Testing and Debugging Cross-Device Applications

Master Thesis

Nina Heyder
<heydern@student.ethz.ch>

Prof. Dr. Moira C. Norrie
Maria Husmann

Global Information Systems Group
Institute of Information Systems
Department of Computer Science
ETH Zurich

13th October 2015

Abstract

Nowadays, many people have access to multiple devices simultaneously. However, those devices are rarely used in a collaborative fashion. Cross-device applications aim to facilitate such collaborations by distributing interfaces among devices and synchronizing data between them. Recently, a lot of frameworks for developing cross-device applications have been created. Despite this abundance of frameworks, only few cross-device applications are used in the wild. One reason for the lack of popularity of cross-device applications could be the limited support for testing and debugging them. Without proper testing, releasing a high-quality product is a difficult task. Our analysis has shown that existing cross-device frameworks provide little to no support for testing and debugging and debugging tools for testing traditional and responsive websites are typically not well suited for debugging multiple devices simultaneously. In this thesis, we have developed XDTools, a set of tools that supports the testing and debugging of cross-device applications, in particular web-based applications, by extending traditional debugging tools with support for multiple devices, e.g. the JavaScript console provided by most browsers, as well as providing some new tools, e.g. connection management. The primary goal of XDTools is to facilitate the testing and debugging of cross-device applications. In order to evaluate XDTools, we developed two sample applications that were tested with XDTools during development. We also conducted a user study where participants used XDTools to complete tasks in a cross-device environment. The study showed promising results and some enthusiastic feedback was provided. Most of the participants appreciated the features provided by XDTools and found that they are well suited for cross-device application testing. Thus, XDTools already improves the debugging process of cross-device applications and serves as a basis for further research in cross-device application testing.

Contents

1 Introduction

Despite the abundance of devices nowadays, up until recently there was no easy way of sharing state information and I/O resources between devices for the end user. Although many users have access to multiple devices at the same time, e.g. their smartphone and laptop, those devices are rarely used in collaboration. Santosa et al. [11] have observed in a field study that many users already use multiple devices in parallel in their workflows and that better functional coordination is needed. In the last few years, cross-device applications have started to fill this gap by facilitating the use of multiple devices in collaboration. Cross-device applications typically run on any device that has access to a modern web browser. The emergence of new web technologies such as Device APIs[1] and WebRTC[2] encouraged the development of a new generation of web-based frameworks that facilitate the development of such cross-device applications.

Despite the large number of frameworks for developing cross-device applications and the identified user needs, there are only few popular cross-device applications. Many of the available cross-device applications are prototypes to showcase the frameworks that they were developed with and most of them are not accessible to the public. In order to release cross-device applications into the wild, they need to be carefully tested and bugs need to be eliminated. However, existing cross-device frameworks have little support for this and either provide no tools for testing applications at all or only very basic tools that are focused on specific aspects, such as seeing what the application looks like on different sets of devices.

In contrast, there are already plenty of practical tools for testing and debugging traditional web applications and many of them can be accessed directly from modern browsers. Google Chrome[3] in particular, but also other browsers, provide quite mature tools for debugging JavaScript, HTML and CSS. Unfortunately, those tools are focused on debugging one device

[1] http://www.w3.org/2009/dap/
[2] http://www.webrtc.org/
[3] http://www.google.com/chrome/

at a time, limiting their usefulness for testing cross-device applications where multiple devices are typically involved simultaneously. Today's devices have many different characteristics, mainly in terms of screen size, but also concerning their input capabilities and connectivity. This diversity of devices requires developers to develop websites that are functional and appealing on all devices. This goal can be achieved by following the principles of responsive design. Many tools have emerged that support testing such websites; some are already built into modern web browsers, others can be accessed as a web service or by installing a program on a desktop PC. In those tools, two different methods for testing responsive websites can be observed: First, different devices can be emulated on a desktop PC. Second, a varied set of actual devices can be used. Google Chrome's Device Mode[4] provides extensive support for emulating devices; apart from emulating the screen size, it can also emulate touch, varying network conditions, location, and more. Other browsers also provide basic facilities for device emulation. When using multiple devices, the developer has to refresh all devices individually whenever the web application has been modified. However, there exist a number of tools that facilitate this. Some allow the developer to reload all devices at once, e.g. Adobe Edge Inspect CC[5], while others also automatically reload devices when files change, e.g. BrowserSync[6]. Some of those tools even allow developers to simultaneously browse their web application on multiple devices. Apart from those tools, there are also web services for testing websites across multiple devices and platforms. Such web services typically include a screenshot generation service that renders a given website on a large number of devices. An example of such a web service is CrossBrowserTesting[7].

Those tools are already a good starting point for testing and debugging cross-device applications. However, web applications targeted at one device at a time and cross-device applications have some fundamental differences that are not accounted for by those tools. In cross-device applications, multiple devices are typically used simultaneously and in a coordinated manner. Also, different devices do not necessarily show the same thing in cross-device applications, which limits the use of mirroring interactions from one device to all other devices. Furthermore, most of the tools for emulating devices focus on emulating one device at a time, requiring the developer to open multiple browser windows, possibly with different user profiles or in incognito mode to prevent the sharing of local resources. Finally, all those tools focus either on emulating devices or on using real devices, but with cross-device applications, it might be desirable to combine both approaches. Due to these differences, testing and debugging cross-device applications is still a challenging task. In the following section, we will describe how our project contributes to conquering the challenges in cross-device application testing and debugging.

1.1 Contributions

Our project aims to facilitate the testing and debugging of cross-device applications by providing multiple tools that assist the developer. As a first step towards achieving our goal, we have analyzed existing tools for debugging web applications. This includes tools built

[4] https://developer.chrome.com/devtools/docs/device-mode
[5] https://www.adobe.com/ch_de/products/edge-inspect.html
[6] http://www.browsersync.io/
[7] http://crossbrowsertesting.com/

directly into browsers as well as external tools. Due to the similarities between cross-device applications and responsive web applications, e.g. the fact that both are supposed to work on devices with many different characteristics, tools for testing responsive websites are of particular interest. During our analysis, we have gathered the limitations of those tools and possible remedies for these limitations. Furthermore, we have also investigated some frameworks for developing cross-device applications. The analysis of those frameworks has shown that even though many frameworks could benefit from cross-device testing tools, few such tools are provided by the frameworks themselves. The limited number of available tools makes exhaustive testing of a cross-device application a difficult task. Finally, we have gathered some requirements for more suitable tools for cross-device application testing based on the limitations of existing tools.

Based on these requirements, we have designed and implemented a new set of tools, called XDTools, for testing and debugging cross-device applications. XDTools allows testing both on real devices and emulated devices and provides a number of different features: Some features have been directly adopted from existing tools while others have been extended to suit the needs of cross-device application testing. Some features that are only useful in a cross-device environment are completely new and not based on any existing tools.

During the development of XDTools, we have implemented two sample cross-device applications using XD-MVC[8], a cross-device application development framework. While developing these applications, we have used XDTools for testing and debugging them. These applications have helped us in multiple ways while developing XDTools: First, actually using XDTools for developing cross-device applications has provided some new ideas for crucial features that were still missing from XDTools. Second, it has helped us improve existing features and also revealed some bugs that could be fixed. Finally, we have also learned something about the usability of XDTools.

Eventually, we have conducted a user study to evaluate the usefulness and suitability of XDTools, compared to traditional methods of testing web applications. During this study, participants got the opportunity to use XDTools for implementing a new feature in a cross-device application and for finding and fixing a bug. The study has shown that XDTools is indeed considered useful for testing cross-device applications. Furthermore, some participants have also provided some new ideas for features and improvements for our existing features as well as the layout of XDTools.

In summary, our project makes the following contributions:

- Analysis of the limitations of existing tools for web application testing in general as well as responsive design testing.
- Development of XDTools, a set of tools for testing and debugging cross-device applications.
- Development of two sample cross-device applications.
- User study for evaluating XDTools.

[8] `https://github.com/mhusm/XD-MVC`

1.2 Structure of This Document

We conclude this chapter by giving an overview of the structure of this document:

In Chapter 2, we present some background information as well as related work to our project. In particular, we describe existing tools for testing web applications as well as cross-device application development frameworks.

In Chapter 3, we describe the requirements for XDTools that we gathered during our analysis.

In Chapter 4, we present XDTools.

In Chapter 5, we describe the architecture and implementation of XDTools in detail.

In Chapter 6, we describe the sample applications that we developed and the insights we gained from them.

In Chapter 7, we describe the user study and present its results.

In Chapter 8, we conclude the work by showing what was achieved and what problems remain, and address possible future work that might be based on XDTools.

2 Background and Related Work

In the following sections, we will present some background information and related work to our project. We will start by analyzing existing tools for testing and debugging traditional web applications, including debugging tools that are integrated directly into browsers. We will also discuss some tools aimed at testing responsive web applications. Finally, we will present a few cross-device application development frameworks that either provide some cross-device testing mechanisms themselves or that could benefit from such mechanisms.

2.1 Web Application Testing and Debuggings

In the following subsections, we will present a few tools supporting the testing and debugging of traditional web applications. Many cross-device applications are web applications and we decided to focus only on those applications. Therefore, tools for debugging traditional web applications are clearly also relevant for our project.

2.1.1 Browser-Integrated Debugging Tools

Over the last few years, browser manufacturers have added more and more tools for debugging web applications directly from the browser. Chrome arguably provides the most advanced debugging tools, but all browsers include at least basic tools for debugging web applications. We will now present a few of those tools provided by browsers themselves.

Chrome DevTools

Google Chrome provides a wide range of features for testing and debugging web applications. First of all, it lets the developer inspect the DOM tree of an application and allows on-the-

fly editing of DOM elements. Furthermore, new CSS rules or properties can be added and existing ones can be modified. Figure 2.1 shows a screenshot of the HTML and CSS inspector in Chrome DevTools. Another useful feature is the JavaScript Console. It has two main purposes: First, it can be used to log diagnostic information in the development process. Second, it is a shell prompt which can be used to interact with the document and DevTools. Chrome DevTools can also be used to debug JavaScript. It lists all scripts that are part of the inspected page. Breakpoints can be set in the scripts and standard controls to pause, resume and step through code are provided. All features mentioned before are also available for debugging remote devices. Remote debugging can be used by connecting a device to the desktop PC with a cable.

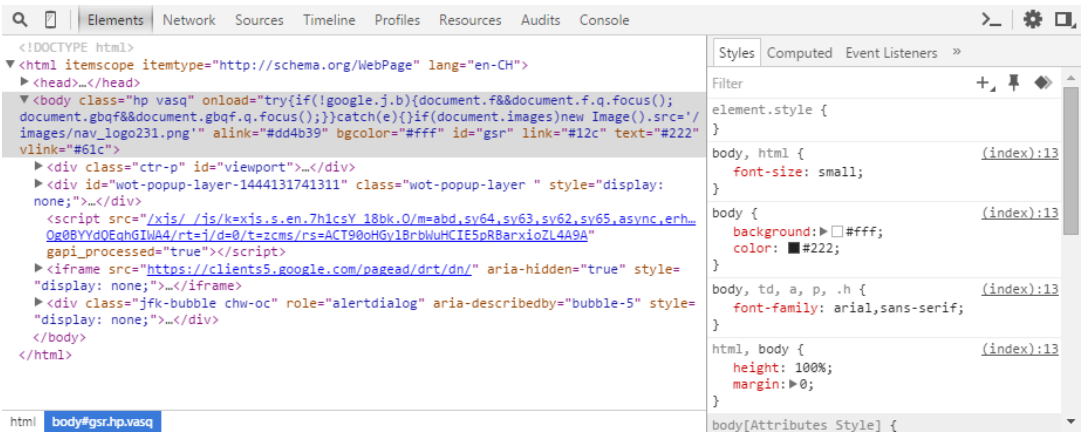


Figure 2.1: HTML/CSS inspector of Chrome DevTools

Chrome DevTools provides many more features useful for debugging, but explaining them all would exceed the scope of this document. While all those features are very useful for testing and debugging web applications, their main disadvantage is that they can only be used to debug one device at a time.

Firefox Developer Tools

Firefox[1] provides developer tools similar to Chrome DevTools. It also has a page inspector that allows developer to inspect the DOM tree and modify the CSS. It also features a console, a JavaScript debugger and remote debugging. Like Google Chrome's DevTools, Firefox's developer tools provides more features that we will not explain here. Furthermore, Firefox gives developers access to a feature called Developer Toolbar. The Developer Toolbar is a command-line tool that can be used to quickly access a number of Firefox's debugging tools. Overall, Chrome DevTools and Firefox Developer Tools are very similar and thus also have the same limitations. Just like Google Chrome, only one device can be debugged at a time in Firefox.

[1] `https://developer.mozilla.org/en-US/docs/Tools`

Summary

Internet Explorer[2], Microsoft Edge[3], Opera[4], Safari[5] and other browsers all provide some kind of developer tools. However, they do not provide any features that we did not already mention before, thus we will not provide any detailed description of the developer tools available in those browsers. All those browsers share the same limitation: Their tools can only be used to debug one device at a time. For cross-device applications however, it would be desirable to debug multiple devices at a time without having to navigate between windows all the time.

2.1.2 Record and Replay

Record and replay has already successfully been used for debugging web applications. It allows developers to quickly reproduce bugs that are otherwise difficult to reproduce. Reproducing bugs in cross-device applications can be even more difficult than in traditional web applications because multiple devices and users are involved simultaneously. In the following, we will describe some tools that support record and replay in web applications.

