source>` element behaves the same as when used with the `<img>` element. Multiple image versions can be provided using width and pixel density descriptors. If width descriptors are used, the sizes attribute has to be added. [81]

```
<picture>
  <source media="(min-width: 40em)" srcset="large-2x.jpg 2x, large-1x.jpg">
  <source media="(min-width: 30em)" srcset="medium-2x.jpg 2x, medium.jpg">
  <img src="small.jpg" alt="Another cat.">
</picture>
```

Listing 32: *Combining art direction and DPR-based selection using the `<picture>` and `<source>` tags.*

The type attribute can be added to `<source>` elements to explicitly specify the image format in form of an Internet media type, formerly known as *Multipurpose Internet Mail Extensions* (MIME) type. This allows to provide images in several image formats and let the user agent choose the one it supports or prefers. [84]

```
<picture>
  <source type="image/webp" srcset="image.webp">         <!-- image in WebP format        -->
  <source type="image/vnd.ms-photo" srcset="image.jpxr"> <!-- image in JPEG XR format     -->
  <img src="image.jpg" alt="A cat.">                     <!-- image in JPG f. (fallback)  -->
</picture>
```

Listing 33: *Implementing a selection based on image formats using the `<picture>` and `<source>` tags.*

An in-depth coverage of the native responsive images solution can be found at [11, Sec. 4.8], [81], [84].

Evaluation

The evaluation was performed according to the evaluation framework that was defined in Section 3.3. The comment column explains the assigned rating.

Functional Req.	**Support**	**Comment**
Resolution-based selection	Full	Fully supported using the `<img>` or `<source>` tag and their srcset and sizes attributes.
Viewport-based selection	Full	Fully supported using the `<img>` or `<source>` tag and their srcset and sizes attributes.
DPR-based selection	Full	Fully supported using the `<img>` or `<source>` tag and their srcset attribute with the pixel density descriptor.
Art Direction	Full	Fully supported using `<source>` elements with the media attribute. The elements must be wrapped by the <picture> element.
Match media types and features	Full	Fully supported using `<source>` elements with the media attribute. The elements must be wrapped by the <picture> element.
Image format-based selection	Full	Fully supported using `<source>` elements with the type attribute to provide the image type. The elements must be wrapped by the `<picture>` element.
User agent-based decisions	Full	The user agent is provided with a list of all image versions and can select one based on its preferences. However, at the moment no major browser considers user-based settings for the image selection process.

Table 21: *Evaluation results of the functional requirements of Solution 6.*

Non-Func. Req.	**Rating**	**Comment**
Browser support	Medium	At the time of evaluation, only supported in Firefox, Chrome, Opera, and the mobile versions of these browsers. Not supported by Internet Explorer, partly supported by Safari. The picturefill polyfill can be used to add (full) support for these browsers.
Performance	High	All image versions are available to the client directly within the HTML code. The browser's lookahead parser can be used to preload images. All responses are cacheable by proxy servers and CDNs.
Reliability	High	If the browser supports the solution, accurate and consistent results can be expected due to standardized implementations.
Integrability	Medium	The markup of existing pages has to be changed and different image versions have to be generated.
Flexibility	High	Re-requesting an image when the user environment changes is up to the user agent, but can be forced using art direction.
Accessibility	High	The `<img>` tag and its alt attribute for an alternative text can be used to correctly outline the content as an image.

Table 22: *Evaluation results of the non-functional requirements of Solution 6.*

System Prereq.	**Needed**	**Comment**
JavaScript	No	JS is not required for supported browsers, but it is required when using the picturefill polyfill to support browsers without native support for the solution.
Server-Side Logic	No	–
Cookies	No	–

Table 23: *Evaluation results of the system prerequisites of Solution 6.*

3.5 Evaluation Results

3.5.1 Overview

Functional Requirements	**Results for Solutions**[1]					
	1	**2**	**3**	**4**	**5**	**6**
Resolution-based selection	Poor	Wide	Wide	Full	Wide	Full
Viewport-based selection	No	No	Medium	Full	Wide	Full
DPR-based selection	Poor	Medium	Wide	Full	Wide	Full
Art Direction	Poor	Medium	Medium	Full	Wide	Full
Match media types / features	No	Medium	No	Full	Medium	Full
Image format-based selection	Medium	Medium	Medium	Medium	Medium	Full
User agent-based decisions	No	No	Medium	No	No	Full

Table 24: *Overall evaluation results of the functional requirements.*

Non-Functional Requirements	**Results for Solutions**					
	1	**2**	**3**	**4**	**5**	**6**
Browser support	High	High	Low	High	High	Medium
Performance	Medium	Medium	High	Medium	Low	High
Reliability	Low	Medium	High	Medium	Medium	High
Integrability	High	High	High	Low	Medium	Medium
Flexibility	Low	Low	Medium	Medium	Medium	High
Accessibility	High	High	High	Low	Medium	High

Table 25: *Overall evaluation results of the non-functional requirements.*

System Prerequisites	**Results for Solutions**					
	1	**2**	**3**	**4**	**5**	**6**
JavaScript	No	Yes	No	No	Yes	No
Server-Side Logic	Yes	Yes	Yes	No	No	No
Cookies	No	Yes	No	No	No	No

Table 26: *Overall evaluation results of the system prerequisites.*

1 Solution 1: User Agent Detection
Solution 2: Cookies
Solution 3: HTTP Client Hints
Solution 4: CSS Background Images
Solution 5: JavaScript
Solution 6: Native HTML5

3.5.2 Discussion

The results of the evaluation are quite clear. The native HTML5 solution fulfills all functional requirements and its non-functional requirements result is good as well. However, the best solution may vary for each project and its use cases. The purpose of this work's evaluation was not to find the best solution, but to provide a guideline and data to support decision making. Below, every solution will be briefly discussed and recommendations are given. Before this is done, a more general discussion about the evaluation follows.

As it was mentioned in Section 3.3, the evaluation was subject to various biases. First and foremost, the evaluation was performed by only one person, the author of this work. A web development background for more than ten years helped with the evaluation, but of course yields to personal experiences and preferences having an influence on the result. It was tried to apply a great sense of objectivity and minimize the influence by developing a formal evaluation framework. However, a personal bias was inevitably present.

The evaluation framework itself is subject to the same issues as stated above. It was created by only one person, the author of this work. Different requirements and evaluation proceedings might have been chosen by another person. An improvement for future work would be to strictly define how to evaluate each individual requirement and when to assign which rating option. Additionally, more gradual rating options for the non-functional requirements are suggested to enhance the current evaluation. However, to minimize the rating bias, more rating options would even more require that every individual rating option is strictly defined.

Some other issues arose when performing the evaluation process itself. An issue was that dimension of the implementation are not well-defined for the evaluated solutions. A very basic implementation of a presented concept is obviously less powerful than a more comprehensive one. Solving this issue artificially by limiting the implementation details of every evaluated solution was not found to be a reasonable solution. Instead, it was tried to evaluate all requirements based on the intrinsic limitations of the particular solution. These are limitations that can not be overcome by the developer or only with a disproportionate amount of time or by greatly increasing the complexity. Many possible implementations and their limitations were considered to find a reasonable frame for all solutions. This procedure was of course subject to a personal bias again. Nevertheless, it was found to be the best option considering the available time and the scope of this work.

A similar issue was the possibility to combine solutions to extend their capabilities. This is commonly done, especially combining server- and client-side solutions to get the best of both worlds. For the evaluation, it was assumed that only the evaluated solution was used. Sensible combinations are outlined in the comments of the evaluation tables and the following discussion of the individual solutions.

Let us now discuss the evaluation results. First, the general differences between server- and client-sided evaluation results are outlined. Server-sided approaches have in common that they are easy to integrate into existing applications or websites. They also provide good support for accessibility features. Their support for the functional requirements greatly varies between the solutions. Thus, the detailed results have to be considered.

Client-sided approaches have less similarities. Overall, their support of the functional requirements is better than of server-sided approaches. This is due to the fact that accessing device characteristics is easier when code runs directly on the client. However, all client-sided solutions have other flaws which have to be considered. The following briefly discusses each solution and gives recommendations.

Solution 1, User Agent Detection, is not recommended to be used as a sole solution for responsive images. The produced results are unreliable on mobile devices, and desktop systems can not be handled accurately at all. It is sufficient to detect if the user is on a mobile device. As desktop and mobile devices nowadays can have similar screen resolutions, knowing only the device category is not sufficient. That said, user agent detection should be used with caution. No system prerequisites make it a suitable for providing fallbacks or enhance other solutions.

Solution 2, Cookies, offer a decent server-sided solution until HTTP Client Hints are widely supported. Cookie-based solutions can easily be integrated into existing applications. The produced results are decent to good, depending on the implementation and the user's device and browser. Matthew Wilcox's Adaptive Images is a recommended implementation. It supports the *resolution-* and *DPR-based selection* use cases. Adding support for other use cases is possible, but cumbersome to implement and not recommended unless absolutely necessary.

Solution 3, HTTP Client Hints, is a very sophisticated approach. Due to its standardization it is future-proof and will work consistently across browsers once supported. The current lack of browser support is the biggest drawback of this solution. At the moment, no major browser implements HTTP Client Hints. Furthermore, the specification is still subject to changes until it is out of draft status. It is strongly recommended to wait for widespread browser support before using HTTP Client Hints as a sole solution. Until then, it can optionally be implemented as a second solution to immediately support the first browsers with client hints support.

Solution 4, CSS Background Images, uses an entirely different approach to display images. This goes along with several advantages and disadvantages. The solution supports most functional requirements very well and has great browser support. Due to its cumbersome implementation is is not recommended for image-heavy websites. Mutually exclusive media queries are needed for performance reasons. If the accessibility of images is important, a solution which uses the `<img>` tag to display images should be preferred. Other than that, it is a recommended solution for websites with few images.

Solution 5, JavaScript, is very flexible in terms of extensibility. Some features might be cumbersome to implement, but JS allows very powerful solutions. A downside is the rather poor performance as the browser's lookahead parser can not be used for most implementations. When using a smartphone with a slow network connection, the advantage of serving optimized images might outperform the absence of the lookahead parser. That said, JS solutions do have some drawbacks, but are sensible option anyway when very customized implementations are needed.

Solution 6, Native HTML5, can be considered the flagship solution of all responsive images approaches. It is a standardized and future-proof approach, which fully supports all evaluated use cases. Its main drawback was the lack of browser support for a long time. However, at the time of writing all browsers except Internet Explorer support at least parts of the specification. Using the picturefill polyfill, it is possible to enable full support for most browsers. Drawbacks when using the polyfill are that the lookahead parser can not be used and JS is required. If this is not an issue, the native HTML5 is strongly recommended. It is expected to be the main responsive images solution when widespread browser support is available.