Timelapse

In [3], Burg et al. describe their record and replay tool Timelapse. Timelapse is a tool for recording, reproducing, and debugging interactive behaviors in web applications. Timelapse is built on Dolos, a record/replay infrastructure that ensures deterministic execution by capturing and reusing user inputs, network responses, and other non-deterministic inputs. Developers can use Timelapse to browse, visualize, and seek within recorded program executions (see Figure 2.2) while simultaneously using familiar debugging tools such as breakpoints and logging. The developers of Timelapse conducted a user study with 14 web developers which showed no significant effect on task times, task success, or time spent reproducing behaviors when developers had access to Timelapse. Expert developers seemed to better integrate Timelapse into their workflow, using Timelapse to accelerate familiar tasks rather than redesigning their workflow. However, Timelapse distracted less-skilled developers that were led astray by unverified assumptions.

[2] https://msdn.microsoft.com/library/bg182326(v=vs.85)
[3] https://msdn.microsoft.com/en-us/library/dn904498(v=vs.85).aspx
[4] http://www.opera.com/dragonfly/
[5] https://developer.apple.com/safari/tools/

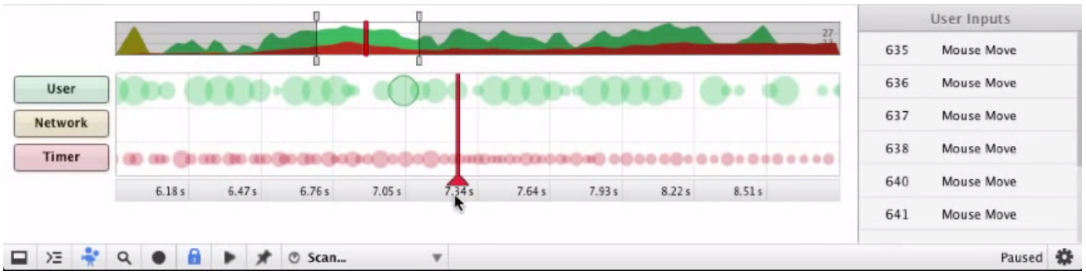


Figure 2.2: Screenshot of Timelapse

Mugshot

Mugshot, developed by Mickens et al. [7], is a record/replay system that captures all events in a JavaScript program, allowing developers to deterministically replay past executions of web applications. The goal of Mugshot is to provide low-overhead, ”always-on” capture and replay for web-deployed JavaScript programs. Mugshot logs explicit user interactions like mouse clicks as well as background activities such as random number generation and the firing of timer callbacks. The client-side log is sent to the developer in response to a trigger like an unexpected exception being caught. The developer can then use Mugshot's replay mode to recreate the original JavaScript execution in their unmodified browser.

WaRR

WaRR [1] is a high-fidelity, ”always-on” tool that records and replays the interactions between users and web applications. The WaRR recorder is embedded directly into the web browser and the WaRR Replayer uses an enhanced, developer-specific web browser that enables more realistic simulation of user interactions based on the recorded traces. Andrica et al. developed two tools on top of WaRR:

- WebErr allows testing of web applications against human errors, i.e. navigation errors and timing errors.
- AUsER automatically generates user experience reports: If a user experiences a bug while using a web application, they press a button and the developers of the application receive the sequence of actions that led to the bug.

Using WebErr, the developers of WaRR were able to find a bug in Google Sites[6].

Rumadai

Yildiz et al. [14] developed Rumadai, a Visual Studio plug-in that helps developers test web applications by recording and replaying client-side events. Rumadai injects JavaScript code into web pages to be deployed at servers. The injected code records user events as well as client-side dynamic content requests and their responses. The recorded events are sent to a

[6] `https://www.google.com/work/apps/business/products/sites/`

database and can be queried by the developers of the web page. The recorded events can then be replayed in a browser using Rumadai seamlessly from Visual Studio.

FireCrystal

FireCrystal [10] is a Firefox extension that allows developers to extract the implementation details of interactive behaviors from other websites. Developers can tell FireCrystal to start recording and then demonstrate the interactive behavior they want to extract. FireCrystal records the interaction, keeping track of DOM changes, JavaScript executions and user input events. The developer can then replay the interactions and FireCrystal displays the HTML, CSS and JavaScript code that affected a particular element at any specific time. FireCrystal also provides an execution timeline that developers can scrub back and forth.

Selenium

Selenium[7] consists of two different parts: The first part, Selenium IDE, allows the developer to record, edit and debug tests. Apart from the recording capability, Selenium IDE also allows editing of scripts by hand. The second part, Selenium WebDriver, accepts commands and sends them to the browser, allowing the developer to control the browser and automate it. The developer can write automated tests in (almost) any programming language, allowing for better integration of Selenium into existing testing environments.

Summary

The tools described above all employ record and replay in some way for improving web applications. FireCrystal focuses more on extracting interactive behaviors from other websites, but it does so by providing record and replay mechanisms. The other tools all focus on debugging web applications. WaRR and Mugshot provide recording mechanisms directly to users of web applications, providing them with means of submitting bug reports to the developers of the applications. Some of those tools are designed for only replaying event sequences on the devices they were recorded on, while others allow replaying event sequences on different devices, e.g. the sequences can be replayed on the developer's machine after a user has recorded and submitted the bug. However, all of those tools have one limitation in common: They focus on replaying an event sequence on one device at a time. In cross-device scenarios, multiple devices can interact with the same application simultaneously and it would be desirable to replay interaction sequences on multiple devices in parallel.

Since record and replay mechanisms have already been shown to help with debugging in traditional web applications, they could certainly also be useful for debugging cross-device applications. In particular, we think that replaying event sequences on multiple devices simultaneously can help accurately simulate multiple users using a cross-device application. Developers usually work alone when fixing a bug, or in very small groups, which makes reproducing bugs that require multiple devices to interact with each other a very difficult task.

[7] `http://www.seleniumhq.org/`

However, if multiple devices replay event sequences simultaneously, mechanisms for accurately timing event execution are also required. Most of the tools described above do not include such mechanisms, or they include some mechanisms that are however not sufficient for configuring timing in a cross-device replaying scenario.

2.2 Responsive Web Application Testing

There are a number of tools for testing responsive websites. Some of them can be accessed on the web as a service, while others are desktop programs that have to be installed. Some focus on testing on real devices and others focus on emulating devices. In the following subsections we will describe some of those tools. Responsive web applications have a few similarities with cross-device applications: Both types of applications are supposed to work on a number of different devices with different screen sizes, input capabilities and more. Thus, tools for testing responsive web applications are to some extent also suited for testing cross-device applications.

2.2.1 Web Services

The following two tools can be accessed through a browser and do not have to be installed.

BrowserStack

BrowserStack[8] allows developers to select browsers and devices and then generate screenshots. It is also possible to live test one device at a time. Furthermore, there are developer tools for remote devices and Selenium cloud testing is possible. Even though BrowserStack allows the developer to see what an application looks like on many devices simultaneously, its usefulness for cross-device application testing is limited: Only one device can be live-tested at a time, making it difficult to accurately simulate a cross-device scenario. Also, even though real iOS devices are used, only emulated Android devices are available. One of the advantages of tools such as BrowserStack is that applications can be tested on many different real devices without having to buy those devices. Providing only emulated Android devices diminishes the usefulness of such a tool.

CrossBrowserTesting

With CrossBrowserTesting, developers can select a number of devices as well as the operating system, browser and resolution and generate screenshots. The layout differences between different devices can then automatically be analyzed. Furthermore, websites can also be live tested and Selenium automated testing is available as well. The main advantage of CrossBrowserTesting is that all screenshots are generated on real devices and a very wide variety of devices is available. Also, interactions can be tested using Selenium. The usefulness of the

[8] `https://www.browserstack.com/`

tool is again limited by the fact that live testing is only possible on one device at once. Additionally, while detecting layout differences is a useful feature in general, it is not yet very mature and some layout differences that are detected seem rather trivial (the body element of a larger device is larger), while other differences are not noticed at all. Figure 2.3 shows CrossBrowserTesting while taking screenshots on multiple devices.

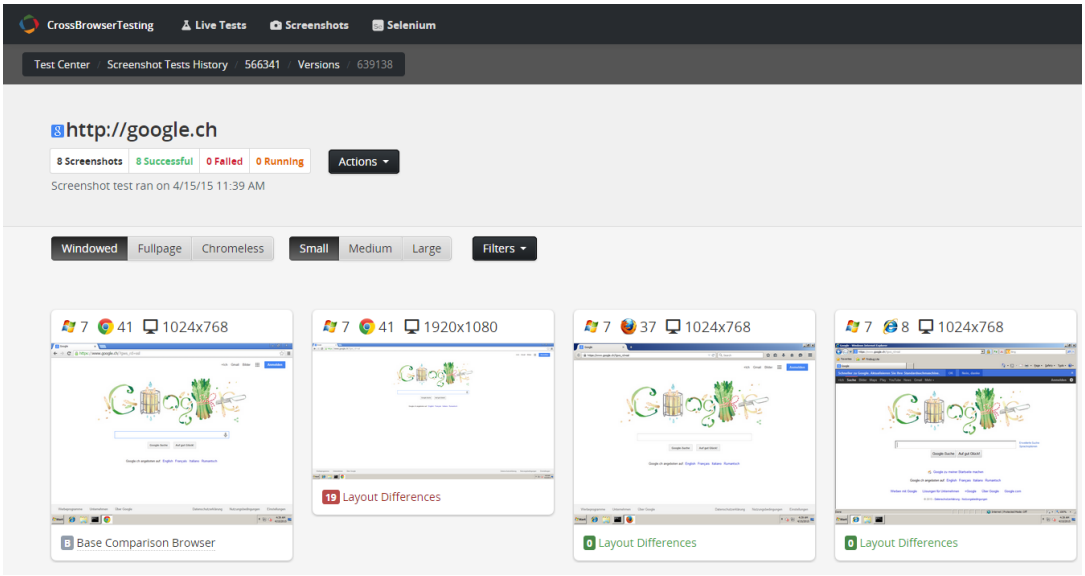


Figure 2.3: Taking screenshots on multiple devices simultaneously

2.2.2 Testing with Actual Devices

The following tools allow the developer to connect multiple real devices to the tool and test their application on them.

Remote Preview

Remote Preview[9] allows synchronizing URLs across multiple devices. This allows fast previewing of a web application on multiple devices. In cross-device scenarios, loading the same URL on all devices can help in applications where devices are paired by opening the same URL on all devices. However, Remote Preview provides a rather limited set of features, thus using Remote Preview alone will probably rarely be enough for successfully testing web applications.

BrowserSync

BrowserSync (see Figure 2.4) provides a large number of features for testing websites. It allows remote debugging of HTML and CSS, can add CSS outlines or box shadows to all elements and add a CSS grid overlay. It can also load a URL on all devices, refresh all

[9] `https://github.com/viljamis/Remote-Preview`

devices and automatically refresh devices when files are changed. Furthermore, it allows synchronizing interactions between devices, i.e. clicks, scrolls, form submits, form inputs and form toggles. It also provides network throttling. The advantages of BrowserSync are that it provides a wide range of features, including synchronizing interactions, which is a feature that distinguishes it from many other tools. However, for cross-device application testing, synchronizing interactions among all devices is of limited usefulness, as different devices have different roles and thus also different responsibilities.

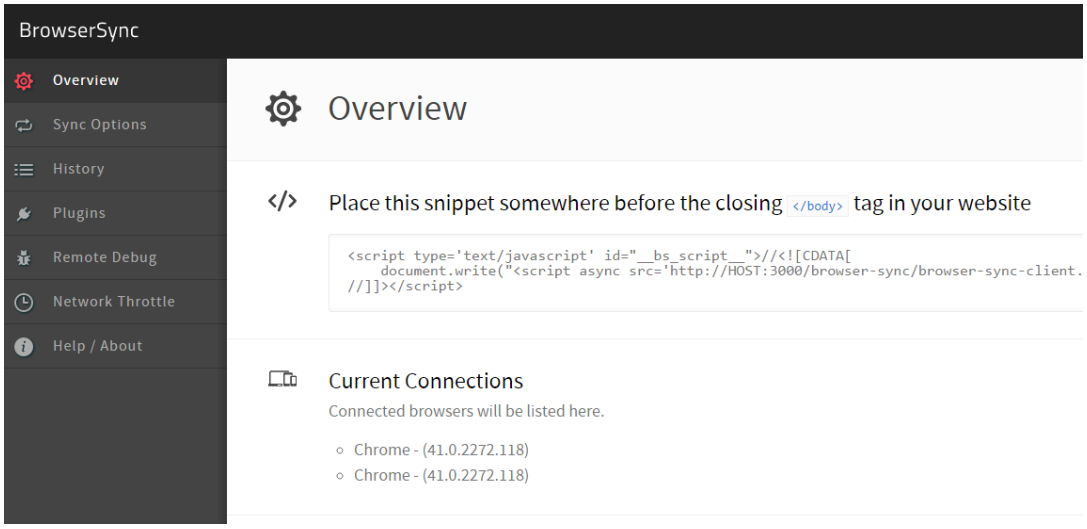


Figure 2.4: Screenshot of BrowserSync

Adobe Edge Inspect CC

Adobe Edge Inspect CC allows developers to take screenshots on all connected devices simultaneously. The screenshots are then automatically transferred to a folder on the desktop PC. It can also refresh all devices simultaneously. Furthermore, the URL that is opened on the desktop PC is loaded on all other connected devices. It also allows remote HTML and CSS debugging using weinre[10]. The main advantage that distinguishes Adobe Edge Inspect CC from other tools is that is simply synchronizes the URL that the developer is currently looking at on the desktop PC, thus it works even if the developer switches tabs or browser windows. However, the fact that the URL cannot be changed from devices other than the desktop PC could be a disadvantage in some scenarios. Also, the installation process is rather extensive: A program needs to be installed on the desktop PC as well as a Chrome extension and an app needs to be installed on all mobile devices that the developer wants to connect. Regarding cross-device applications, again, refreshing all devices at once can be useful as well as synchronizing URLs in some cases, but other than that, it does not provide any features that help with cross-device application testing.

[10] https://people.apache.org/~pmuellr/weinre-docs/latest/

Ghostlab

Ghostlab[11] is one of the most advanced tools for responsive web application testing and provides features similar to BrowserSync. It can also load a URL on all devices or refresh all devices at once and refresh automatically when files are changed. It provides means for synchronized browsing as well as synchronized HTML and CSS inspection on multiple devices. Furthermore, it can automatically fill out forms and provides remote Javascript debugging. Synchronization can be turned on and off on a per-device basis, which makes the tool more useful than tools that simply synchronize all devices. However, the usefulness is still limited because constantly changing the devices that should be synchronized is time-consuming and error-prone. Also, interactions on cross-device applications typically do not happen in a synchronous fashion.

2.2.3 Device Emulation

The following tools allow the developer to emulate mobile devices on a desktop PC.

Chrome Device Mode

Google Chrome provides extensive support for emulating devices with its Device Mode (see Figure 2.5). If the developer opens the DevTools, they can switch to the Device Mode and emulate different devices. The developer can select an arbitrary device from a list of predefined devices or create a custom device. They can also enable network throttling, touch emulation and location emulation. The large number of different aspects that are emulated in Device Mode make it very well suited for testing responsive web applications. While most other device emulation tools are limited to emulating resolution and sometimes touch, Chrome Device Mode allows to emulate many other aspects as well.

[11] `http://www.vanamco.com/ghostlab/`

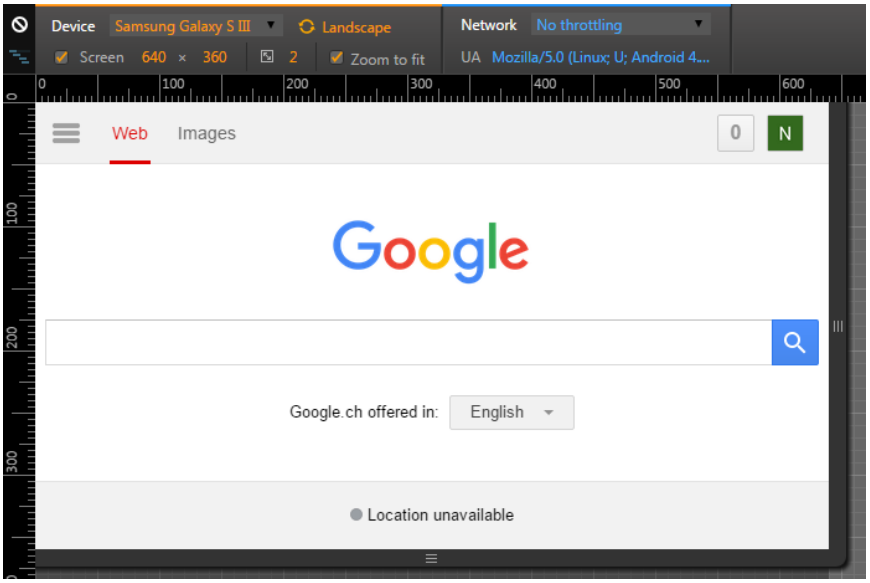


Figure 2.5: Screenshot of Chrome Device Mode

Firefox's Responsive Design View

Firefox's Responsive Design View also allows developers to emulate devices, similar to Chrome's Device Mode, but it provides fewer features. Instead of focusing on existing devices, it provides a short list of common resolutions that can be selected by the developer. It also allows to emulate a custom resolution. Additionally, the orientation of the devices can be switched, touch can be emulated, and screenshots of the displayed web page can be taken.

2.2.4 Summary

While the tools described above already provide a wide variety of features that are immensely useful for responsive web application testing as well as web application testing in general, the distinguishing characteristics of cross-device applications lead to a limited usefulness of those tools. In summary, there are two different approaches to testing cross-device applications in the tools described above: First, applications can be tested on real devices, either through web services or the developer's own real devices. Second, emulated devices can be used. In most of those tools, only one device can be tested at a time, but in cross-device scenarios, multiple devices are typically involved. In some of the tools, interactions can be synchronized among multiple connected real devices. However, this is also problematic for testing cross-device applications because not all devices perform the same interactions and those that do, do not necessarily perform them at exactly the same time. Two features that some of the tools mentioned above provide and that would definitely also be useful for cross-device applications are refreshing all devices at once and loading a URL on all devices. Remote HTML, CSS and JavaScript debugging are also desirable in a cross-device scenario. Something similar to Selenium testing is clearly also useful in cross-device scenarios. However, depending on the device, different tests would be needed and it should be possible to run those tests in parallel.

2.3 Cross-Device Application Development Frameworks

In the following subsections, we will describe some frameworks that facilitate the development of cross-device applications. Some of those frameworks already provide some basic tools for testing the applications developed with them, but others include none and could benefit a lot from such tools.

2.3.1 Frameworks With Tool Support

The following frameworks already include some mechanisms for testing and debugging applications, but most of those mechanisms are limited to specific aspects of the applications and require the applications to be developed with the framework.

XDStudio

In [9], Nebeling et al. present their web-based GUI builder, XDStudio (see Figure 2.6). XDStudio is designed to support interactive development of cross-device web interfaces. It has two complementary authoring modes: *Simulated authoring* allows designing for a multi-device environment on a single device by simulating other target devices. *On-device authoring* allows the design process itself to be distributed over multiple devices. The design process is still coordinated by a main device, but directly involves target devices. The user can switch between two different modes: In *use mode*, the user can interact normally with the interface loaded into the editor. In *design mode*, the user can manipulate the interface directly. XDStudio also allows the specification of *distribution profiles* in terms of involved devices, users and target user interfaces.

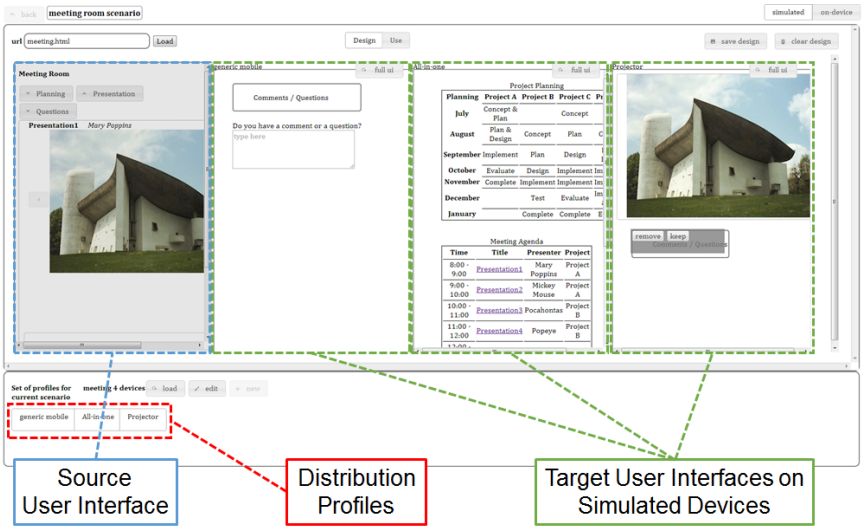


Figure 2.6: Screenshot of XDStudio

Although XDStudio includes mechanisms for inspecting interactive designs on emulated devices and on connected real devices, there is no specific support for debugging such as

access to the console. Thus, XDStudio is well suited for seeing what an application looks and feels like on different devices, but debugging the application when something does not work as expected is still a difficult task.

XDSession

XDSession, developed by Nebeling et al. [8], is a framework for cross-device application development based on the concept of *cross-device sessions*. The session controller supports management and testing of cross-device sessions with *connected or simulated devices* at run time. The session inspector enables inspection and analysis of multi-device/multi-user sessions with support for *deterministic record and replay* of cross-device sessions. A session consists of users, devices, and information.

XDSession provides a capture and replay mechanism for user interactions and changes to sessions and provides a basic device emulation mode that allows developers to emulate one device at a time. However, the record and replay mechanism only works if the framework's API is used for the manipulations. Also, XDSession supports debugging at a rather high level of abstraction and is thus better suited for finding problems in interactions rather than bugs in the source code.

Weave

Weave is a web-based framework for creating cross-device *wearable* interaction by scripting, developed by Chi et al. [4]. It provides a set of high-level APIs for developers to easily *distribute UI output* and *combine sensing events and user input* across mobile and wearable devices. Devices can be manipulated regarding their capabilities and affordances, rather than low-level specifications. Weave also has an integrated authoring environment for developers to program and test cross-device behaviors. Developers can test their scripts based on a set of simulated or real wearable devices.

Weave allows testing on emulated and real devices and the developer also has access to a log panel, but no further support for debugging is provided.

WatchConnect

In [5], Houben et al. present WatchConnect. WatchConnect is a toolkit for rapidly prototyping cross-device applications and interaction techniques with smartwatches. It provides an *extendable hardware platform* that emulates a smartwatch, a *UI framework* that integrates with an existing UI builder and a rich set of *input and output events* using a range of built-in sensor mappings. WatchConnect is built around a wired prototyping watch, a smart watch emulator composed of a display, a number of sensors and a microprocessor. Applications can thus be tested and debugged directly on the watch prototype. WatchConnect also includes some tools for supporting developers while debugging, but those tools are focused on the machine learning algorithms and the recording of sensor data.

2.3.2 Web-Based Frameworks

The following frameworks allow developers to develop cross-device web applications but include no tools for testing and debugging the applications developed with them and would certainly benefit from such tools.

XD-MVC

XD-MVC is a framework that combines cross-device capabilities with MVC frameworks. It can be used as a plain JavaScript library or in combination with Polymer. The framework consists of a server-side and a client-side part. For communicating, either a *peer-to-peer* or a *client-server* approach can be used. Developers that use XD-MVC can assign *roles* to devices that are connected to the application. Depending on the role that a device is assigned, different parts of the interface are shown on the device.

Connichiwa

Connichiwa by Schreiner et al. [12] is a framework for developing cross-device web applications. It runs local web applications on one of the devices without requiring an existing network or internet connection. *No remote server* is used, instead a native helper-application runs a web server on-demand. The native application automatically *detects other devices* using Bluetooth Low Energy, which are then connected by sending the IP address of the local web server over Bluetooth. Connichiwa's *JavaScript API* gives easy access to common functions like device detection and connection and it also provides JavaScript events to notify about device detection and connection.

DireWolf

DireWolf is a framework for developing distributed web applications based on widgets. It was developed by Kovachev et al. [6]. Direwolf provides a *framework for easy browser-based distribution* of web widgets between multiple devices, facilitates extended *multi-modal real-time interactions* on a federation of personal computing devices and provides *continuous state-preserving widget migration*. DireWolf helps managing a set of devices and handles communication and control of distributed parts of the web application.

Panelrama

Panelrama, developed by Yang et al. [13], is a web-based framework for the construction of applications using distributed user interfaces. It introduces a new *XML element*, "panel", which may be placed around groupings of control and it facilitates the *distribution and synchronization of panels* among the connected devices. The developer can specify the *state information* that should be synchronized across devices as well as the suitability of panels to different types of devices. An *optimization algorithm* distributes panels to devices that maximize their match for the developer's intent.

Polychrome

Polychrome is a web application framework for creating web-based collaborative visualizations that can span multiple devices. It was developed by Badam et al. [2]. It supports cobrowsing new web applications as well as legacy websites with *no migration costs*, an *API to develop new web applications* that can synchronize the UI state on multiple devices to support synchronous and asynchronous collaboration and *maintenance of state and input events* on a server to handle common issues with distributed applications. Polychrome provides the interaction and display space distribution mechanisms to create new collaborative web visualizations that utilize multiple devices and it provides framework modules to store the user interaction. Combined with the initial state of the website, the interaction logs are useful for synchronizing devices within the collaborative environment, consistency management and interaction replay.

2.3.3 Summary

In the sections above, we have described several different frameworks for developing cross-device applications. Some of them are only useful for specific types of applications, while others are suited for cross-device applications of all types. Despite this abundance of frameworks available, little support for testing and debugging the applications that are developed with them is provided. DireWolf, Polychrome and XD-MVC include no mechanisms for testing and run in an unmodified browser. Thus, they could benefit from a web-based tool for testing cross-device applications. Panelrama applications also run in an unmodified browser. The developers of Panelrama mention that a tool for simulating device configurations and previewing the distribution was built on request of some developers. No further details about the tool were mentioned, but the fact that developers requested such a tool shows that it is often difficult to imagine what an application would look like on different device configurations. Thus, the developers of Panelrama applications would certainly also benefit from a cross-device testing tool. Connichiwa requires the installation of a helper application, but the application itself still runs in an unmodified browser. Thus, applications developed with Connichiwa could also be tested with a web-based cross-device testing tool.