All evaluated solutions have in common that they are somehow affected by lacking browser support or inconsistent implementations of features. As it has been outlined in Section 2.1, enhancing the web with its many stakeholders, users and devices is a hard task. A big issue is that all browser vendors decide for themselves if and when a feature is implemented. Furthermore, some vendors do not strictly follow specs of the W3C and the WHATWG. The results are many incompatibilities and inconsistent behavior. In order to encounter this issue, it is strongly recommended to do a lot of testing. This includes testing browsers as well as devices. Testing desktop browsers can easily be done by installing the latest version of all major browsers. Most of them provide developer tools, which allow to inspect the HTML, DOM, and network traffic. There are several methods for testing on mobile devices. The developer tools of some desktop browsers allow to emulate devices by changing the viewport resolution, its DPR, and even limit the network speed to simulate a mobile connection. A better but less convenient alternative is to use real devices for testing. If this is not an option, services like BrowserStack are a good alternative [126]. BrowserStack deploys hundreds of operating system and browser combinations which can be accessed remotely for testing purposes. [74, p. 85]

The following discusses alternatives to the evaluated solutions. First, the not selected CSS image-set method and the CSS content property approach are covered. The image-set() CSS function is well-supported but currently only supports one use case – serving CSS background images in multiple, DPR-optimized variants. The CSS content property is not recommended as it is a rather hacky approach and inevitably introduces the risk of loading two images per responsive image [93].

Another alternative is to deploy a separate mobile website, also called m-dot site. User agent detection can be used to redirect new visitors to the mobile version if they are using a mobile device. An advantage of this approach is that not only the appearance but also the code can be optimized for the target devices. In contrast to RWD that uses the same HTML document for all devices, not needed stylesheets or JS libraries can easily be omitted to reduce the total size of the page. An issue with separate mobile websites is the great device fragmentation. New devices and device types are being introduced frequently. Creating a separate website for every emerging device category is not sustainable, expensive, and extremely difficult to maintain. That said, there are use cases for separate mobile websites, but a responsive design is recommended in most cases. [1, p. 8], [35, p. 9], [127]

A fundamentally different alternative is to create a native smartphone app. Its advantages are that very customized and powerful solutions are possible. More information about the user and the environment are available through sensors and operating system APIs. On the other hand, native apps have several downsides as well. First and foremost, an app has to be installed by the user before it can be used. Users probably prefer a website to spontaneously look up some information. Secondly, the code of native apps is dependent on the operating system and might have to be implemented again to support additional systems. These two issues can be overcome by creating apps using HTML, CSS, and JS. This method is becoming increasingly popular and companies like Spotify and Netflix deploy this approach. It uses an app-embedded browser to display the a website, which is implemented with a look and feel similar to a truly native smartphone app. [127]

The following discusses the future prospects of responsive images. Already in 2011, Bruce Lawson and Anne van Kesteren reasoned about the actual need for responsive images. *"There's a school of thought that says everything will be 300ppi and networks will be fast enough, so this is really an intermediate problem until everyone starts using high-res graphics and all displays go from 150 to 300 [Note: PPI]. [...]"* [128]. They might be right that the Resolution-based selection use case is less of an issue in the long run. Many devices are shipped with HiDPI screens and mobile Internet connections are getting faster and more reliable. However, this change will take years, and even longer in less developed countries. Additionally there are several other responsive images use cases that would still be valid, eg. art direction. That said, the demand for responsive images is certainly present, but some use cases might slowly fade away in the long term.

Another topic of future developments are new image formats. A responsive image image format stores several image versions in a single file. The browser can load as much of the file as needed – more data results in a better image quality. However, this is only a long term solution. Yoav Weiss states that *"a new format will take too long to implement and deploy, and will have no fallback for older browsers."* [129]

4 Proof of Concept: Travel Website using Responsive Fullscreen Images

This section describes the implementation of a website with responsive fullscreen images. First, the idea and concept is outlined and the scope is set in Section 4.1. The motivation for using fullscreen images and their implementation is covered in Section 4.2. Following next, Section 4.3 describes the prototypical implementation of the web application. The purpose of this section is to demonstrate the implementation of a real-world example. The implementation and its findings are discussed in Section 4.4.

4.1 Introduction

The concept of the following web application was the initial stimulus for choosing the topic of this work. The idea is to create a website that helps the users to choose their next travel destination. This is done by displaying fullscreen images of possible destinations. The overall user interface is planned to be very simple and minimal. The focus is put on high quality images of beautiful locations. Only some concise facts about the destination are shown in a small box. The next destination can be requested by clicking or tapping on the current image. Displaying the images in fullscreen mode makes them more expressive [130, p. 156]. The intention is to arouse the user's wanderlust and provoke impulsive buying behavior for flight tickets. The final application is planned to display real-time flight fares from an airport near the user to the shown destination. For the booking process the user should be redirected to the website of the responsible travel agency or airline. This setup keeps the administration effort and complexity of the application to a minimum. However, it is out of the scope of this work to implement the entire application. Thus, Section 4.3 only covers the implementation of a prototype. The focus is put on responsive images together with fullscreen images.

4.2 Fullscreen Images

Firstly, let us set the scope of fullscreen images. There are several types of fullscreen. When referring to fullscreen images in a web-context, it usually means images which cover the entire viewport. The browser window might not be maximized and user interface elements like the address bar are still visible. That said, the word *screen* in fullscreen can be misleading, as the image does not necessarily cover the entire screen.

Another type of fullscreen is the browser's native fullscreen mode. Most browsers have this feature, which hides the user interface of the browser and expands the viewport to the size of the entire screen. This fullscreen mode can not only be enabled using the browser's user interface, but also using the JS Fullscreen API. The prototype of this section mainly focuses on fullscreen images in terms of the viewport, but also uses the Fullscreen API to let the user activate the fullscreen mode of the browser.

4.2.1 Motivation

The motivation for using fullscreen images are diverse. The bigger images are displayed, the more attention they gain. Very expressive websites are can be created using fullscreen images. This expressiveness is an easy method to arouse emotions in the user. The emotional component of design is very important for the user experience. The user's perception of a brand or website can greatly be influenced by linking it with positive emotions. Fullscreen images are not the only or best, but an easy way to arouse emotions. Less textual content is needed and the image can speak for itself. It might be easier to buy some beautiful and aesthetic photographs than to develop a content strategy and produce loads of content. [3, p. 8], [130, p. 156]

The motivation for using a fullscreen images application in this work is rather technical. Fullscreen images cover the entire viewport and almost the entire screen when the browser is maximized. If all common screens should be supported, images with resolutions of up to 3840×2160 pixels (Ultra HD) and even higher have to be served. Images at such resolutions easily have a filesize of 3 megabytes and more. As we learned earlier, a lot of data is wasted when high-resolution images are delivered to low(er)-resolution devices. This overhead is most evident for fullscreen images and thus, they greatly demonstrate the need for responsive images.

Beside the above responsive images issue, other issues arise when working with fullscreen images. Some of them are outlined in the following section and can partly be solved using the responsive images use cases.

4.2.2 Issues

The main issue with fullscreen images is that they have to fill the entire viewport. The earlier presented fluid images technique causes images to be displayed with the width of their parent element. This technique does only consider the width and not the height. Thus, it is only applicable for fullscreen images if the image and the screen have the same aspect ratio. Otherwise the image does not cover the entire parent element. The following example will help understanding. An image with an intrinsic aspect ratio of 4:3 (landscape) is displayed on a 9:16 (portrait) smartphone screen. The fluid images technique can be used to display the image with the width of the viewport, but because of the different aspect ratios there will be a large gap at the bottom of the viewport. The left smartphone in the below figure illustrates the issue. [131]

There are two methods to approach this issue. One is to ignore the image's aspect ratio and squeeze or stretch the image to fill the viewport. The result is that the image looks distorted, which is not what we want. The better option is to upscale the image until it cover the entire viewport while preserving the image's aspect ratio. This approach solves the issue of correctly displaying the image as fullscreen image, but introduces another issue. The upscaling causes parts of the image to grow outside of the viewport. These parts are being cropped and are not visible anymore. The issue is that important objects of the image could be cropped, which causes the image to lose its informative value. This is most evident for images with important objects near the edges. The issue can be encountered by repositioning the image so that the important objects are within the viewport. This does not reduce the size of the cropped area, but ensures that the invisible area does not contain essential information. Both presented issues can be encountered with the techniques presented in the next section. [131]

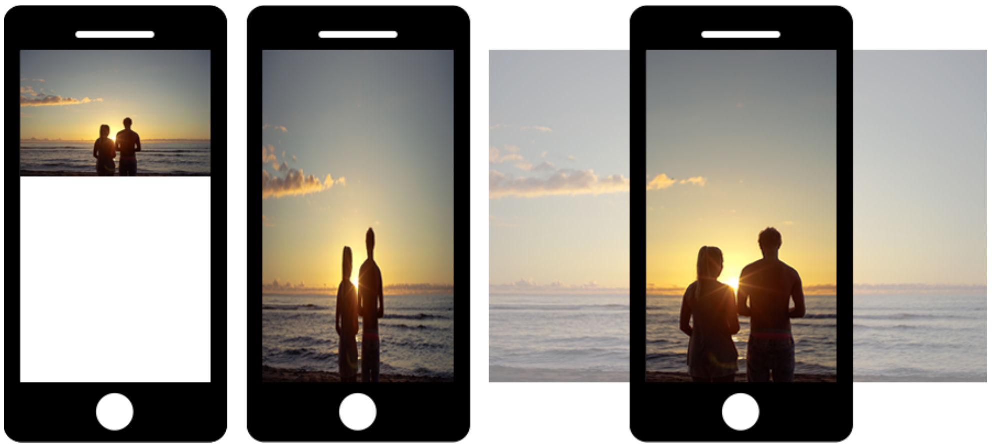

Figure 3: *Fluid image technique (left), scaling while ignoring the aspect ratio (middle), and correct scaling while preserving the aspect ratio, but with cropped areas.*

4.2.3 Approaches

There are several common approaches to implement fullscreen images. Technically, the goal is to display an image covering its entire parent element. For fullscreen images, this parent element happens to have the size of the viewport, but the goal remains the same. The following briefly covers three common approaches to implement fullscreen images.