Some of the frameworks above allow the emulation of devices as well as the connection of real devices. Emulating multiple devices at a time as well as connecting real devices is a crucial feature for testing and debugging cross-device applications, but without any additional tools that help with debugging, finding and fixing bugs in the applications is still a difficult task. The framework that provides the most support for testing so far is probably XDSession, but as described above, its mechanisms are still not enough for successfully debugging cross-device applications at the source code level.

3 Requirements

In the previous chapter, we have described some tools that provide debugging mechanisms that would also be useful for cross-device testing and debugging. However, most of those mechanisms need to be extended to fulfill the needs of a cross-device testing and debugging tool. In the following sections, we will describe the key requirements for such a tool, i.e. XDTools.

3.1 Emulation of Multiple Devices

Device emulation is already very popular for testing responsive websites. The number of physical devices that a developer has access too is typically rather limited, thus testing on emulated devices helps cover a wider range of devices. In its simplest form, device emulation is just resizing a browser window so it looks and feels like a device with a different resolution. However, manually resizing a browser window such that it has the exact resolution of a real device is difficult. Furthermore, just changing the screen size does not realistically emulate a real device: Mobile devices typically use touch interactions, often have poor network connectivity and have access to a wide range of sensors. Those limitations have led to the emergence of more sophisticated tools for device emulation. Advanced device emulation tools such as the Device Mode in Chrome DevTools emulate different screen sizes, touch capabilities, network conditions, as well as location and acceleration sensors. They also typically provide a list containing some existing devices. In addition, the developer can create custom devices. However, all such tools have one limitation in common: They can emulate only one device per browser tab or window and even if multiple browser windows are used, those browser windows share the same local resources such as local and session storage. This limits the usefulness of such tools for cross-device application testing. Cross-device applications typically run on multiple independent devices simultaneously and thus should not share any local resources. In practice, developers employ a number of different mechanisms to

prevent the sharing of local resources: First, multiple different browser profiles can be used. Second, the incognito mode provided by most browsers can be used to prevent sharing of local resources. Lastly, multiple independent browsers can be used. However, those solutions all have some limitations: Using multiple independent browsers obviously limits the number of devices that can be emulated to a rather small number and requires the installation of all those browsers. Also, all those solutions require the developer to open multiple windows simultaneously and arrange them on the screen. This is tedious and frequent switching between browser windows is required. Furthermore, tasks like creating multiple browser profiles are time-consuming and might not be what the developer wants. Finally, not all browsers support multiple browser profiles, limiting the number of browsers on which an application can be tested. But not only the process of emulating multiple devices itself is tedious: If the developer actually wants to use browser debugging tools, those tools have to be opened for each window separately which requires additional screen space. The screen space of the developer's machine can also be a limiting factor: If the screen has a full HD screen but the developer wants to emulate a full HD device as well as some mobile devices simultaneously, those devices cannot be ordered such that all devices are visible at the same time. The developer would need to put one window in full screen and switch browser windows when they want to access the other devices. Constantly switching between devices can be tedious and the consequences of interactions performed on one device cannot be observed in real-time on other devices because switching between windows requires some time. Also, emulating resolutions larger than Full HD, e.g. a 4k TV, is even more difficult.

Those limitations all contribute significantly to the difficulty of cross-device application testing. Using these limitations, we gathered a number of requirements for device emulation in XDTools:

- It should be possible to *emulate multiple devices* in one browser window.
- The emulated devices should *not share any local resources*. The solution for preventing the sharing of local resources should be *scalable and robust*.
- The screen size should not be a major limiting factor concerning the number of devices that can be emulated simultaneously. This can be realized by *scaling devices* down without changing their resolution.
- The developer should have access to a list containing some *existing devices*.
- The developer should be able to *dynamically change the resolution* of emulated devices.
- The developer should be able to add and save *custom devices*.

Scaling devices down while keeping the resolution and resizing devices are two different concepts. The difference between the two is illustrated in Figure 3.1.

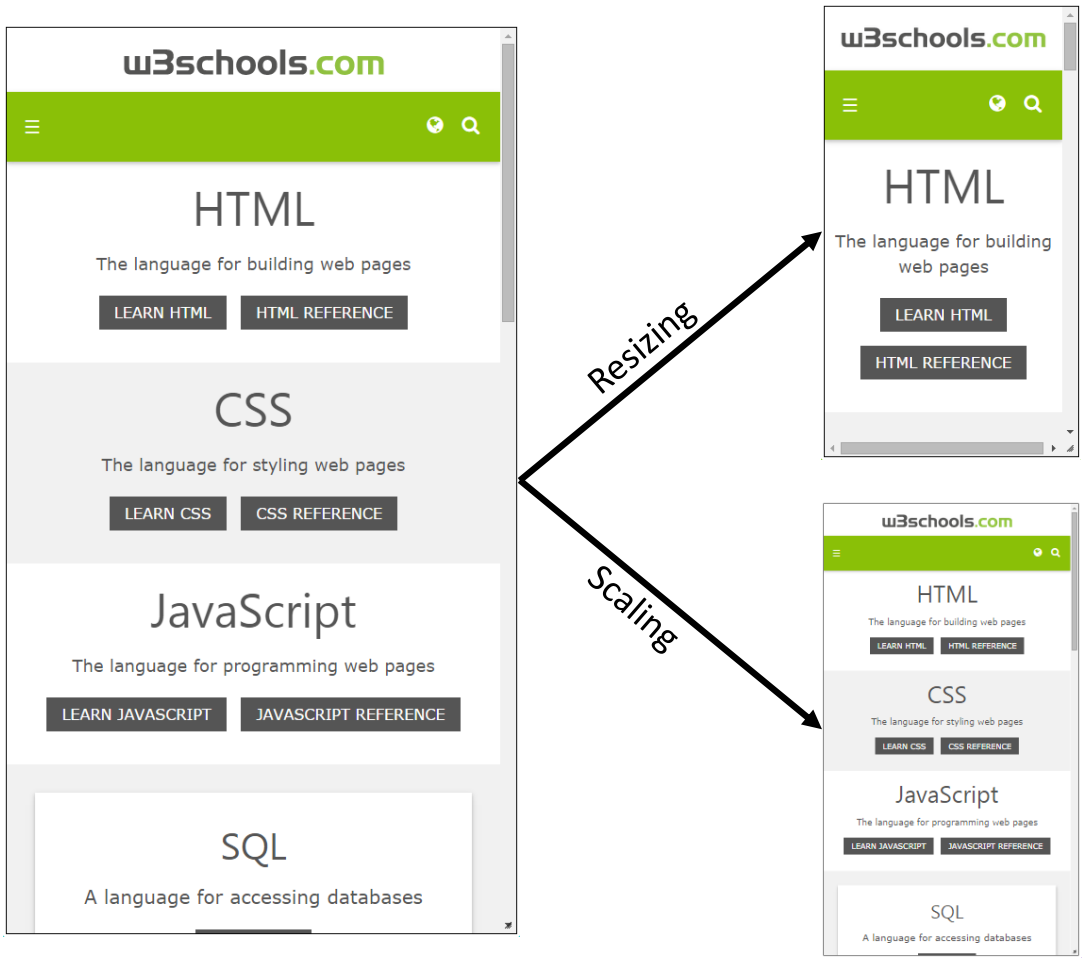


Figure 3.1: Difference between resizing and scaling a device

3.2 Easy Integration of Real and Emulated Devices

Emulating devices is a versatile tool for testing applications on many different devices. However, it does not completely eliminate the need for testing on real devices: Device emulation is always limited to certain aspects that are being emulated, e.g. screen size, resolution, touch interactions and location. However, not every little detail of a real device can be emulated accurately. The following list provides an overview of some of the other limitations regarding testing on emulated devices:

- Touch interactions: Even though modern device emulators can also emulate touch interactions, performing a gesture with the mouse will never feel the same as the actual touch interaction. An interaction that works great with the mouse might feel awkward when performed on a real device and vice-versa. Also, multi-touch interactions such as pinching are difficult to emulate in a realistic way on an emulated device without real touch support.
- Interrupts: While using an application on a real device, the user might be interrupted

by many different things, e.g. the arrival of a text message. Those interrupts cannot be simulated in a realistic way on an emulated device.

- Performance: A desktop PC typically has much more computing power than a mobile device. If an application performs poorly on mobile devices, this might not even be noticed if the developer only tests on emulated devices.
- Display: The display quality and thus also the look of an application varies greatly depending on the device. Only emulating devices on a desktop PC cannot account for those differences in display quality.
- Sensors: Modern devices have a large number of different sensors that cannot all be emulated realistically. One particular problem is the orientation of the device: A user might switch between landscape mode and portrait mode on purpose or accidentally at multiple points. Although the orientation of emulated devices can also be switched, this does not accurately simulate the behavior of a real user.

Although this list gives a good overview of the limitations of testing on emulated devices only, it is by far not complete and what happens on a real device cannot always be foreseen by testing on emulated devices. Thus, testing on real devices is crucial for the successful development of a cross-device application.

The importance of testing on real devices leads to a new requirement for XDTools: It should easily be possible to *connect real devices* to XDTools.

3.3 Easy Switching of Device Configurations

Cross-device applications are typically used by different groups of users and thus also different devices. Even the same user may sometimes use their mobile phone and laptop simultaneously and at other times only their mobile phone or tablet. Thus, the number of devices that are used simultaneously cross-device application and those devices' characteristics may vary greatly. Depending on the devices connected to a cross-device application, the UI distribution and the behavior of the application might be different. A cross-device application needs to be able to support all different device scenarios. Some cross-device applications are targeted to specific scenarios, e.g. a presentation room with multiple big screens that are always present, as well as some mobile devices that are only in the room when their owner is attending a presentation. In such a scenario, the developer would probably want to have access to the two static devices whenever they are testing the application and dynamically add some mobile devices. However, in other scenarios there might be no static devices at all, e.g. a public place where all visitors can connect their devices to an application related to the place. Due to the large number of different device scenarios that could be used with a cross-device application, it is a key requirement for XDTools that the developer can quickly *create multiple different device scenarios* and *easily switch* between those device configurations.

3.4 Integration with Debugging Tools

Many of the features integrated into the debugging tools of browsers are also immensely useful for debugging cross-device applications, and some might even help more when debugging cross-device applications than when debugging traditional web applications. However, those tools are limited to debugging one device at a time. We have established before that XDTools should allow emulation of multiple devices in the same browser window. While this already simplifies the debugging of multiple devices somewhat, it also introduces some new difficulties. The messages that are logged from the emulated devices are all shown in the browser debugger tools of the same window. This aggregation of logging messages is useful for seeing the messages from all devices at one glance, but identifying the device that a message came from is more difficult. The same limitation applies to JavaScript errors that are shown in the console but cannot easily be related to a device. Also, trying out things in the console by sending commands to devices becomes more difficult. Google Chrome allows the developer to switch between different frames in the console and thus address different frames with commands, but it is not always obvious which frame corresponds to which device. Also, the developer might want to try out the same thing on multiple devices and would have to switch between multiple frames to address all of them. Further limitations of the browser-included debugging tools include that navigating to the HTML of an emulated device can be rather tedious, that CSS can only be applied to one device at a time and that function breakpoints can only be added on one device at a time. Especially the last limitation can make cross-device application testing difficult because different devices have different responsibilities and not all devices might use all JavaScript functions. Consequently, adding a breakpoint inside a function on one device might not help with debugging the function at all, if the function is not called on that device. The existence of real devices complicates things even more. By connecting the device to the desktop PC via cable, the developer gets access to remote debugging, but debugging multiple devices at the same time gets even more complicated. The remote debugging of each device is opened in a new window, thus the developer once again has to navigate between multiple windows, something that we wanted to avoid by emulating multiple devices in one browser window. Also, getting an overview of the logging messages from all devices and sending commands to multiple devices becomes even more difficult. However, integrating existing debugging tools into XDTools is clearly desirable: Most of those tools have been around for quite some time. Thus, they are already well tested and have gone through a series of improvements. Also, almost all web developers have already used those tools for extensive testing and are already familiar with them. However, XDTools needs to extend those tools to support debugging on multiple devices simultaneously. Using all this information, we derived the following requirements for XDTools:

- Logging messages from all devices should be aggregated in one place and the device the messages originated from should be easy to identify.
- JavaScript errors from all devices should be aggregated and easily identifiable.
- Sending JavaScript commands to multiple devices at a time should be possible and easy to achieve.
- It should be easy to inspect the HTML of specific devices.

- It should be possible to add CSS to multiple devices at the same time (see Figure 3.2).
- The developer should be able to add breakpoints to multiple devices simultaneously.
- If technically feasible, all of the above requirements should be applied to both real and emulated devices.

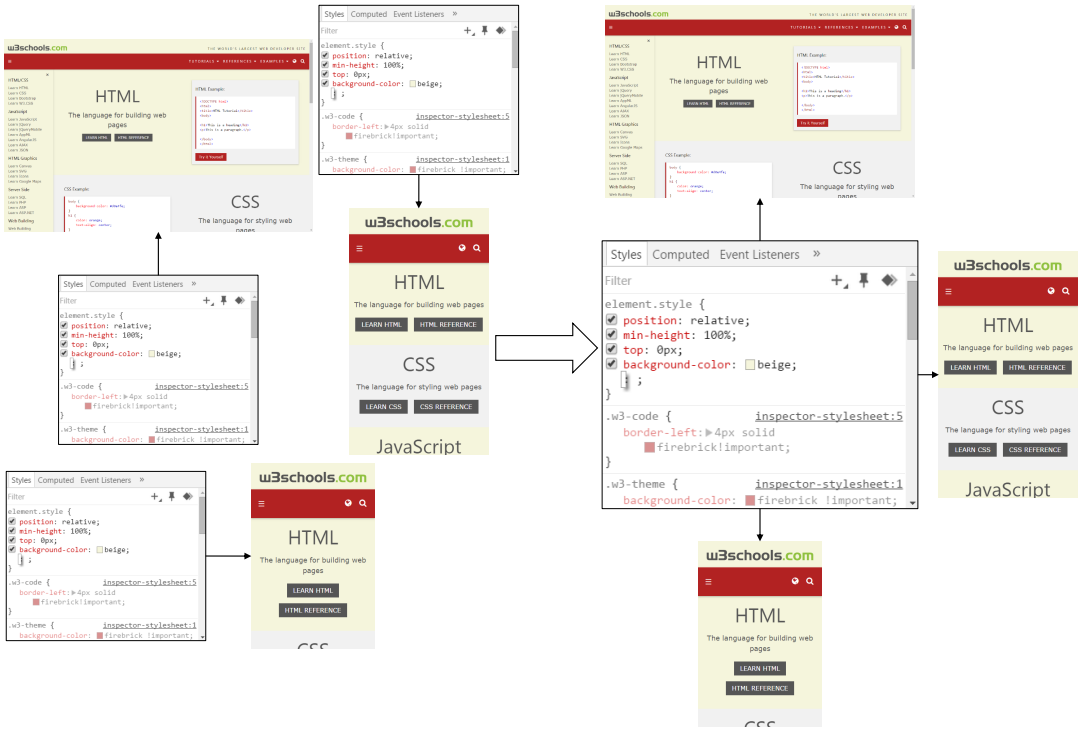


Figure 3.2: CSS editor aggregation

3.5 Automatic Connection Management

In order to use a cross-device application with multiple devices simultaneously, those devices need to be paired with each other. The mechanisms for pairing devices differ between the various cross-device application frameworks: With some frameworks, all devices that open the cross-device application are paired implicitly. In other frameworks, for example XD-MVC, devices can be paired by copying the URL from one device to the other devices. Other frameworks have more complicated mechanisms for connecting devices. In Connichiwa, one device runs a local web server and uses Bluetooth to detect nearby devices. The device then sends the IP of the web server over Bluetooth, enabling the other devices to access the received IP in a web browser. All of those mechanisms have one thing in common: Devices are connected by opening a specific URL in the browser. However, other ways of connecting devices are also feasible: Some frameworks provide a function that can be called from one device to connect the device to another device. Also, many of the papers describing cross-device application development frameworks do not describe how devices are connected. Finally, cross-device applications can also be implemented independently of any framework

and might use completely different mechanisms for connecting devices. Thus, it is impossible to derive all mechanisms that could possibly be used for connecting devices in a cross-device application.

However, if the developer wants to debug a cross-device application, reconnecting the devices every time a new device configuration is loaded or possibly even when devices are refreshed is tedious and time-consuming. Thus, it is desirable to have some easy way of connecting devices. From this, we derive the next requirement for XDTools: It should be possible to *automatically and manually connect devices* to each other. If possible, the provided connection mechanism should work *independent* of the connection mechanism used in the cross-device application under test.

3.6 Coordinated Record and Replay

Record and replay has been used previously for recording and replaying user interactions in traditional web applications and especially AJAX web applications. The non-deterministic and asynchronous nature of web applications contributes much to the value of record and replay in web applications. When a bug is encountered in a web application, it is often difficult to determine the exact steps for reproducing the bug. Reproducing bugs becomes even more difficult in cross-device scenarios where multiple devices are involved and the interactions performed on one device have implications on other devices. Also, cross-device applications are often used by multiple users at the same time and it is difficult for one developer to simulate multiple users interacting with their devices simultaneously. Thus, we believe that record and replay can benefit cross-device application developers even more than developers of traditional web applications. However, precise mechanisms for timing the replay are needed if we want to simulate multiple users simultaneously. Also, simply replaying interactions is not enough: If something goes wrong in the application, we need some way of pausing the replay and inspecting the state of the devices. XDTools should implement a record and replay mechanism that fulfills the following requirements:

- All *interactions* with the device should be *recorded*.
- It should be possible to *replay a sequence of interactions on another device* than the device that recorded the sequence.
- It should be possible to *pause the replay* of the interaction sequence.
- *Accurate timing* of replays should be possible.
- The developer should be able to *store event sequences* for later use, e.g. for testing them again in future iterations of the application under test.

4
XDTools

XDTools is an integrated set of tools that can be loaded directly into the browser. XDTools allows developers to test and debug their cross-device applications in multiple ways. The following sections will describe XDTools in detail.

4.1 Overview of Features

In Figure 4.1, the complete interface of XDTools except for record and replay can be seen.

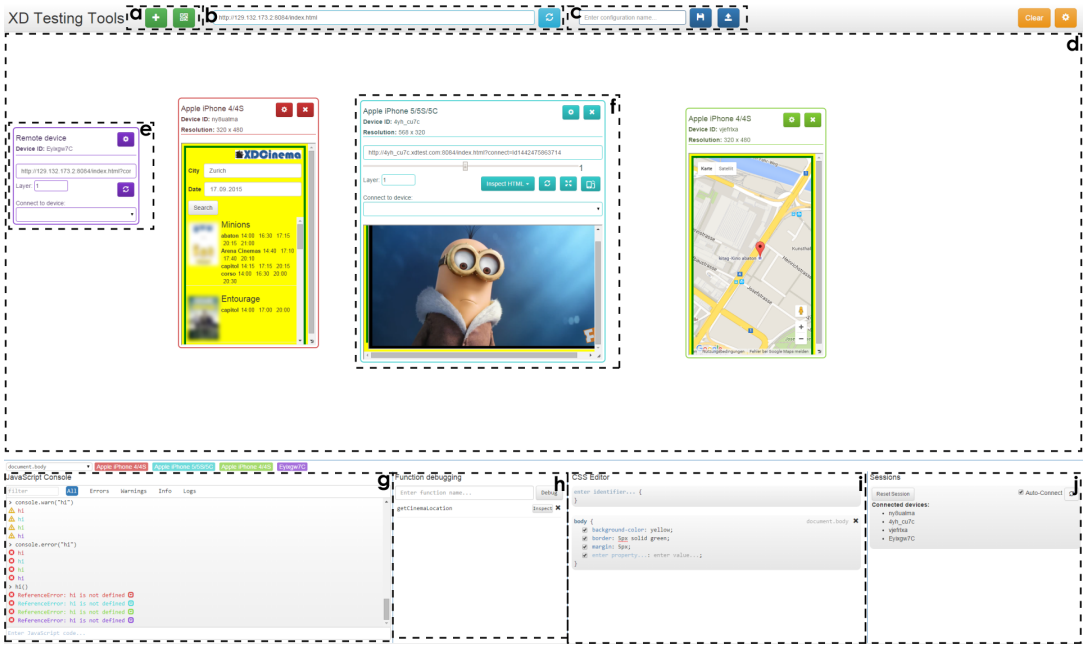


Figure 4.1: The complete interface (without record and replay)

The individual parts of XDTools are labeled in the screenshot:

a) Buttons for adding emulated devices and displaying the QR code that can be scanned with real devices to connect them to XDTools.

b) Input field for loading a URL on all devices and button to refresh all devices.

c) Loading and saving device configurations.

d) Area where the devices can be positioned.

e) Proxy of a connected real device.

f) Emulated device.

g) Shared JavaScript console.

h) Function debugging.

i) Shared CSS editor.

j) Session management.

Figure 4.2 shows a picture of the remote device that is connected to XDTools in the screenshot shown before.

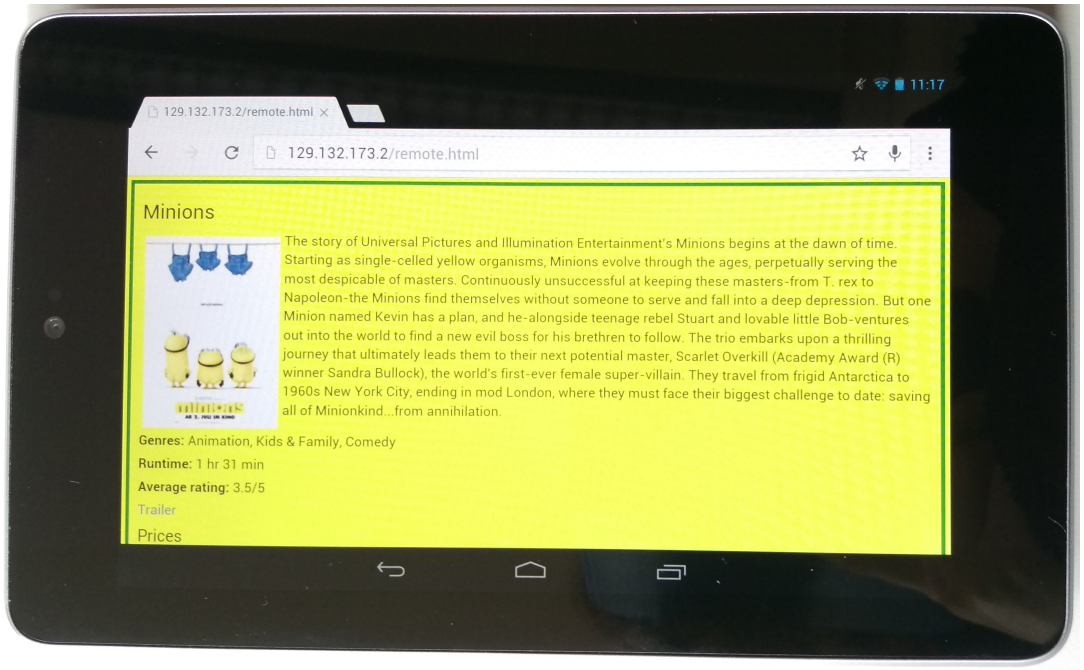


Figure 4.2: The connected remote device

XDTools also provides some options for disabling individual features, such as the shared JavaScript console. If a feature is disabled, the respective interface component is simply hidden. The options also show all stored custom devices, device configurations and event sequences and allow the developer to delete any of them. A screenshot of the options can be seen in Figure 4.3.

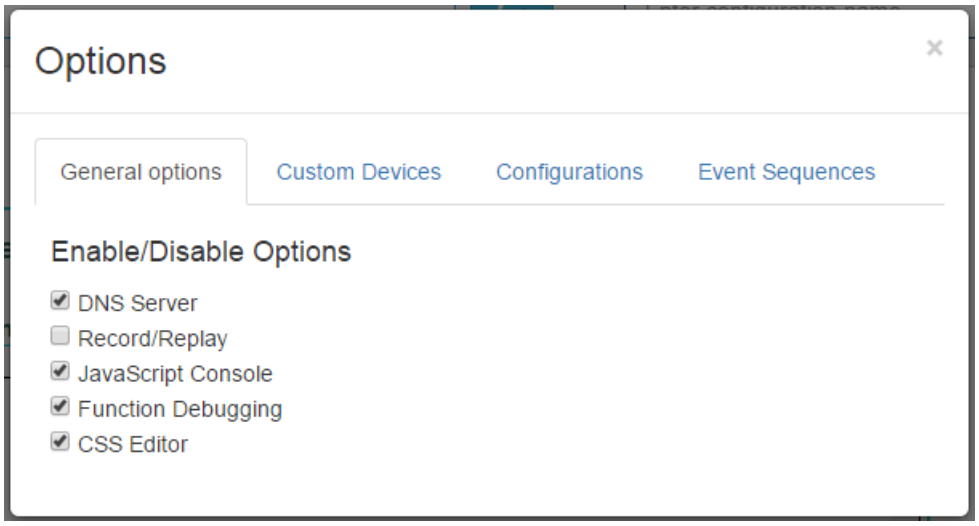


Figure 4.3: Options of our main application

Devices can be activated and deactivated. If a device is inactive, it is not included in the following features:

- Shared JavaScript console
- Shared CSS editor
- Function debugging

All those features perform some interactions with the devices automatically, thus they should ignore inactive devices. Other features like record and replay still include inactive devices because manual interaction is needed before something happens with the device. Devices can be activated and deactivated by clicking on their name/ID that is displayed above the JavaScript console. When the device is active, the background of its name is in the color of the device, otherwise it is grey and the text color is in the color of the device. Figure 4.4 illustrates the difference between active and inactive devices. The first device is inactive, the other two devices are active.