CSS Background Images

CSS background images can easily be configured to cover the entire target element using the `background-size: cover;` declaration. This technique preserves the aspect ratio of the image. The next step is to display the element with the size of the viewport. This can be done by setting its width and height to 100%. Before this works, the width and height of the `<html>` and `<body>` element have to be set to 100% as well. The following listing shows how to implement a fullscreen image using a CSS background image. [132]

```
<div id="an-image"></div>

<style>
  html, body {
    width: 100%;
    height: 100%;
    margin: 0;
  }

  #an-image {
    position: absolute;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    background-image: url(http://placehold.it/1680x1050); /* placehold.it automatically
                                                             generates placeholder images
                                                             with the given resolution.    */
    background-size: cover;
    background-position: center center;
}
</style>
```

Listing 34: *Implementing a fullscreen background image using CSS background images.*

The background-size property can only be applied to CSS background images. Images that are displayed using the `<img>` tag are not supported. In consequence, this approach can not be combined with responsive images solutions that require the `<img>` tag, eg. the native HTML5 solution. However, a similar `<img>`-compatible property is introduced in the lastly covered approach.

The cropping of important objects issue can be encountered using the background-position property. It allows to adjust the positioning of the image within the element. The default value is `0% 0%`, which matches the top left corner of the target element with the top left corner of the image. The declaration `background-position: 50% 50%;` can be used to horizontally and vertically center the image within the element. [14, p. 245]

This approach offers very good browser support. It can very well be combined with the CSS Background Images responsive images solution. The same limitations as stated in the evaluation apply.

JS-based Approach

There are several methods to implement fullscreen images using JS. Some JS fullscreen image scripts utilize the before covered CSS Background Images approach. This is done by dynamically creating a new element and setting the background-image property to the image URL. The URL(s) can be passed to the script by a parameter or get fetched from attributes of a HTML tag. The implementation and result of the fullscreen image is the same as for the just covered CSS Background Images approach.

Another JS-based option is to implement fullscreen images using the `<img>` tag. This requires to emulate the native functionality of the `background-size: cover;` declaration using JS. The task is to manually set the width and height of the image to values which are sufficient to cover the entire the viewport. The aspect ratio of the image has to be preserved when calculating the new dimensions. The image can then be positioned absolutely to adjust which parts of the image are cropped and invisible. The following listing shows how to manually set the required CSS declarations. Please note that it is a static example just to demonstrate the calculation. The JS code of Section 4.3.5 shows how to automate the calculation and setting of the values. The values have to be recalculated every time the size of the viewport changes.

```
<img src="http://placehold.it/1024x768">

<style>
  html, body {
    width: 100%;
    height: 100%;
    margin: 0;
    overflow: hidden; /* do not show the parts of the img which overlap the body element   */
  }

  /* let us assume a 1024×768 image (4:3 aspect ratio) on a 1280×800 viewport (16:10)      */
  img {
    position: absolute;
    width: 1280px; /* the width should cover the entire viewport                          */
    height: 960px; /* the height is calculated using to the aspect ratio ( 1280 / 4 × 3)   */
    left: 0;       /* the width matches the viewport exactly – no offset needed           */
    top: -80px;    /* the image height (960) is higher than the viewport (800), an offset of
                      -80px ((960-800) / 2) results in the image being vertically centered */
  }
</style>
```

Listing 35: *Implementing a fullscreen image using the* `<img>` *tag and manual positioning using CSS.*

The cropping issue can be handled using the *top*, *right*, *bottom* and *left* CSS properties to position the image so that no important areas of the image are cropped. The jQuery plugin FocusPoint implements this functionality [133].

JS-based fullscreen images approaches have the same characteristics as responsive images JS approaches. They allow very flexible implementations and browser support is high. Some fullscreen image scripts use the CSS background image approach and fallback to the `<img>` element if the browser does not support the background-size property. A drawback is the dependency on JS. The positioning and resizing of images can be cumbersome to implement and might be slower than native browser functions. There are several ready-to-use scripts available that implement JS-based background images [134]–[137].

CSS object-fit Property

The object-fit CSS property is the `<img>`-compatible equivalent of the background-size property. The value *cover* ensures that the image always covers the entire size of the `<img>` element while preserving the image's aspect ratio. This approach makes the need to manually resize and position the image obsolete. Its compatibility with the `<img>` tag allows to use it with the native HTML5 responsive images solution, which results in a very powerful combination. [94, Sec. 4], [131]

```
<img src="http://placehold.it/1680x1050">

<style>
  html, body {
    width: 100%;
    height: 100%;
    margin: 0;
  }

  img {
    position: absolute;
    left: 0;
    top: 0;
    width: 100%;
    height: 100%;
    object-fit: cover;
  }
</style>
```

Listing 36: *Implementing a fullscreen image using the `<img>` tag and the object-fit CSS property.*

The cropping issue can be handled the same way as with CSS background images. The equivalent property is called object-position and can take the same arguments as background-position. However, object-position uses a default value of `50% 50%` (center), while background-position has a default value of `0% 0%` (top left).

The object-fit fullscreen approach is very powerful as it can be used with the `<img>` element. Thus, it can be combined with all responsive images solutions except the CSS Background Image responsive images solution. The object-fit approach together with the native HTML5 responsive images solution was selected for the implementation of the prototype.

A drawback of the object-fit approach is its rather poor support by older browsers [138]. No version of Microsoft's Internet Explorer does not support the object-fit property.

4.3 Implementation of a Prototype

It is out of the scope for this work to implement and document the entire application as described in Section 4.1. Thus, it was decided to implement only a prototype to demonstrate the concept. The focus is put on responsive images, and their combination with fullscreen images. The purpose is to demonstrate the implementation of responsive images using a real-world example. The rapid prototyping strategy is used to obtain results quickly. The strategy's aim is to deliver a working application as soon as possible. This is done using very new techniques which allow an easy implementation. Some of these techniques are not yet supported in all major browsers. In this case, suggestions for workarounds are given. However, due to applied rapid prototyping strategy, the code is not intended for use on a production site. More work will be needed to add support for all browsers and devices.

4.3.1 Overview

The following gives an overview of the implementation. It was tried to keep the application's architecture as simple as possible. HTML, CSS, and JS were used on the client-side, and PHP for the server-sided logic. The application consists of only one HTML document, which is dynamically changed using JS. This HTML document contains the homepage, which is delivered to the user on the first visit. The homepage already contains a fullscreen image together with the text “We will help you finding your next travel destination.”, which should arouse the user's wanderlust.

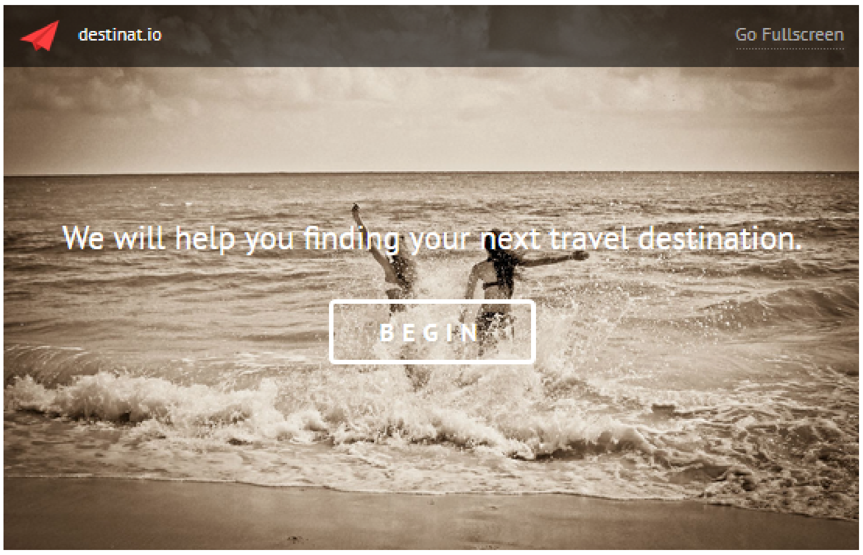

Figure 4: *Screenshot of the homepage of the prototype.*

After a click on the “Begin” button the actual presentation of destinations starts. The destinations are loaded dynamically using AJAX. A server-sided PHP script randomly selects one destination from an associated array. The destination is returned to the client as a JSON-encoded object. The client-sided JS code initiates the image preloading. Once the image has been loaded, it is displayed and the location information are updated. The location information is shown in a semi-transparent box, which changes its size according to the size of the viewport. The images are implemented using the native HTML5 responsive images solution. The object-fit CSS property is used to display the images in fullscreen mode. A JS-based fullscreen approach is used as a fallback if the object-fit property is not supported.

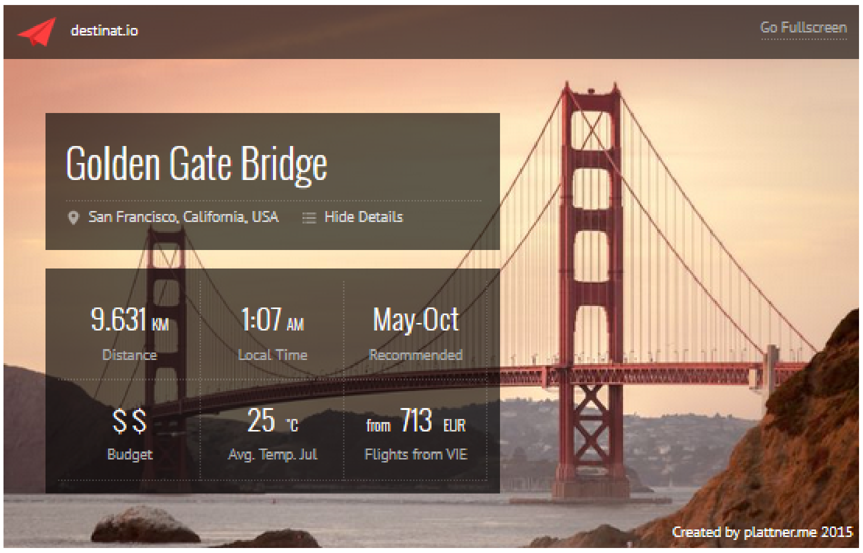

Figure 5: *Screenshot of the main view of the prototype on a desktop monitor.*

The prototype of this work is shipped with 22 sample destinations and 7 homepage images. The images were downloaded from free stock images websites [139], [140]. Six individual data fields were chosen to be displayed in the information box. The distance to the destination, the local time at the destination, the recommended months for a trip, the needed budget, the average temperatures of the current month, and most importantly, the current flight fares. As this prototype's purpose is just to demonstrate the concept, the displayed data might not be correct. Random values are use for some fields like the average temperature or the flight fares.

It is out of the scope for this work to document every detail of the prototype. The following describes some relevant aspects of the implementation. Please note that some less relevant parts of the code listings have been omitted for didactic reasons.

4.3.2 File Structure

Knowing the file structure of the application is helpful to get an overview of the architecture. The following table briefly describes the relevant files and directories.

File or Directory	Sources	Description
css/style.css	–	The CSS Stylesheet which contains all custom styling and media queries to adapt the design to small devices.
img/	[139]–[141]	This directory holds UI graphics (logo and icons), and the destinations/ and homepage/ directories.
img/destinations		This directory contains high-resolution images of the destinations. The actually served downscaled versions are located in the scaled/ subdirectory.
img/homepage		This directory contains high-resolution images for the homepage (first page). The actually served downscaled versions are located in the scaled/ subdirectory.
js/common.js	–	The main JS file which contains the client-sided logic.
js/picturefill.js	[123]	Polyfill for the native HTML5 responsive images solution.
js/modernizr.custom.js	[142]	Modernizr is a script to perform feature detections. It is used to detect object-fit, srcset attribute, and fullscreen API support.
js/hammer.js	[143]	Hammer.JS is a library to implement touch gestures. It is used to allow the user to request the next destination by swiping to the left on touch devices.
js/jquery.min.js	[144]	Google's jQuery library, used for selecting elements, DOM manipulations, and AJAX requests.
destination_data.php	–	This PHP file holds the destination data (title, location, image filename etc.) in an associative array.
get_image.php	–	This PHP script is called using AJAX. It randomly selects a destination from the above file, formats the data, and returns it to the client as a JSON object.
index.php	–	This PHP file mainly contains the HTML structure of the website. It also includes a short piece of PHP code to randomly select an image for the homepage.

Table 27: *File structure of the prototype.*

The three main files of the applications are the index.php HTML document, the style.css stylesheet, and the common.js JS file. The following listing shows the basic structure of the index.php file. The inner content of the main div containers has been omitted due to a lack of space.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>destinat.io</title>
    <link href="css/style.css" rel="stylesheet">
    <!-- some meta tags and external fonts have been ommited in this listing. -->
  </head>
  <body>
    <div class="welcome">
      <!-- "We will help you finding your next travel destination" message and button. -->
    </div>
    <header class="main">
      <!-- top bar with the logo, title etc.
    </header>
    <div class="fullscreen-images">
      <!-- container for the fullscreen images -->
    </div>
    <div class="information-box">
      <!-- container of the information about the destination. -->
    </div>
    <div class="loading">
      <!-- a semi-transparent fullscreen overlay which serves as a loading indicator -->
    </div>
    <script src="js/jquery.min.js"></script>
    <script src="js/picturefill.js"></script>
    <script src="js/modernizr.custom.js"></script>
    <script src="js/hammer.js"></script>
    <script src="js/common.js"></script>
  </body>
</html>
```

Listing 37: *The basic HTML structure of the prototype. (index.php)*

The following listing shows the structure of the destination_data.php file, which stores the destination data in an associative array.

```
<?php
$destinations = array();
$destinations[] = array(
  'title' => 'Siena',                 // title of the destination
  'location' => 'Tuscany, Italy',     // location of the destination
  'distance' => '670',                // distance, hardcoded from Vienna for the prototype
  'timezone' => 0,                    // timezone as offset from Europe/Vienna timezone
  'recommended_months' => 'Jun-Sep',  // recommended travel months for the destination
  'average_temperature' => mt_rand(25,38), // randomly set the average monthly temperature
  'flight_fare' => mt_rand(150, 300), // randomly set the flight fare
  'image_file' => '7',                // the number of the destination's image file
  'position_x' => 85,                 // the horizontal focus point of the image
  'position_y' => 50                  // the vertical focus point of the image
);
// more destinations follow here
```

Listing 38: *The associative array which holds the destination data. (destination_ data.php)*

The common.js file is less straightforward to explain and is covered in small parts over the next sections.

4.3.3 Responsive Fullscreen Images

This section covers the implementation of responsive images and their combination with fullscreen images. The images of the destinations are implemented using the native HTML5 responsive images solution. The srcset attribute is used to implement a resolution-based selection. The provided widths are 360, 720, 960, 1280, 1650, and 1920 pixels. Higher resolutions are recommended but the used source images were only available with up to 1920 pixels in width. The following listing shows how the images are implemented in the HTML code.

```
<img class           = "fullscreen-image current"
     sizes           = "100vw"
     data-position-x = "0.2"
     data-position-y = "0.5"
     srcset          = "img/destinations/scaled/1-360.jpg 360w,
                        img/destinations/scaled/1-720.jpg 720w,
                        img/destinations/scaled/1-960.jpg 960w,
                        img/destinations/scaled/1-1280.jpg 1280w,
                        img/destinations/scaled/1-1680.jpg 1680w,
                        img/destinations/scaled/1-1920.jpg 1920w">
```

Listing 39: *Prototype's implementation of responsive images using the srcset attribute.*

No src attribute is provided for the `<img>` tag. However, this does not cause unexpected behavior as stated in Section 3.4.6. Browsers that support the native HTML5 solution do not need a src attribute. For unsupported browers, the picturefill polyfill ensures that the src attribute is set before the element is added to the DOM. The sizes attribute is set to 100vw (viewport width) as the images are meant to be display in fullscreen mode and thus, with 100% width of the viewport. The data-position-x and data-position-y attributes contain the focus point of the image to correctly position the image on the viewport. These two attributes are used by the fallback in Section 4.3.5. The srcset attribute contains six image resources with a width descriptor for the provided widths.

Now that we have implemented the responsive images, we will next cover how to display them in fullscreen mode. The CSS object-fit property approach was selected to display the images in fullscreen mode. The following CSS rule is applied to all fullscreen images on the page.

```
.fullscreen-image {
  position: absolute;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  object-fit: cover;
  z-index: 100;
}
```

Listing 40: *Prototype's implementation of fullscreen images using the object-fit property.*

Section 4.2.2 introduced the issue of losing important parts of the image when it is cropped to fit the viewport. This issue can be encountered by controlling the cropping process. The object-position property can be used to specify how the image should be positioned. The default value is `50% 50%`, which horizontally and vertically centers the image. However, if the main object of the image is not within the center region, the value can be adjusted. For example, if the main object is on the top right corner, the value to apply is `100% 0%`. For the prototype, the focus point of each image can be set in the associative array of destination_data.php using the indices `position_x` and `position_y`. The following listing shows how the object-position property is set using JS. This is done just before the next image is displayed.

```
// check if both focus points have been set
if (data.position_x && data.position_y) {
  // apply the object-position property using jQuery's css() function
  $nextImage.css('object-position', data.position_x + '% ' + data.position_y + '%');
}
```

Listing 41: *Positioning the image to using object-position to control which areas are cropped.*

The below figure shows how object-position controls the positioning and thus, the cropping process. The image of the Italian destination Siena shows a tower on the right side of the image. The object-position property is set to `85% 50%` to ensure that this tower is always within the viewport regardless of the device's width.

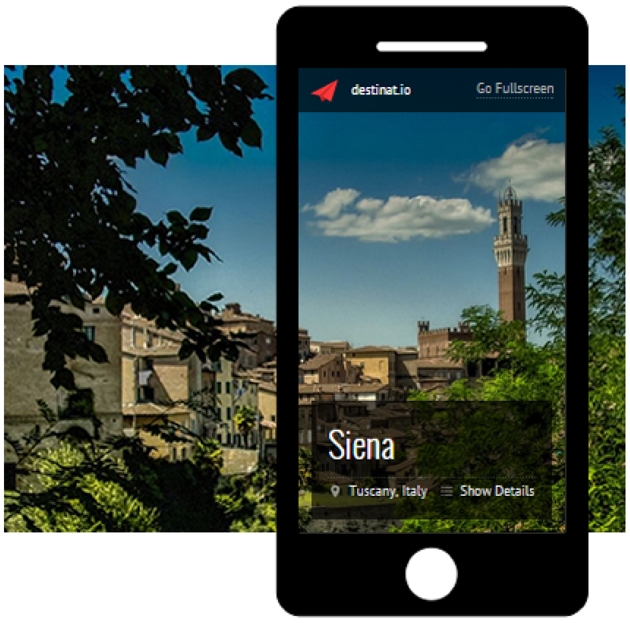

Figure 6: *The result of using object-position to control the cropping process.*

Not all browsers support the object-fit and object-position properties [138]. Safari only supports the object-fit property, but not object-position. The object-position property is the only method to control the cropping process when the object-fit property is used to display images in fullscreen mode. That said, if the cropping has to be controlled, the object-fit fullscreen approach is not suitable for browsers without object-position support. A workaround is to fallback to a JS-based fullscreen approach and use a plugin like FocusPoint [133] to adjust the cropping process. The prototype does provide a fallback for object-fit and object-position. Its implementation is covered in Section 4.3.5.

The dynamic switching between destinations is implemented using JS. A click anywhere within the viewport area initiates the loading of the next destination. Touch and keyboard control is supported as well and is covered in Section 4.3.4. This is done by binding the respective input event handlers to the `<body>` element. These event handlers execute the loadNextImage function. The function sends an HTTP GET request to the get_image.php script using AJAX. The get_image.php script randomly selects one destination from the `$destinations` array of the destination_data.php file. The selected destination data is sent back to the client as a JSON-encoded object. The image_tag property of this object contains the ready-to-use HTML code of the destination's `<img>` element. The tag is put together by the get_image.php script. jQuery is used on the client to append the new `<img>` tag to the `<div class="fullscreen-images">` container using the appendTo function. The following listing shows the loadNextImage function. Please note that less important parts of the function have been omitted due to a lack of space.

```
function loadNextImage() {
  var $currentImage = $('.current');
  $.getJSON('get_image.php', function(data){
    $nextImage = $(data.image_tag).hide();
    $nextImage.one('load', function() {
      $nextImage.appendTo('.fullscreen-images').show().addClass('current');
      $currentImage.remove();
    }
  }
}
```

Listing 42: *Conceptual working of the loadNextImage function to load and display the next destination.*

4.3.4 Responsive User Interface

This section covers the implementation of the responsive user interface. It is not within the scope of this work to explain every step in great detail. The header bar on the top of the page is implemented using a percentage width of 100%. This causes the bar to adjust to the screen size automatically. Other than that, the destination information box is the main element which needs adaption to the screen size. The following listing shows how this is done using CSS media queries. Please note that some less important declarations have been omitted.