Figure 4.4: Active and inactive devices

4.2 Emulation of Multiple Devices

XDTools allows the developer to emulate multiple devices simultaneously. The developer can either select the devices that they want to emulate from a list of existing devices or create a custom device. The developer can create a custom device just for one-time use, or they can save it to the list of existing devices for later use. This allows developers to quickly extend the devices they have access to with new devices. Devices can be assigned to one of four categories:

- Desktop devices: For simplicity, this category includes desktop PCs as well as laptops.
- Tablets
- Mobile phones
- Wearables

Although some devices may not easily be classified or fit into multiple categories, e.g. 2-in-1 devices that can either be used as a tablet or laptop by just plugging in a keyboard, those categories give a rough overview of the different types of devices. This categorization of devices also makes it easier for developers to emulate different types of devices without having to know any exact device names. By just adding some devices from each category, the developer can make sure that a large range of devices is covered, as devices in the same category typically have similar properties such as screen size and input modalities. The list of existing devices includes multiple devices from each of those categories. For the desktop devices, the list provides some typical resolutions of desktop PCs and laptops. The list for tablets and mobile phones includes most of the well-known devices in that area. The list of wearables so far only includes some smart watches, as most other wearables do not have

access to a modern web browser (yet). Figure 4.5 shows the menu for adding emulated devices while the developer is adding a device from the list of predefined devices.

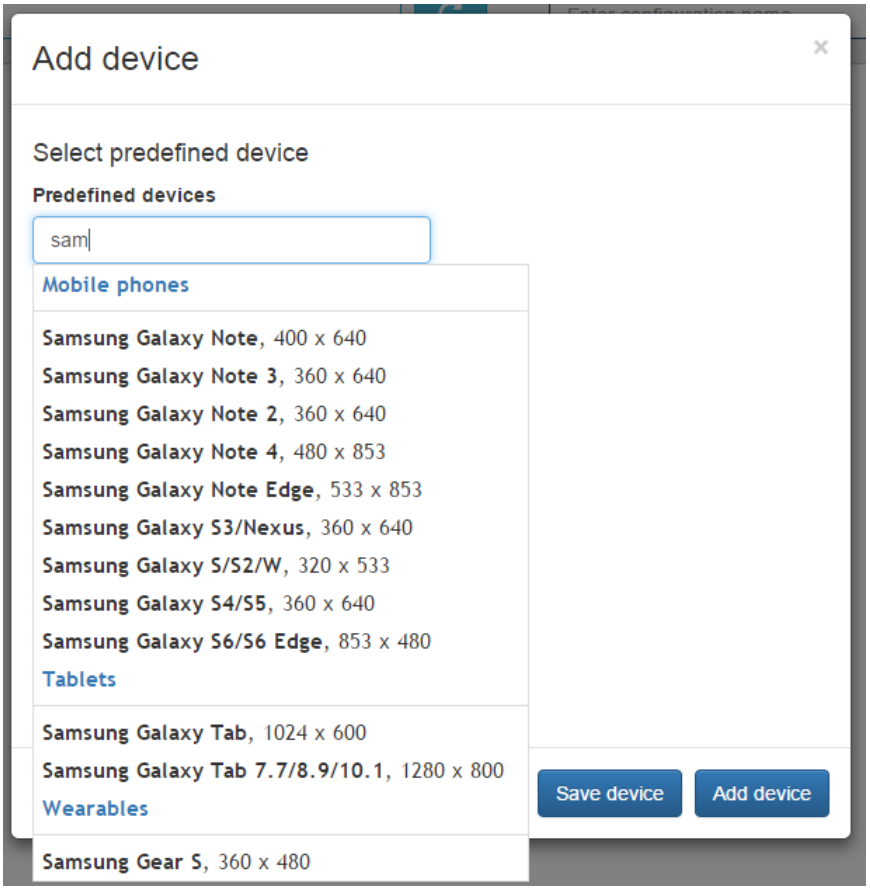


Figure 4.5: Adding an emulated device

Once the developer has created an emulated device (see Figure 4.6), the device is assigned a unique ID and they can move it to the desired location on the screen. Instead of ordering devices automatically, we chose to give the developer the freedom to choose how to place the emulated devices. This makes it easier to accurately simulate specific scenarios, e.g. a presentation room where two large screens are placed next to each other. To conquer the challenge of limited screen size, all emulated devices can be scaled up and down. Scaling a device does not change the resolution of the device and thus has no influence on the look of the application. This allows the developer to have more space available for devices. If the developer has difficulties performing an interaction on a device because it is scaled down, they can just scale it up again. However, dynamically changing the resolution of an emulated device is also possible. The developer can continuously increase or decrease the resolution of the device to immediately see what the application looks like on different resolutions. Finally, the developer can also change the layer of the device. Changing the layer of a device essentially moves the device to the front or back, thus devices can overlap each other and the developer can move the device that they want to use to the front.

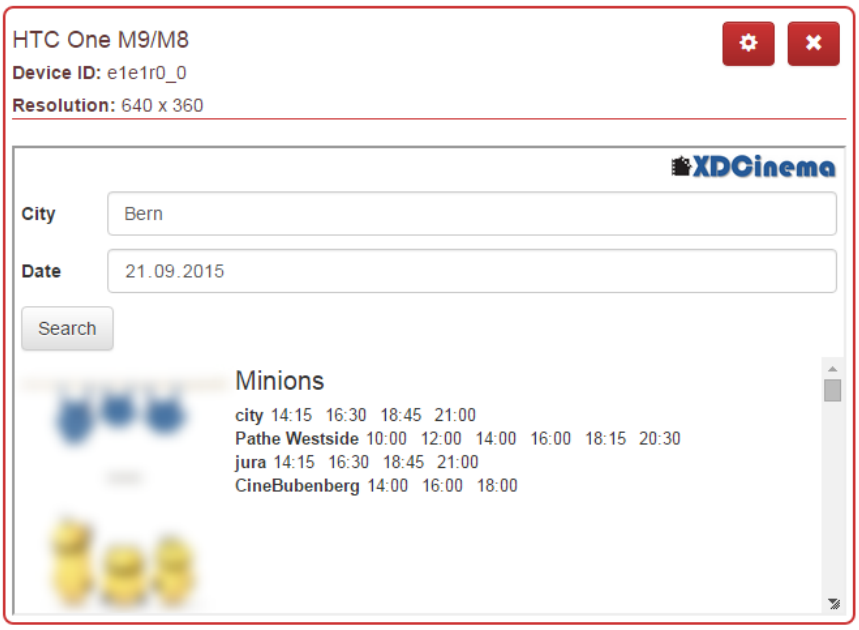


Figure 4.6: An emulated device

Apart from its unique ID, each device also has a unique color. The border of the emulated device is colored with this color and the color is used in multiple other places for identifying the device. The devices also have a settings menu. In the settings menu, the developer can configure the following things:

- The URL of the device.
- The scaling of the device.
- The orientation of the device.
- The layer of the device.
- The device can be refreshed.
- The developer can inspect the HTML of the device.
- The developer can connect the device to another device by choosing the other device's ID from a drop-down menu.

A device's settings are not constantly used by the developer and displaying them at all times would occupy valuable screen space. Thus, the settings menu can be extended and collapsed by clicking a button. An example of a settings menu can be seen in Figure 4.7.

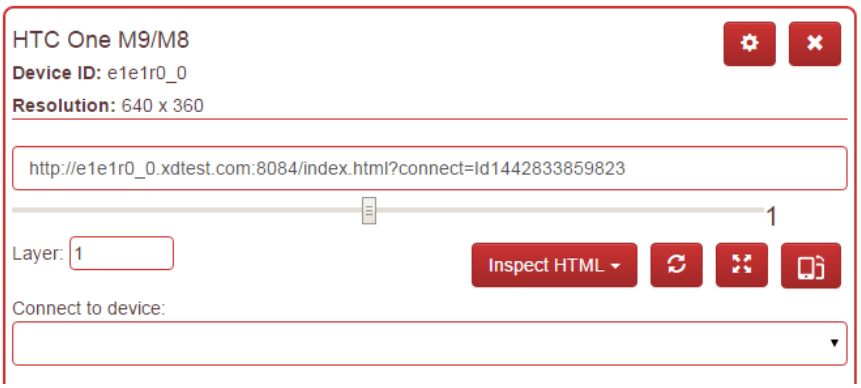


Figure 4.7: Settings menu of an emulated device

Finally, XDTools also includes a mechanism that prevents the sharing of local resources between emulated devices.

4.3 Easy Integration of Real and Emulated Devices

QR codes have become increasingly popular over the last few years and almost all devices nowadays are equipped with at least one camera. Thus, XDTools includes a QR code that can be displayed and scanned by developers to connect a real device to XDTools. As a fallback mechanisms, devices can also be connected by typing a URL in the browser of the device. This makes it easy and efficient to connect a large number of real devices to XDTools. When the URL is opened, the device loads the application under test. The real devices only show the application under test and no additional interface elements. The developer can use the application on the real device, but everything related to testing and debugging is coordinated through the developer's machine running XDTools. Each connected device is represented by a proxy within XDTools. The proxy of the real device also contains a settings menu similar to the one of an emulated device. However, the settings menu is missing a few things:

- Scaling of the device: There is no need to scale real devices, as the application under test is shown on the device itself.
- Switching orientation: The orientation of a real device can be switched on the real device itself by simply rotating the real device.
- Inspecting the HTML: The developer's machine has no access to the HTML of the real device, thus it cannot be inspected.

Figure 4.8 shows the settings menu of a remote device. The settings menu only contains the interface elements for setting the URL of the device, refreshing the device, changing the layer of the device and connecting the device to other devices.

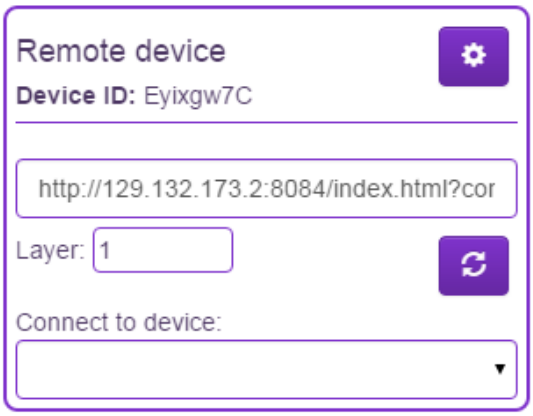


Figure 4.8: Settings menu of a remote device

The proxy of the real device can be moved around just like the emulated devices and its settings menu can also be collapsed and extended. Furthermore, each real device also has a unique ID and color for easy identification.

The developer can use emulated devices alongside real devices. Thus, XDTools allows the developer to test their application on only emulated devices, only real devices, or both at the same time. This flexibility makes it easy to test a large number of different scenarios.

4.4 Easy Switching of Device Configurations

XDTools allows the developer to create some emulated devices, save this device configuration for later, and then re-use it. A device configuration stores the following information for each emulated device:

- The device's name and ID.
- The resolution of the device.
- The position of the device.
- The scaling of the device.
- The layer of the device.

Storing device configurations makes it easy for the developer to re-use device configurations and switch between different device configurations efficiently. Thus, testing a cross-device application with many different device scenarios can be done without much effort. A user can just load one device configuration, try out the application and switch to the next device configuration if everything works as expected. The developer can also create a device configuration where only the static devices used with the application under test are saved in the configuration and dynamically add more devices.

If the developer wants to save a device configuration, they can type the name of the device configuration into an input field and then click the save button to save the current device

configuration under that name. If they want to load a device configuration, they can type its name into the same input field and click the load button to load the device configuration with that name (if it exists). Autocompletion is used for providing suggestions for existing device configurations that could be loaded (see Figure 4.9).

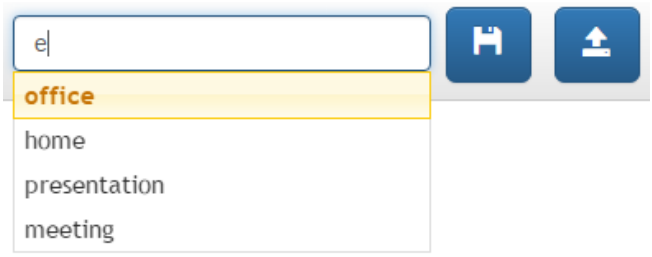


Figure 4.9: Saving/Loading Device Configurations

4.5 Integration with Debugging Tools

In the previous chapter, we described the requirements for a tight integration of XDTools with some of the most useful browser debugging tools. In the following, we will describe how the integration of the debugging tools is realized in XDTools.

4.5.1 Shared JavaScript Console

XDTools includes a JavaScript console that is shared between all emulated and real devices. Each emulated and real device forwards all logging messages to the console. The console then aggregates the logging messages from all devices. Each message is displayed in the color of the device that it was sent from. This color-coding makes it trivial to identify the device a message originated from. Apart from forwarding logging messages, the devices also forward their JavaScript errors. Whenever a JavaScript error occurs, the JavaScript error message is sent to the console, together with the stack trace if available. When a JavaScript error with stack trace is received by the console, the stack trace is split into multiple lines where each line represents one entry of the stack trace. By default, the developer only sees the error message itself without the stack trace. By clicking on a button next to the error message, the developer can extend and collapse the stack trace. Figure 4.10 shows what such a stack trace looks like in the JavaScript console.

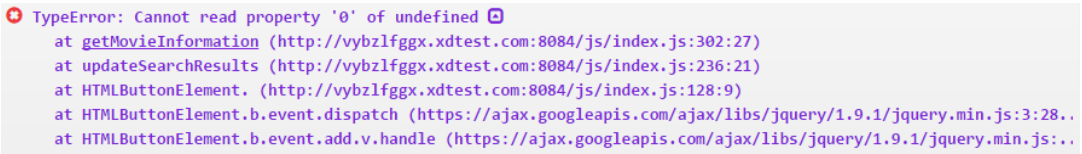


Figure 4.10: Stack trace

For easier identification of the type of a message, all messages are assigned to one of four categories:

- Warnings: This includes all warning logging messages.
- Errors: This includes all error logging messages and JavaScript errors.
- Info: This includes all info logging messages.
- Log: This includes all other types of messages.