```
@media screen and (max-width: 420px) {
  .information-box {
    width: 90%;   /* change the width of the entire information box to 90% of the viewport */
    left: 5vw;    /* set the left and bottom distance from the edges to 5% of the viewport */
    bottom : 5vw; /* width (90% box + 2 × 5% left and (implicitly) right margin = 100%)
  }
  .single-info-box {
    width: 50%;   /* change the width of individual each info box from 33.3% to 50% to
                     display two boxes next to each other instead of three.             */
  }
}
```

Listing 43: *CSS media query to adapt the information box to narrow viewports.*

The above CSS media queries and declarations cause the information box to only display two information fields side by side. As the below figure shows, the result is an optimized view for narrow devices like smartphones.

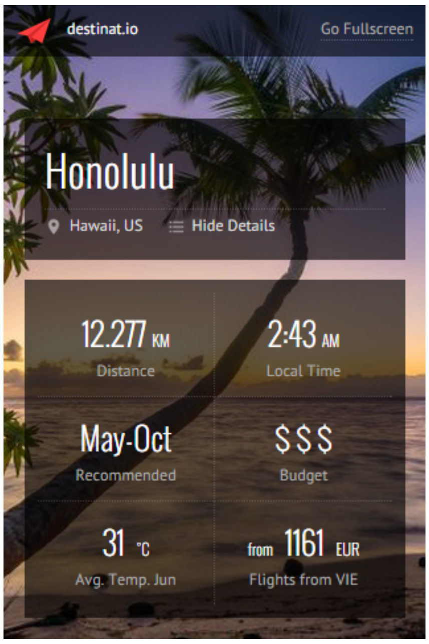

Figure 7: *Screenshot of the main view of the prototype on a narrow viewport, eg. a smartphone.*

As we learned in Section 2.4, being responsive is not only about adapting the website to the screen of the device. It is about taking advantage of other device capabilities as well. The prototype supports various input types to fulfill this issue. The request of the next destination can be triggered by clicking with the mouse, pressing the right arrow key, or swiping to the left on devices with a touch screen.

The following listing shows how to implement each of the three methods using jQuery, and the Hammer.JS library for the touch event.

```
// bind events handlers to the click and keydown events
$(document).on('click', 'body', function() {
  loadNextImage(); // call loadNextImage() when the user clicks somewhere within the <body>
}).keydown(function(event) {
  if (event.which == 39) {
    loadNextImage(); // call loadNextImage() when the user hits the right arrow key
  }
});

// bind an event handler to the swipeleft event provided by the Hammer.JS library
var hammer = new Hammer(document.body);
  hammer.on("swipeleft", function(event) {
  loadNextImage(); // call loadNextImage() when the user swipes to the left on a touch device
});
```

Listing 44: *Binding click, keydown and swipeleft events to the* `<body>` *element to initiate the loading of the next image.*

Another feature to increase the user experience is the Fullscreen API [24]. It gives the fullscreen images even more space by displaying the website in real fullscreen mode, without the browser's user interface. This is especially interesting

```
$('.go-fullscreen').on('click', function(event){
  var element = document.documentElement;
  if(element.requestFullscreen) {
    element.requestFullscreen();
  } else if(element.mozRequestFullScreen) {
    element.mozRequestFullScreen();
  } else if(element.webkitRequestFullscreen) {
    element.webkitRequestFullscreen();
  } else if(element.msRequestFullscreen) {
    element.msRequestFullscreen();
  }
  event.stopPropagation();
  return false;
});
```

Listing 45: *Using the Fullscreen API with the prefixed functions of all major browser vendors.*

for our fullscreen images website. The following listing shows how to implement the Fullscreen API using all vendor prefixes for best compatibility [145]. The target element is the `document.documentElement` DOM property, which represents the `<html>` element.

4.3.5 Fallbacks and Workarounds

All workarounds and fallbacks are implemented with the support of Modernizr [142]. Modernizr is a JS library to detect a wide range of device and browser features. Feature detection can be used to progressively enhance a website or disable features which require unsupported browser functions. The prototype uses Modernizr to detect support for the srcset attribute, the object-fit property, and the fullscreen API. The results of the detection can be accessed using the `window.Modernizr` object. The object contains one property which is either true (supported) or false (unsupported) for every tested feature. Modernizr additionally adds CSS classes to the `<html>` element to indicate the feature detection results. For example, the class *srcset* is added if the srcset attribute is supported, and *no-srcset* otherwise.

The srcset attribute detection is needed for the picturefill polyfill. Picturefill calls itself several times during the page loading phase and on a resize of the browser window to (re-)evaluate all `<img>` and `<picture>` tags. However, the images of the prototype are injected dynamically into the DOM. This requires picturefill to be called manually after the injection. Modernizr is used to only call picturefill if no srcset support is available.

```
if ( ! Modernizr['srcset']) {
  picturefill({ elements: $nextImage.get() }); // call picturefill() for the next image.
}
```

Listing 46: *Using Modernizr to conditionally call the picturefill function if no srcset attribute support was detected.*

The detection of the object-fit property is needed to fallback to a JS-based fullscreen images approach if the property is not supported. In that case, the resizing and positioning of the fullscreen image has to be done manually. The following listing shows the custom resize function that emulates the `object-fit: cover;` functionality. The code is based on the jQuery Backstrech plugin, a fullscreen images script [136].

```
if ( ! Modernizr['object-fit']) {
  resize($nextImage); // call the resize function only if the object-fit is not supported.
}

function resize($imgElement) {
  // get the viewport width and height and calculate the image's aspect ratio.
  var imageCSS = {left: 0, top: 0}
    , viewportWidth = $(window).width()
    , imageWidth = viewportWidth
    , viewportHeight = $(window).height()
    , imageAspectRatio = $imgElement.prop('naturalWidth') / $imgElement.prop('naturalHeight')
    , imageHeight = imageWidth / imageAspectRatio
    , imageOffset;

  if (imageHeight >= viewportHeight) {
    // the image's aspect ratio is smaller than the viewport's aspect ratio
    imageOffset = (imageHeight - viewportHeight) / 2; // calculate the offset to center the
    imageCSS.top = '-' + imageOffset + 'px';          // image vertically
  }
  else {
    // the image's aspect ratio is bigger than the viewport's aspect ratio
    imageHeight = viewportHeight;                   // set the height to the viewport height
    imageWidth = imageHeight * imageAspectRatio;    // calculate width using the aspect ratio
    imageOffset = (imageWidth - viewportWidth) / 2; // calculate the offset to center the
    imageCSS.left = '-' + imageOffset + 'px';       // image horizontally
  }

  imageCSS.width = imageWidth;   // add the correct width to the css object
  imageCSS.height = imageHeight; // add the correct height to the css object
  $imgElement.css(imageCSS);     // apply the width, height, top, and left properties
}                                // to the image
```

Listing 47: *Set the with and height of an image to cover the viewport.*

The detection of the Fullscreen API is used to hide the “Go Fullscreen” link in browsers without Fullscreen API support [146]. This is done using the CSS classes that Modernizr adds to the `<html>` element. The following listing shows the CSS rule to hide the “Go Fullscreen” link in unsupported browsers.

```
html.no-fullscreen .go-fullscreen-link {
  // this rule is only applied if Modernizr added the no-fullscreen class to the <html> tag
  display: none;
}
```

Listing 48: *Hide the “Go Fullscreen” link based on the Modernizr feature detection result.*

4.4 Discussion

The created prototype demonstrates how responsive images can be implemented and combined with fullscreen images using a real-world example. The purpose was to deepen the understanding of responsive images and their application. Another aim was to present workaround and fallback strategies. The Modernizr script is recommended to detect browser support for certain features. Using Modernizr, a workaround for the object-fit property was implemented and tested in several browsers. It could be shown that the native HTML5 responsive images solution is ready for production use. Older and unsupported browsers can easily be supported using the picturefill polyfill. A drawback of the polyfill is the dependency on JS. This issue can be encountered by serving the `<img>` tags with a default image in the src attribute. The default images is then used for also non-JS users. However, the default image might be preloaded by JS-enabled browsers and then be replaced by another image version using JS. That said, even though the native HTML5 responsive images solution is highly recommended, the implementation details are always a tradeoff between support and performance. The ideal decision does vary based on the requirements of every individual project.

As stated at the beginning of Section 4.3, it was out of the scope of this work to implement the entire application. The following presents the limitations of the prototype and suggests ideas for future improvement.

An issue with the prototype is that the used srcset attribute only considers the width of the image. Images are selected considering the width descriptors. The descriptor values are matched with the calculated value of the sizes attribute, which is 100vw for fullscreen images. However, the aspect ratio of the image and the screen may greatly vary. Most used images have a widescreen aspect ratio, eg. 16:9, but a user's device might be in portrait mode with an aspect ratio of 9:16. The result is that the image has to be upscaled by $1.\overline{7}$ (=16/9) on the client. Figure 3 (p. 87) illustrates the issue. The result is a great reduction of the image quality and thus, a poor user experience. This issue can be encountered using the `<picture>` element and deliver pre-cropped images based on the aspect ratio using art direction.

Another recommendation for future enhancements of the prototype is to provide more image versions. More than the six widths of the prototype should be supplied. Resolutions of up to 3840 pixels in width are needed to provide high quality images to Ultra HD screens. The images can additionally be served in several image formats, eg. WebP.

The prototype was tested in all major browsers, several mobile devices, and tablets. However, more testing is highly recommended. For example, an issue was encountered when testing iPhones running iOS version 8. The Safari browser of the given phone supports the srcset attribute with the pixel density descriptor, but not the width descriptor. A bug in Safari causes the width descriptors to be evaluated without considering the DPR, which results in the wrong image being selected [147]. Repeatedly calling the picturefill function of the picturefill polyfill after the page has loaded does solve this issue. This bug shows that many devices, including older ones, have to be tested before launching a website. More incompatibilities are likely to be encountered and can be fixed using workarounds. More on testing responsive websites is covered in Section 3.5.2.

5 Conclusion

The RWD technique has helped web developers to create websites that adapt to the device used to access them. However, RWD does not sufficiently address images. The same image desktop-optimized files are delivered to all devices. Responsive images approaches address this issue by delivering properly dimensioned images for every device to reduce the overall page size. This work evaluated six common responsive images solutions according to a created evaluation framework. The purpose was to provide a guideline and support software architects and developers in their decision making when planning web projects.

The following briefly outlines the findings and their implications. It was shown that the current responsive images issues are mostly solved. The results clearly depict that the native HTML5 solution is the most mature evaluated solution. The solution supports all functional requirements and performs very well for the non-functional requirements. None of the evaluated system prerequisites are needed in order to use the native HTML5 solution. The lack of browser support can easily be overcome with the picturefill polyfill. However, the polyfill requires JS to be available. The solution is standardized and future-proof. Thus, the native HTML5 solution is strongly recommended for most projects.

The following proposes a guideline for the selection of a responsive images approach. As stated above, the native HTML5 solution should be examined first. In the rare cases where it is not sufficient – likely due to a lack of integrability or a lack of browser support – other solutions should be considered as well. The evaluation in Section 3.3 and its results and discussion in Section 3.5 provide a thorough overview of common solutions. The HTTP Client Hints solution was found to be a highly integrable approach and is recommended for legacy applications once it is widely supported. The CSS Background Images solution is a sensible approach for websites with only a few images that need to be displayed in fullscreen mode.

The creation of the prototype of a travel website using fullscreen images demonstrated the implementation of a real-world example. It showed that fallbacks can be provided for many browser incompatibilities using feature detection, polyfills, and custom workarounds. However, a lot of testing with a wide range of different devices is highly recommended. The inconsistent behavior of many browsers and devices causes unexpected results, which can only be detected by testing.

The future prospects of responsive images are encouraging. The native HTML5 solution is expected to be supported by all major browsers shortly. This will take longer for HTTP Client Hints, but it will then offer a valuable addition. The image-set CSS property is planned to be enhanced, which could prove useful. More research needs to be done on responsive image formats. Their idea is to store multiple image versions in a single file and let the browser load only the parts needed.

References

[1] E. Marcotte, *Responsive Web Design*, 2nd ed. New York: A Book Apart, 2014.

[2] A. Brisbane, “Speakers Give Sound Advice,” *The Post-Standard*, New York, 28-Mar-1911.

[3] S. P. Anderson, *Seductive Interaction Design: Creating Playful, Fun, and Effective User Experiences*. Berkeley, CA: New Riders, 2011.

[4] “Interesting Stats,” *HTTP Archive*. [Online]. Available: http://httparchive.org/interesting.php. [Accessed: 03-Aug-2015].

[5] T. Kadlec, “Why we need responsive images,” *Tim Kadlec’s Blog*, 11-Jun-2013. [Online]. Available: http://timkadlec.com/2013/06/why-we-need-responsive-images/. [Accessed: 06-Jan-2015].

[6] W3C Working Group, “The history of the Web,” *W3C Wiki*. [Online]. Available: http://www.w3.org/wiki/The_history_of_the_Web. [Accessed: 17-Mar-2015].

[7] I. Hickson, “Interview with Ian Hickson, HTML editor,” *HTML5 Doctor*, 08-Jan-2013. [Online]. Available: http://html5doctor.com/interview-with-ian-hickson-html-editor/. [Accessed: 26-Feb-2015].

[8] Apple, “Configuring the Viewport,” *Apple Developer*, 01-Oct-2007. [Online]. Available: https://developer.apple.com/library/mac/documentation/AppleApplications/Reference/SafariWebContent/UsingtheViewport/UsingtheViewport.html. [Accessed: 18-Mar-2015].

[9] W3C Working Group, “The web standards model - HTML, CSS and JavaScript,” *W3C Wiki*, 2012. [Online]. Available: http://www.w3.org/community/webed/wiki/The_web_standards_model_-_HTML_CSS_and_JavaScript. [Accessed: 23-Mar-2015].

[10] D. Shea, “CSS Zen Garden: The Beauty of CSS Design,” 2003. [Online]. Available: http://www.csszengarden.com/. [Accessed: 23-Mar-2015].

[11] WHATWG, “HTML Living Standard,” 2015. [Online]. Available: https://html.spec.whatwg.org/multipage/. [Accessed: 28-Feb-2015].

[12] W3C Working Group, “HTML5 W3C Recommendation,” 28-Oct-2014. [Online]. Available: http://www.w3.org/TR/html5/. [Accessed: 23-Mar-2015].

[13] M. MacDonald, *HTML5: The Missing Manual*. O’Reilly Media, 2013.

[14] D. S. McFarland, *CSS3: The Missing Manual*, 3rd ed. Sebastopol, CA: O’Reilly Media, 2013.

[15] T. Atkins, H. W. Lie, and E. J. Etemad, “CSS Values and Units Module Level 3,” 30-Jul-2013. [Online]. Available: http://www.w3.org/TR/2013/CR-css3-values-20130730/. [Accessed: 24-Jul-2015].

[16] C. Zapponi, “Programming Languages and GitHub,” *GitHut*, 2015. [Online]. Available: http://githut.info/. [Accessed: 21-Apr-2015].

[17] D. Flanagan, *JavaScript: The Definitive Guide*, 6th ed. Beijing ; Sebastopol, CA: O’Reilly Media, 2011.

[18] A. Freeman, *Pro jQuery 2.0*, 2 edition. Apress, 2013.

[19] P. Gasston, *The modern Web: multi-device Web development with HTML5, CSS3, and JavaScript*. San Francisco: No Starch Press, 2013.

[20] A. Popescu, "Geolocation API Specification," 24-Oct-2013. [Online]. Available: http://www.w3.org/TR/2013/REC-geolocation-API-20131024/. [Accessed: 15-Jul-2015].

[21] A. Kostiainen and M. Lamouri, "Battery Status API Specification," 09-Dec-2014. [Online]. Available: http://www.w3.org/TR/2014/CR-battery-status-20141209/. [Accessed: 15-Jul-2015].

[22] W3C Working Group, "Device APIs Working Group - W3C," 2015. [Online]. Available: http://www.w3.org/2009/dap/. [Accessed: 15-Jul-2015].

[23] M. Lamouri and M. Cáceres, "The Screen Orientation API Specification," 28-Apr-2015. [Online]. Available: http://www.w3.org/TR/2015/WD-screen-orientation-20150428/. [Accessed: 15-Jul-2015].

[24] WHATWG, "Fullscreen API Standard," 27-Jul-2015. [Online]. Available: https://fullscreen.spec.whatwg.org/. [Accessed: 07-Aug-2015].

[25] "Why do people disable JavaScript?," *Stackexchange*, 14-Dec-2010. [Online]. Available: http://programmers.stackexchange.com/a/26186. [Accessed: 04-Jul-2015].

[26] A. Gustafson, "Understanding Progressive Enhancement," *A List Apart*, 07-Oct-2008. [Online]. Available: http://alistapart.com/article/understandingprogressiveenhancement. [Accessed: 04-Jul-2015].