Depending on the type of message, a different symbol is used in front of the message when it is displayed in the shared JavaScript console. This makes it easy to see if a message is an error or just a simple logging message.

The large number of messages that are displayed in the console due to the aggregation of many devices makes it difficult to look for a specific message. To solve this problem, the messages in the console can be filtered. The messages in the console can be filtered by the type of message, e.g. show only error messages, or by text. If the console messages are filtered by text, all messages that do not contain the text to filter by anywhere are hidden. If the filter is removed, the hidden messages are shown again. If a device is deactivated, it does not forward any messages or errors anymore and all existing messages are filtered out until the device is activated again.

The console also allows sending commands to multiple devices. If the developer wants to send a command to the devices, they can type the command into an input field below the console and press the enter key. The developer can either send the command to all devices or they can deactivate some devices and only send the message to a subset of devices. Being able to send a command only to a subset of devices is important in a cross-device setting because some commands might only make sense on some devices and executing them on all devices could lead to potential errors or a decrease in performance. The return values of commands are also displayed in the console, again color-coded to match the device they came from. Thus, otherwise complicated tasks like checking the value of a global variable on all devices become trivial. The console works exactly the same for emulated and real devices. This aggregation of console outputs from both emulated and real devices at least partially eliminates the need for remote debugging.

The shared JavaScript console also has a history function similar to the console in Chrome DevTools. By using the arrow keys, the developer can navigate through the history and call previous commands again.

The screenshot in Figure 4.11 shows the shared JavaScript console including a few examples of how to use it. First, the developer tries to call a function but makes a typo. Because the function with the typo in the name does not exist, an error message is displayed by all devices that received the command. The developer then notices the typo, types the correct function and sees the return values of the function (the coordinates of a cinema) on the devices. Finally, the developer wants to see which roles have been assigned to the devices. They type the name of the variable that stores the roles and the value of this variableis shown in the console for all devices.

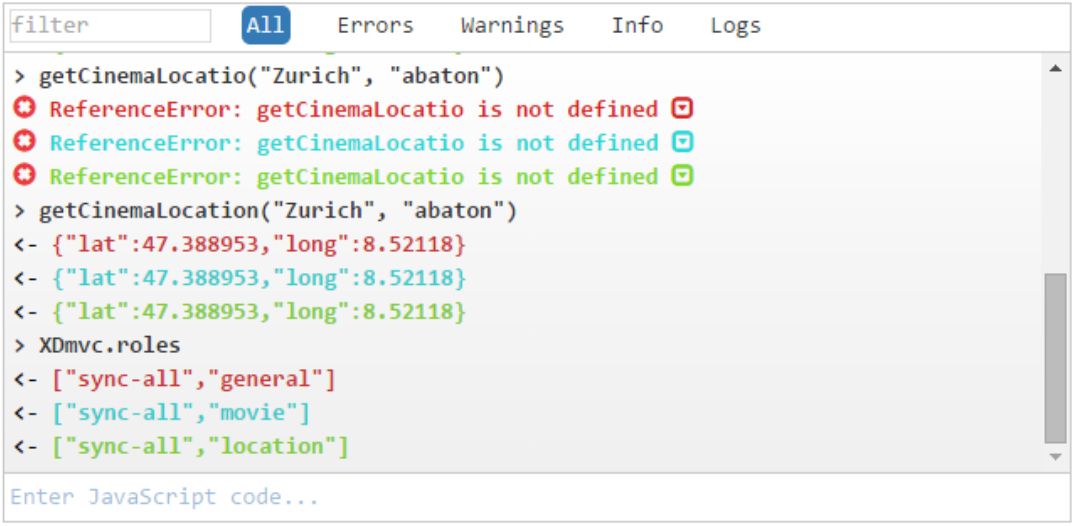


Figure 4.11: Shared JavaScript console

4.5.2 Function Debugging and Inspection

Function debugging allows developers to debug a function on multiple emulated devices without having to add breakpoints on each device individually.

In an area at the bottom of XDTools, the developer has access to an input field where they can type the name of a function that they want to debug. Below the input field, the list of functions that are currently being debugged is shown. Whenever one of the debugged functions is called on any device, the debugger pauses at the beginning of the function and the developer can perform their debugging actions. The device on which the function is called is highlighted until the function call returns. This makes it easy to identify the device that is currently being debugged.

Figure 4.12 shows a screenshot while a function is being debugged. In the screenshot the highlighted device, the Chrome DevTools with the source code of the debugged function and the list of debugged functions can be seen.

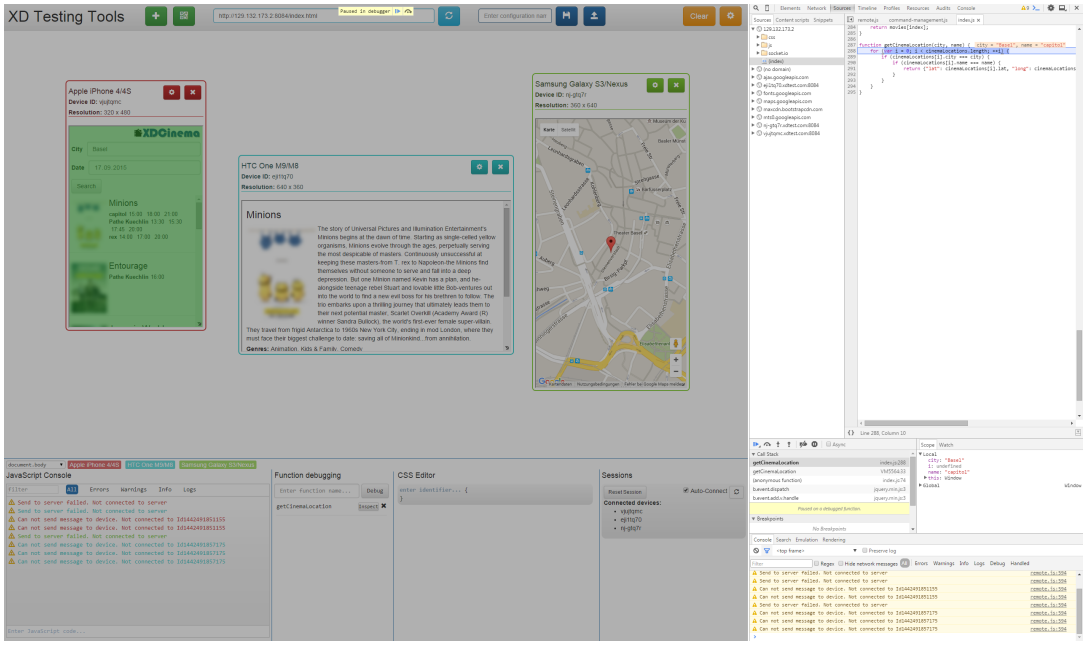


Figure 4.12: The complete interface while debugging a function

If the developer is done debugging a function, they can remove it from the list of debugged functions.

The developer can also just inspect the source code of the function without it being called by clicking on a button next to the function name in the list of debugged functions. If this button is clicked, an emulated device is picked at random and the source code of the function is opened on this device.

Unfortunately, function debugging only works on emulated devices. Chrome's DevTools do not have access to the JavaScript code of a connected real device, thus debugging the function using the DevTools is impossible without remote debugging.

4.5.3 HTML Inspection

XDTools allows the developer to directly jump into the HTML of an emulated device. Inside the settings menu of each emulated device, a button can be clicked to inspect the HTML of the device. Clicking this button opens the `body` element of the device in the Chrome DevTools.

HTML inspection is only available on emulated devices for the same reasons that we already mentioned before.

4.5.4 Shared CSS Editor

XDTools also includes a custom CSS editor. The CSS editor is designed to feel similar to the CSS editors typically provided by browsers. The developer can specify a selector and then add some rules that are applied to HTML elements that match the selector. The CSS rules are applied to all active devices, including both real and emulated devices. All CSS

rules can be deactivated and activated again, edited, or removed completely. This allows the developer to quickly change the CSS of multiple devices and immediately see the result. This is considerably less effort than adding rules to all devices individually or editing the CSS file, saving it and reloading all devices.

The CSS editor also has some autocomplete functionality: If the developer starts typing the name of a property, the property is automatically completed using the first CSS property name that matches the text typed by the developer (see Figure 4.13).

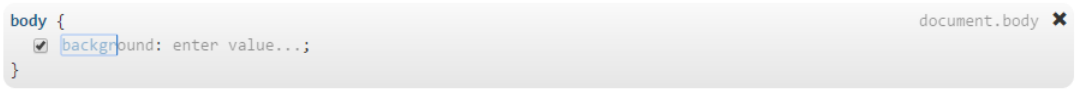


Figure 4.13: Autocompletion in the CSS editor

Figure 4.14 shows a screenshot of the CSS editor in action. In this screenshot, some CSS rules are added to the `body` element of all devices.

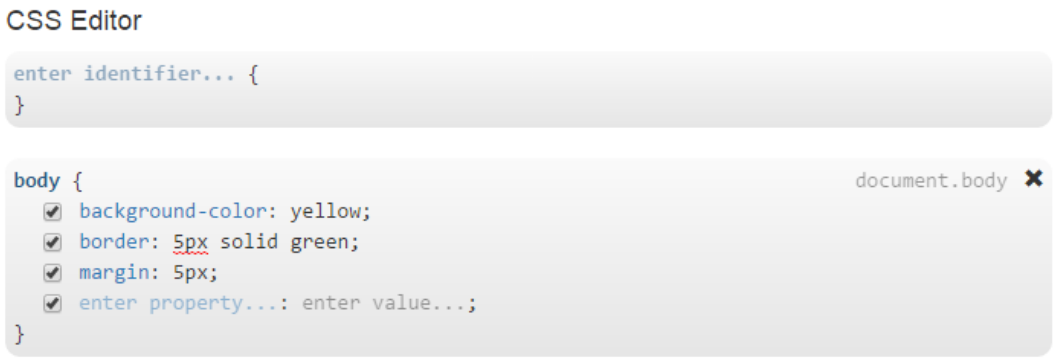


Figure 4.14: Shared CSS Editor

In Figure 4.15, the effects of the CSS shown in the screenshot before are shown.

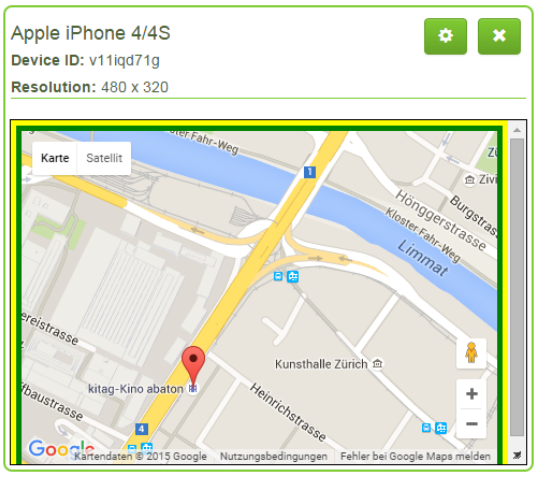


Figure 4.15: CSS applied to emulated device

4.6 Automatic Connection Management

The devices in XDTools can either be connected automatically or manually. Each set of connected devices represents a session. For each session, a checkbox allows the developer to toggle on or off auto-connect. When the first device is created or connected, auto-connect is switched on by default. If auto-connect is on, all newly created and connected devices will automatically be connected to that session. Thus, if the developer simply adds or connects a number of devices, they will automatically all be connected. If the developer wants to have multiple sessions, they can turn off auto-connect and then add more devices. Each device also has a drop-down menu in its settings menu for manually connecting to other devices. Only one session can have auto-connect enabled; if the developer switches on auto-connect for one session, it is switched off for all other sessions.

All current sessions are displayed at the bottom of XDTools. Each session displays all connected devices (see Figure 4.16). The developer can also refresh all devices in a session or reset a session. Resetting a session assigns new IDs to all devices and thus erases the local resources of the devices.

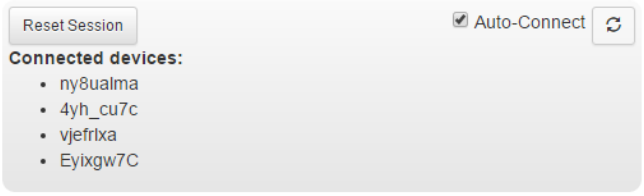


Figure 4.16: A device session

So far, automatic connection is only possible for applications that connect via URL. However, the URL required for connecting depends on the application under test. XDTools provides a custom connection function that can be adjusted by the developer such that it returns the appropriate URL.

4.7 Coordinated Record and Replay

XDTools allows the developer to record a sequence of interactions on a device and replay them. Recording can be started on one device at any time. Once recording has started, the developer can perform the desired interactions on the device. After finishing recording, the interactions can be replayed on the same device, moved to other devices, or saved for later. Furthermore, event sequences can be cut into multiple parts. The timing of the replays can be configured arbitrarily by dragging and dropping event sequences. This allows the developer to configure replays in many different ways: They can be executed in parallel, one after another, or anything in-between. The replays can be started on all devices simultaneously or on one device at a time. This makes it easy to simulate multiple users and devices. Since the events contained in an event sequence are performed by a real user, i.e. the developer, the timing of the individual events in an event sequence is very realistic.