[27] R. Fielding and J. Reschke, "RFC 7231 - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content," *Internet Engineering Task Force*, 2014. [Online]. Available: http://tools.ietf.org/html/rfc7231#section-5.3. [Accessed: 13-Jul-2015].

[28] A. Barth, "RFC 6265 - HTTP State Management Mechanism," *Internet Engineering Task Force*, 2011. [Online]. Available: http://tools.ietf.org/html/rfc6265. [Accessed: 29-Jul-2015].

[29] I. Grigorik, "HTTP caching — Web Fundamentals," *Google Developers*, 01-Jan-2014. [Online]. Available: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching#defining-optimal-cache-control-policy. [Accessed: 08-Aug-2015].

[30] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1," *Internet Engineering Task Force*, 1999. [Online]. Available: http://tools.ietf.org/html/rfc2616. [Accessed: 29-Jul-2015].

[31] "StatCounter Global Stats," 2015. [Online]. Available: http://gs.statcounter.com/. [Accessed: 25-Apr-2015].

[32] "Browser Trends April 2015: StatCounter vs NetMarketShare," *SitePoint*. .

[33] A. Deveria, "Can I use... Support tables for HTML5, CSS3, etc," *caniuse*, 2015. [Online]. Available: http://caniuse.com/. [Accessed: 18-Mar-2015].

[34] Mozilla Foundation, "Mozilla Developer Network," 2015. [Online]. Available: https://developer.mozilla.org/en-US/. [Accessed: 18-Mar-2015].

[35] T. Kadlec, *Implementing responsive design: building sites for an anywhere, everywhere web*. Berkeley, CA: New Riders, 2013.

[36] T. Garsiel and P. Irish, "How Browsers Work: Behind the scenes of modern web browsers," *HTML5 Rocks*, 05-Aug-2011. [Online]. Available: http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/. [Accessed: 06-Jan-2015].

[37] A. Davies, "How the Browser Pre-loader Makes Pages Load Faster," 22-Oct-2013. [Online]. Available: http://andydavies.me/blog/2013/10/22/how-the-browser-pre-loader-makes-pages-load-faster/. [Accessed: 25-Jun-2015].

[38] P. Thomakos, "How Javascript Loading Works - DOMContentLoaded and OnLoad," 14-Jun-2011. [Online]. Available: http://ablogaboutcode.com/2011/06/14/how-javascript-loading-works-domcontentloaded-and-onload/. [Accessed: 18-Jul-2015].

[39] P.-P. Koch, "A tale of two viewports — part two," *Quirksmode*, 2010. [Online]. Available: http://www.quirksmode.org/mobile/viewports2.html. [Accessed: 03-Jan-2015].

[40] P.-P. Koch, *The Mobile Viewports*. CSS Day Amsterdam, 2014.

[41] P.-P. Koch, "A tale of two viewports — part one," *Quirksmode*, 2010. [Online]. Available: http://www.quirksmode.org/mobile/viewports.html. [Accessed: 03-Jan-2015].

[42] G. Cummins, "Difference between visual viewport and layout viewport?," *Stackoverflow*, 13-Jun-2011. [Online]. Available: http://stackoverflow.com/a/6333966. [Accessed: 15-Jul-2015].

[43] P.-P. Koch, "A Pixel is not a Pixel is not a Pixel," *Quirksmode*. [Online]. Available: http://www.quirksmode.org/blog/archives/2010/04/a_pixel_is_not.html. [Accessed: 27-Feb-2015].

[44] P.-P. Koch, "Meta viewport," *Quirksmode*, 13-Apr-2014. [Online]. Available: http://www.quirksmode.org/mobile/metaviewport/. [Accessed: 24-Jul-2015].

[45] A. van Kesteren, "CSSOM View Module," *W3C*, 17-Dec-2013. [Online]. Available: http://www.w3.org/TR/cssom-view/. [Accessed: 24-Jul-2015].

[46] P.-P. Koch, "Desktop media query bugs 2: DPR and zoom level," *Quirksmode*, 03-Dec-2013. [Online]. Available: http://www.quirksmode.org/blog/archives/2013/12/desktop_media_q_1.html. [Accessed: 28-Jul-2015].

[47] OpenSignal, Inc., "Android Fragmentation Report August 2014," Aug-2014. [Online]. Available: http://opensignal.com/reports/2014/android-fragmentation/. [Accessed: 23-Apr-2015].

[48] Google, "The New Multi-Screen World: Understanding Cross-Platform Consumer Behavior." Google, 2012.

[49] P.-P. Koch, "screen.width is useless," *Quirksmode*, 13-Nov-2013. [Online]. Available: http://www.quirksmode.org/blog/archives/2013/11/screenwidth_is.html. [Accessed: 22-Jul-2015].

[50] J. Koch, "screen width is still useless," *Sevenval Blog*, 10-Mar-2015. [Online]. Available: http://blog.sevenval.com/4013/screen-width-is-still-useless/. [Accessed: 08-Aug-2015].

[51] "Gartner Says Worldwide PC Shipments Declined 5.2 Percent in First Quarter of 2015," 09-Apr-2015. [Online]. Available: http://www.gartner.com/newsroom/id/3026217. [Accessed: 23-Apr-2015].

[52] "Is Mobile Bringing About the Death of the PC? Not Exactly... - comScore, Inc." [Online]. Available: http://www.comscore.com/Insights/Blog/Is-Mobile-Bringing-About-the-Death-of-the-PC-Not-Exactly. [Accessed: 23-Apr-2015].

[53] "2 Billion Consumers Worldwide to Get Smart(phones) by 2016 - eMarketer," 11-Dec-2014. [Online]. Available: http://www.emarketer.com/Article/2-Billion-Consumers-Worldwide-Smartphones-by-2016/1011694. [Accessed: 25-Apr-2015].

[54] J. Stark, "The 10 principles of mobile interface design," 12-Apr-2012. [Online]. Available: http://www.creativebloq.com/mobile/10-principles-mobile-interface-design-4122910. [Accessed: 07-Jan-2015].

[55] D. Storey, "See your site like the rest of the world does. On the Nokia X2-01," 12-Sep-2012. [Online]. Available: http://generatedcontent.org/post/31441135779/mobileweb-row. [Accessed: 25-Apr-2015].

[56] Leichtman Research Group, "Internet-connected TVs in the US," 06-Jun-2014. [Online]. Available: http://www.leichtmanresearch.com/press/060614release.html. [Accessed: 25-Apr-2015].

[57] N. Leenheer, "How well does your browser support HTML5?," *HTML5test*, 2015. [Online]. Available: https://html5test.com/results/other.html. [Accessed: 25-Apr-2015].

[58] "Smart TV," *Wikipedia*, 25-Jul-2015. [Online]. Available: https://en.wikipedia.org/wiki/Smart_TV. [Accessed: 08-Aug-2015].

[59] J. Grigsby, "The Immobile Web," *Breaking Development Conference*, Apr-2012. [Online]. Available: https://vimeo.com/44036520. [Accessed: 25-Apr-2015].

[60] "Developing for TVs," *Dev.Opera*. [Online]. Available: https://dev.opera.com/tv/. [Accessed: 25-Apr-2015].

[61] A. Debenham, "Test of game console browsers," 2014. [Online]. Available: http://console.maban.co.uk/. [Accessed: 02-Aug-2015].

[62] "Smart watches shipments worldwide 2013-2015," *Statista*, 2014. [Online]. Available: http://www.statista.com/statistics/302722/smart-watches-shipments-worldwide/. [Accessed: 01-Aug-2015].

[63] E. Marcotte, "Responsive Web Design," *A List Apart*, 25-May-2010. [Online]. Available: http://alistapart.com/article/responsive-web-design. [Accessed: 01-Mar-2015].

[64] J. Grigsby, "Defining Responsiveness," *Cloud Four Blog*, 07-Jan-2014. [Online]. Available: http://blog.cloudfour.com/defining-responsiveness/. [Accessed: 11-Jan-2015].

[65] L. Danger Gardner, "What We Mean When We Say 'responsive,'" *A List Apart*, 06-Mar-2014. [Online]. Available: http://alistapart.com/column/what-we-mean-when-we-say-responsive. [Accessed: 11-Jan-2015].

[66] J. Zeldman, "Evolving Responsive Web Design," *Jeffrey Zeldman's Blog*, 09-Mar-2014. [Online]. Available: http://www.zeldman.com/2014/03/09/evolving-responsive-web-design/. [Accessed: 11-Mar-2015].

[67] J. Allsopp, "A Dao of Web Design," *A List Apart*, 07-Apr-2000. [Online]. Available: http://alistapart.com/article/dao. [Accessed: 10-Jan-2015].

[68] J. Nielsen, "Computer Screens Getting Bigger," *Nielsen Norman Group*, 07-May-2012. [Online]. Available: http://www.nngroup.com/articles/computer-screens-getting-bigger/. [Accessed: 11-Mar-2015].

[69] E. Bidelman, "'Mobifying' Your HTML5 Site," *HTML5 Rocks*, 03-Mar-2011. [Online]. Available: http://www.html5rocks.com/en/mobile/mobifying/. [Accessed: 09-Aug-2015].

[70] W3C Working Group, "W3C Recommendation: Media Queries," *W3C*, 19-Jun-2012. [Online]. Available: http://www.w3.org/TR/css3-mediaqueries/. [Accessed: 10-Mar-2015].

[71] M. Carver, *The Responsive Web*. Manning, 2015.

[72] T. Kadlec, "Beyond Responsive," *Tim Kadlec's Blog*, 07-Jan-2014. [Online]. Available: http://timkadlec.com/2014/01/beyond-responsive/. [Accessed: 13-Mar-2015].

[73] B. Frost, "Beyond Media Queries: Anatomy of an Adaptive Web Design," *Brad Frost's Blog*, 07-Aug-2012. [Online]. Available: http://bradfrost.com/blog/mobile/beyond-media-queries-anatomy-of-an-adaptive-web-design/. [Accessed: 13-Mar-2015].

[74] S. Jehl, *Responsible Responsive Design*. A Book Apart, 2014.

[75] M. Wilcox, *Adaptive Images*. 2012.

[76] W3C Working Group, "Images in HTML," *W3C Wiki*, 14-Mar-2014. [Online]. Available: http://www.w3.org/wiki/Images_in_HTML. [Accessed: 01-May-2015].

[77] I. Grigorik, "Image optimization — Web Fundamentals," *Google Developers*, 07-May-2014. [Online]. Available: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization. [Accessed: 08-Aug-2015].

[78] I. Grigorik, *High Performance Browser Networking: What every web developer should know about networking and web performance*, 1 edition. O'Reilly Media, 2013.

[79] W3C Working Group, "Use Cases and Requirements for Standardizing Responsive Images," *W3C*, 07-Nov-2013. [Online]. Available: http://www.w3.org/TR/respimg-usecases/. [Accessed: 01-Mar-2015].

[80] J. Nielsen, "Website Response Times," *Nielsen Norman Group*, 21-Jun-2010. [Online]. Available: http://www.nngroup.com/articles/website-response-times/. [Accessed: 08-Aug-2015].

[81] E. Portis, "Responsive Images in Practice," *A List Apart*, 04-Nov-2014. [Online]. Available: http://alistapart.com/article/responsive-images-in-practice. [Accessed: 11-Jan-2015].

[82] Y. Weiss, "Picture Element Implementation in Blink," *Indiegogo*, 20-Mar-2014. [Online]. Available: http://www.indiegogo.com/projects/661393/fblk. [Accessed: 01-Mar-2015].

[83] Microsoft, "picture element - Internet Explorer Web Platform Status and Roadmap," *modern.IE*, 13-May-2015. [Online]. Available: https://status.modern.ie/pictureelement. [Accessed: 13-May-2015].

[84] Y. Weiss, "Native Responsive Images," *Dev.Opera*, 19-Aug-2014. [Online]. Available: https://dev.opera.com/articles/native-responsive-images/. [Accessed: 27-Jul-2015].

[85] C. Coyier, “Which responsive images solution should you use?,” *CSS-Tricks*, 11-May-2012. [Online]. Available: https://css-tricks.com/which-responsive-images-solution-should-you-use/. [Accessed: 22-Jun-2015].

[86] D. Walsh, “Responsive Images: The Ultimate Guide,” *David Walsh’s Blog*, 16-Oct-2014. [Online]. Available: http://davidwalsh.name/responsive-images. [Accessed: 22-Jun-2015].

[87] K. Damball, “One Size Fits Some: A Guide to Responsive Design Image Solutions,” *Toptal Engineering Blog*, 05-Dec-2014. [Online]. Available: http://www.toptal.com/responsive-web/one-size-fits-some-an-examination-of-responsive-image-solutions. [Accessed: 22-Jun-2015].

[88] B. Smus, “High DPI Images for Variable Pixel Densities,” *HTML5 Rocks*, 22-Aug-2012. [Online]. Available: http://www.html5rocks.com/en/mobile/high-dpi/. [Accessed: 08-Aug-2015].

[89] Microsoft, “Internet Explorer Web Platform Status and Roadmap,” *modern.IE*, 2015. [Online]. Available: https://status.modern.ie/. [Accessed: 22-Jun-2015].

[90] “WebKit Bugzilla,” 2015. [Online]. Available: https://bugs.webkit.org/. [Accessed: 22-Jul-2015].

[91] Mozilla Foundation, “Mozilla Bugzilla,” 2015. [Online]. Available: https://bugzilla.mozilla.org/. [Accessed: 22-Jun-2015].

[92] J. Grigsby, “The Forgotten Responsive Images Spec: image-set(),” *Cloud Four Blog*, 27-Oct-2014. [Online]. Available: http://blog.cloudfour.com/the-forgotten-responsive-images-spec-image-set/. [Accessed: 22-Jun-2015].

[93] N. Gallagher, “Responsive images using CSS3,” *Nicolas Gallagher’s Blog*, 19-May-2011. [Online]. Available: http://nicolasgallagher.com/responsive-images-using-css3/. [Accessed: 22-Jun-2015].

[94] T. Atkins and Fantasai, “CSS Image Values and Replaced Content Module Level 3,” 29-May-2015. [Online]. Available: https://drafts.csswg.org/css-images-3/. [Accessed: 08-Aug-2015].

[95] “Most Common User Agents,” *Tech Blog*. [Online]. Available: https://techblog.willshouse.com/2012/01/03/most-common-user-agents/. [Accessed: 25-Jun-2015].

[96] ScientiaMobile, “WURFL - Mobile Device Database,” 13-May-2015. [Online]. Available: http://wurfl.sourceforge.net/. [Accessed: 13-May-2015].

[97] A. Andersen, “History of the browser user-agent string,” *WebAIM*, 03-Sep-2008. [Online]. Available: http://webaim.org/blog/user-agent-string-history/. [Accessed: 26-Jun-2015].

[98] Mozilla Developer Network, “Browser detection using the user agent,” *Mozilla Developer Network*, 10-Mar-2014. [Online]. Available: https://developer.mozilla.org/en-US/docs/Browser_detection_using_the_user_agent. [Accessed: 25-Jun-2015].

[99] M. Wilcox, “Adaptive Images in HTML.” [Online]. Available: http://adaptive-images.com/. [Accessed: 27-Jun-2015].

[100] Ş. Ghiţă, “Mobile Detect - lightweight PHP class for detecting mobile devices (including tablets),” 2015. [Online]. Available: http://mobiledetect.net/. [Accessed: 27-Jun-2015].

[101] Mozilla Developer Network, “Same-origin policy,” *Mozilla Developer Network*, 2015. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. [Accessed: 22-Jul-2015].

[102] P.-P. Koch, “Browser compatibility — viewports,” *Quirksmode*, 13-Nov-2013. [Online]. Available: http://www.quirksmode.org/mobile/tableViewport.html. [Accessed: 24-Jul-2015].

[103] Y. Weiss, “Preloaders, cookies and race conditions,” *Yoav Weiss’ Blog*, 28-Sep-2011. [Online]. Available: http://blog.yoav.ws/preloaders_cookies_and_race_conditions/. [Accessed: 08-Jul-2015].

[104] I. Grigorik, “HTTP Client Hints (Internet Draft),” *GitHub*, 18-Jun-2015. [Online]. Available: http://igrigorik.github.io/http-client-hints/. [Accessed: 14-Jul-2015].

[105] Y. Weiss, “Client Hints - Browser implementation considerations,” *GitHub*, 06-Jul-2015. [Online]. Available: https://github.com/yoavweiss/http-client-hints/blob/95fd1867c6ecaf475d59520bb52f32ce7ae6cc0b/browser_implementation_considerations.md. [Accessed: 14-Jul-2015].

[106] R. Mulhuijzen, “Best Practices for Using the Vary Header,” *Fastly - The Next Gen CDN*, 18-Aug-2014. [Online]. Available: https://www.fastly.com/blog/best-practices-for-using-the-vary-header/. [Accessed: 14-Jul-2015].

[107] Y. Weiss, “Client Hints 1,” *WebKit-dev Mailing List Archive*, 23-Apr-2015. [Online]. Available: https://lists.webkit.org/pipermail/webkit-dev/2015-April/027381.html. [Accessed: 14-Jul-2015].

[108] M. Stachowiak, “Client Hints 2,” *WebKit-dev Mailing List Archive*, 05-May-2015. [Online]. Available: https://lists.webkit.org/pipermail/webkit-dev/2015-May/027418.html. [Accessed: 14-Jul-2015].

[109] Y. Weiss, “Content-DPR header,” *WebKit-dev Mailing List Archive*, 29-May-2015. [Online]. Available: https://lists.webkit.org/pipermail/webkit-dev/2015-May/027469.html. [Accessed: 14-Jul-2015].

[110] Y. Weiss, “Bug 145380 – Add Content-DPR header support,” *WebKit Bugzilla*, 26-May-2015. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=145380. [Accessed: 14-Jul-2015].

[111] Y. Weiss, “Intent to Ship: DPR, Width, and Viewport-Width client hints–Google Groups,” 19-Jun-2015. [Online]. Available: https://groups.google.com/a/chromium.org/d/topic/blink-dev/vvX1vCQihDE/discussion. [Accessed: 14-Jul-2015].

[112] I. Grigorik, “Bug 935216 – Implement Client-Hints HTTP header,” *Mozilla Bugzilla*, 24-Jun-2015. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=935216. [Accessed: 14-Jul-2015].

[113] Microsoft, “HTTP Client Hints - Internet Explorer Status and Roadmap,” *modern.IE*, 14-Jul-2015. [Online]. Available: https://status.modern.ie/httpclienthints?term=client%20hints. [Accessed: 14-Jul-2015].

[114] A. Deveria, “CSS3 Media Queries browser support,” *caniuse*, 2015. [Online]. Available: http://caniuse.com/css-mediaqueries. [Accessed: 10-Mar-2015].

[115] D. Knight, “weblinc/media-match,” *GitHub*, 08-Jun-2013. [Online]. Available: https://github.com/weblinc/media-match. [Accessed: 22-Jul-2015].

[116] T. Kadlec, “Media Query & Asset Downloading Results,” *Tim Kadlec’s Blog*, 10-Apr-2012. [Online]. Available: http://timkadlec.com/2012/04/media-query-asset-downloading-results/. [Accessed: 19-Jul-2015].

[117] C. Coyier, “Retina Display Media Query,” *CSS-Tricks*, 14-Feb-2013. [Online]. Available: https://css-tricks.com/snippets/css/retina-display-media-query/. [Accessed: 23-Jul-2015].

[118] A. Deveria, “resolution media feature browser support,” *caniuse*, 2015. [Online]. Available: http://caniuse.com/css-media-resolution. [Accessed: 24-Jul-2015].

[119] L. Weintraub and E. A. Eklund, “LayoutUnit – WebKit,” 06-Apr-2013. [Online]. Available: http://trac.webkit.org/wiki/LayoutUnit. [Accessed: 24-Jul-2015].

[120] O. Saunders, “Using JavaScript to estimate connection speed,” *Blog / Decade City*, 08-May-2013. [Online]. Available: https://decadecity.net/blog/2013/05/08/using-javascript-to-estimate-connection-speed. [Accessed: 24-Jul-2015].

[121] N. C. Zakas, “Empty image src can destroy your site,” 30-Nov-2009. [Online]. Available: http://www.nczonline.net/blog/2009/11/30/empty-image-src-can-destroy-your-site/. [Accessed: 22-Jul-2015].

[122] M. Marquis, “srcN tests,” *GitHub*, 03-Oct-2013. [Online]. Available: http://wilto.github.io/srcn-polyfills/. [Accessed: 29-Jul-2015].

[123] S. Jehl, M. Marquis, M. Engel, A. Jegtnes, S. Forst, and A. Farkas, “Picturefill,” 2015. [Online]. Available: https://scottjehl.github.io/picturefill/. [Accessed: 29-Jul-2015].

[124] A. Deveria, “picture element browser support,” *caniuse*, 2015. [Online]. Available: http://caniuse.com/picture. [Accessed: 29-Jul-2015].

[125] A. Deveria, “srcset attribute browser support,” *caniuse*, 2015. [Online]. Available: http://caniuse.com/srcset. [Accessed: 29-Jul-2015].

[126] BrowserStack, “Cross Browser Testing Tool. 300+ Browsers, Mobile, Real IE.,” 2015. [Online]. Available: https://www.browserstack.com/. [Accessed: 28-Jul-2015].

[127] D. Klein, “How to decide: Mobile websites vs. mobile apps,” *Adobe Inspire Magazine*, 2012. [Online]. Available: http://www.adobe.com/inspire/2012/02/mobile-websites-vs-mobile-apps.html. [Accessed: 08-Aug-2015].

[128] B. Lawson, “Notes on Adaptive Images (yet again!),” *Bruce Lawson’s personal site*, 08-Dec-2011. [Online]. Available: http://www.brucelawson.co.uk/2011/notes-on-adaptive-images-yet-again/. [Accessed: 08-Aug-2015].

[129] Y. Weiss, “Responsive image format,” *Yoav Weiss’ Blog*, 07-May-2012. [Online]. Available: http://blog.yoav.ws/responsive_image_format/. [Accessed: 08-Aug-2015].

[130] T. Van Gorp, *Design for Emotion.* Waltham, MA: Morgan Kaufmann, 2012.

[131] C. Mills, “The CSS3 object-fit and object-position Properties,” *Dev.Opera*, 06-Jan-2011. [Online]. Available: https://dev.opera.com/articles/css3-object-fit-object-position/. [Accessed: 07-Aug-2015].

[132] J. Gube, “Responsive Full Background Image Using CSS,” *Six Revisions*, 30-Jun-2014.

[Online]. Available: http://sixrevisions.com/css/responsive-background-image/. [Accessed: 07-Aug-2015].

[133] J. Menz, "jquery-focuspoint plugin," *GitHub*. [Online]. Available: https://github.com/jonom/jquery-focuspoint. [Accessed: 31-Jul-2015].

[134] J. Salvat, "jquery-vegas plugin," *GitHub*, 28-Apr-2015. [Online]. Available: https://github.com/jaysalvat/vegas. [Accessed: 31-Jul-2015].

[135] A. Vanderzwan, "jquery-MaxImage plugin," *GitHub*, 2015. [Online]. Available: https://github.com/akv2/MaxImage. [Accessed: 31-Jul-2015].

[136] S. Robbin, "jquery-backstretch plugin," *GitHub*, 14-Feb-2014. [Online]. Available: https://github.com/srobbin/jquery-backstretch. [Accessed: 06-Aug-2015].

[137] S. Zach and D. Zach, "jquery-supersized plugin," *GitHub*, 14-Aug-2014. [Online]. Available: https://github.com/buildinternet/supersized. [Accessed: 07-Aug-2015].

[138] A. Deveria, "CSS object-fit browser support," *caniuse*, 2015. [Online]. Available: http://caniuse.com/object-fit. [Accessed: 31-Jul-2015].

[139] M. Chouinard and C. Gimmer, "Beautiful Free Stock Photos (CC0)," *StockSnap.io*, 2015. [Online]. Available: https://stocksnap.io/. [Accessed: 06-Aug-2015].

[140] T. Lengemann, B. Jospeh, and I. Joseph, "Free stock photos," *Pexels*. [Online]. Available: http://www.pexels.com/. [Accessed: 06-Aug-2015].

[141] "Iconfinder - 575,000+ free and premium icons," 2015. [Online]. Available: https://www.iconfinder.com/. [Accessed: 06-Aug-2015].

[142] "Modernizr," 2015. [Online]. Available: http://v3.modernizr.com/download/. [Accessed: 06-Aug-2015].

[143] J. Tangelder, "Hammer.JS," 2015. [Online]. Available: http://hammerjs.github.io/. [Accessed: 06-Aug-2015].

[144] Google, "jQuery," 2015. [Online]. Available: https://jquery.com/. [Accessed: 06-Aug-2015].

[145] D. Walsh, "Fullscreen API Implementation," *David Walsh's Blog*, 23-Dec-2013. [Online]. Available: http://davidwalsh.name/fullscreen. [Accessed: 07-Aug-2015].

[146] A. Deveria, "Fullscreen API browser support," *caniuse*, 2015. [Online]. Available: http://caniuse.com/fullscreen. [Accessed: 07-Aug-2015].

[147] S. Pieters, "Bug 135935 – srcset with w descriptor and sizes does not behave as expected," *WebKit Bugzilla*, 14-Aug-2014. [Online]. Available: https://bugs.webkit.org/show_bug.cgi?id=135935. [Accessed: 08-Aug-2015].

[148] P. LePage, "Images in Markup — Web Fundamentals," *Google Developers*, 30-Apr-2014. [Online]. Available: https://developers.google.com/web/fundamentals/media/images/images-in-markup?hl=en. [Accessed: 09-Aug-2015].