After a device finishes recording, the event sequence is visualized (Figure 4.17).

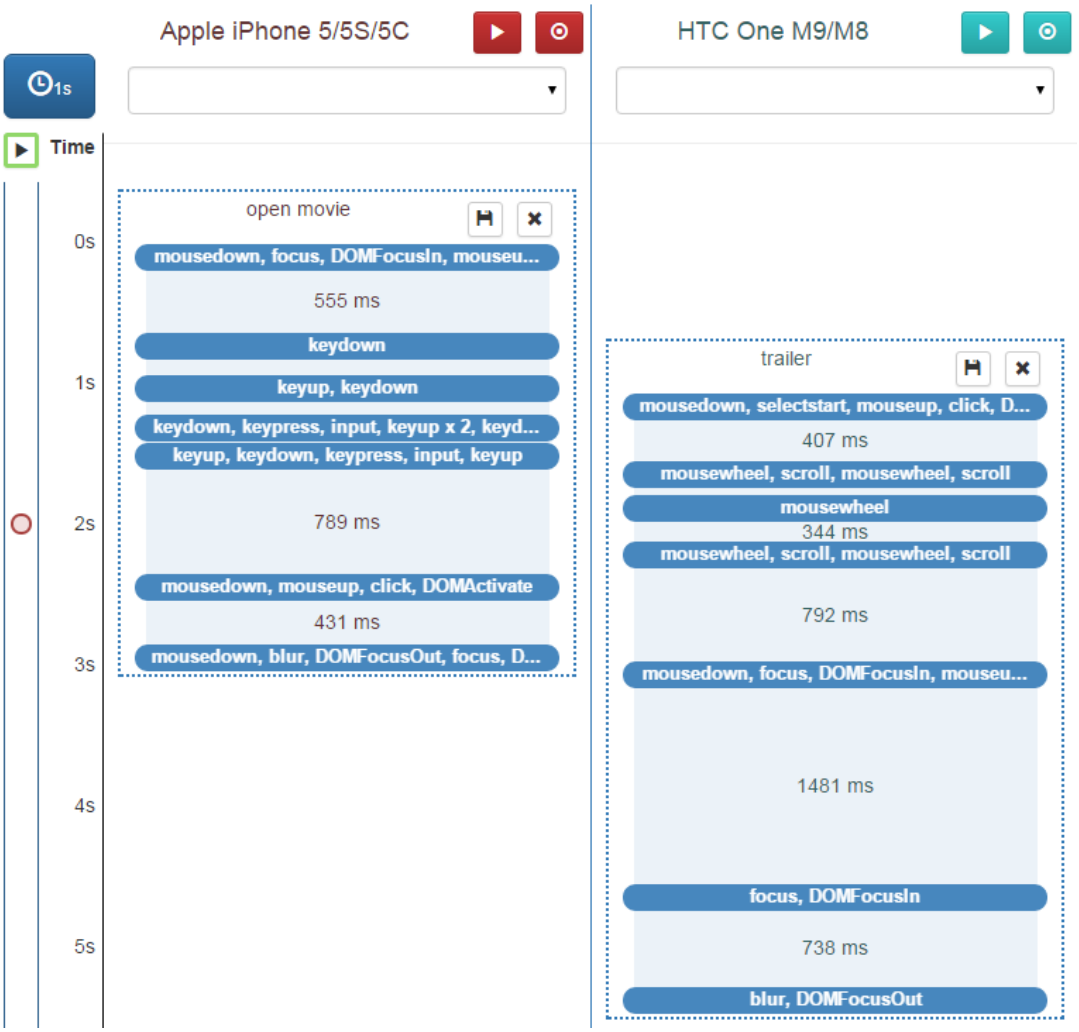


Figure 4.17: Record and Replay

The developer can also add breakpoints to the replay. Next to the event visualizations, a timeline is shown as well as a narrow empty column where the developer can click to add a breakpoint at a certain point in time. As soon as all events that occur before this point of time have been replayed, the replay pauses and the developer can inspect the state of the devices. If a breakpoint is reached, the breakpoint that was reached is highlighted. After the developer chooses to continue, replay is continued until the next breakpoint is reached. After the last breakpoint has been reached, the replay continues until all events have been replayed. The developer can also add breaks of one second between events to delay all following events by clicking on a button and dragging the mouse to the place where the break should be inserted. This allows the developer to spend some more time looking at the application under test before the replay continues without having to set breakpoints.

5 Implementation

The following sections describe the architecture of XDTools, the technologies used for implementing XDTools as well as some implementation details.

5.1 Architecture

The architecture of XDTools is illustrated in Figure 5.1. XDTools consists of two applications: a main application that runs on the developer's machine and a helper application that runs on the real devices. Both the main application and the helper application are pure web applications and are provided by a server that is also responsible for forwarding communication between the main application and the helper applications.

Additionally, a local DNS server runs on the developer's machine. It is responsible for creating different subdomains for the emulated devices. The DNS server is required for properly testing and debugging on emulated devices, but if only real devices are used, it is not needed and can be disabled in the options of XDTools.

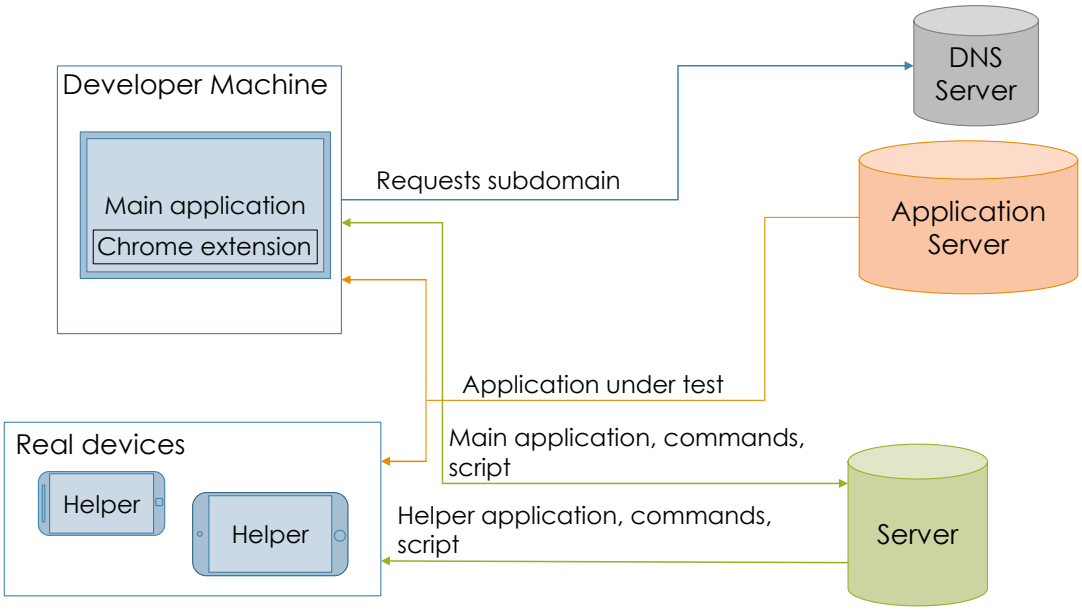


Figure 5.1: Architecture of XDTools

Furthermore, a Chrome extension runs in the browser of the developer's machine. This extension is needed for function debugging and inspection as well as HTML inspection. The Chrome extension communicates with the main application via the server.

Finally, some web server needs to provide the application under test. However, any web server can be used and XDTools does not place any restrictions. The only requirement is that the developer injects a small script that is hosted on our server at the top of each HTML page of the application. The script is responsible for performing many actions that would otherwise be difficult, such as executing JavaScript, recording events and adding CSS to the application under test.

5.2 Choice of Technologies

Since XDTools includes desktop devices as well as mobile devices, it should be platform-independent. Thus, XDTools is implemented using standard web technologies, i.e. HTML5, CSS3 and JavaScript. This allows XDTools to be run in any modern web browser without requiring installation of any additional software. The fact that cross-device applications are also typically web-based makes web technologies even more suited for XDTools. XDTools does not rely on any database, instead HTML5 Local Storage[1] is used for the data that we want to store. XDTools only need to store three things:

- Custom devices for emulation that were added by the developer.
- Device configurations.
- Event sequences for replaying.

[1] `http://www.w3schools.com/html/html5_webstorage.asp`

All those things only need to be accessed by the developer that created them. Thus, it makes sense to only store the data locally. Furthermore, the amount of data that needs to be stored is rather low.

5.2.1 Server Side

On the server side, XDTools uses Node.js[2]. Node.js is a JavaScript runtime that uses asynchronous I/O with an event-driven programming model. It is lightweight and scales well with a large number of connections. Furthermore, it has the advantage that both the client and server side are implemented using JavaScript. Thus, no data conversions are required. In XDTools, fast communication between the desktop PC and the connected real devices is required and Node.js fulfills this requirement. Node.js' package ecosystem, npm[3], provides a large number of modules which extend the capabilities of Node.js. XDTools uses the following modules:

- Express[4]: Express is a minimal and flexible web application framework that provides a set of features for web and mobile applications. In XDTools, Express is responsible for serving the content to the clients.
- shortid[5]: shortid generates short URL-friendly unique IDs and is responsible for generating the device IDs for the emulated and real devices in XDTools. As each emulated device gets a subdomain based on its ID, shortid is ideal for our purpose.
- Socket.IO[6]: Socket.IO enables real-time bidirectional event-based communication. If available, it uses HTML5 WebSockets, otherwise it uses fallback mechanism like AJAX[7] long-polling. In XDTools, Socket.io is used for the communication between the server and clients.

Socket.io is better suited for XDTools than AJAX for multiple reasons: First, we need to be able to send push data to clients from the server when sending commands to devices and AJAX is intended to allow clients to pull data from the server. Also, data transfer with AJAX comes with a significant overhead because HTTP headers are transmitted with each piece of data. As we mentioned before, fast communication between the server and the devices is a requirement for XDTools and AJAX would not fulfill this requirement sufficiently. However, AJAX is used for communicating with the DNS server.

DNS Server

XDTools requires a DNS server for registering the subdomains for the emulated devices. It should be easy to dynamically register new subdomains to the DNS server. Furthermore,

[2] https://nodejs.org/en/
[3] https://www.npmjs.com/
[4] http://expressjs.com/
[5] https://github.com/dylang/shortid
[6] http://socket.io/
[7] http://www.w3schools.com/ajax/

the DNS server should forward any unknown domains to the standard DNS server. rainbow-dns[8] fulfills both those requirements. It is a Node.js-based DNS server with an HTTP API that makes it trivial to register new domains from the main application of XDTools. When starting the server, the IP address of the standard DNS server can be passed as an argument, allowing rainbow-dns to forward unknown domains to the standard DNS server.

5.2.2 Client Side

The client side of XDTools is implemented using JavaScript with jQuery[9]. jQuery allows easy selection and modification of HTML elements as well as easy event handling, features that are crucial to XDTools. As an additional helper, we use jQuery UI[10]. jQuery UI is mainly used for the autocomplete functionality used in several places and for easy resizing of the individual interface components of XDTools. It is also used for resizing the emulated devices to change their resolution. The Twitter Bootstrap[11] framework is used to give XDTools a nice and uniform look-and-feel. It is also used for facilitating the implementation of things like modal boxes and tabs.

XDTools uses QR codes for connecting real devices to the server. We use the jQuery-based QR Code library jQuery.qrcode[12] for generating those QR codes. Using jQuery selectors, the content of an HTML element can be set to a QR code with the desired content and size.

HTML5 Drag and Drop[13] is used for allowing accurate positioning of devices and timing of event sequences. HTML5 `postMessage` is used for communication between the application under test in the `iframe` and XDTools. Thus, no additional library for communication is needed inside the application under test.

The script that is injected into the application under test is implemented in pure JavaScript. Using libraries like jQuery inside this script could lead to conflicts with other libraries in the application under test or different versions of jQuery. Also, loading jQuery into the application creates some additional overhead and might not be what the developer of the application wants.

The Chrome DevTools extension is implemented using JavaScript. It uses the Command Line API[14] for accessing the functions needed to inspect HTML, debug functions, and more. The extension also uses Socket.IO for communicating with the server.

5.3 Emulation of Multiple Devices

Devices are emulated using `iframes`: Each emulated device is represented by its own `iframe` that loads the application under test. However, loading the same domain inside multiple `iframes` would lead to the sharing of local resources between emulated devices.

[8] `https://github.com/asbjornenge/rainbow-dns`
[9] `https://jquery.com/`
[10] `https://jqueryui.com/`
[11] `http://getbootstrap.com/`
[12] `https://larsjung.de/jquery-qrcode/`
[13] `http://www.w3schools.com/html/html5_draganddrop.asp`
[14] `https://developer.chrome.com/devtools/docs/console-api`

In order to prevent this, a unique subdomain based on the ID of the device is registered with the local DNS server. All subdomains point to the application under test, but as the application is accessed through different domains, no local resources are shared between the emulated devices. Domains that are unknown to the local DNS server are forwarded to the standard DNS server.

The resolution of each `iframe` corresponds to the resolution of the device that it represents in CSS pixels. Currently, only the resolution of the target devices is emulated, but in the future, XDTools could be extended to support emulation of other aspects. The scaling of the device is set using the CSS property ”transform”, e.g. by setting ”transform: scale(0.5)” to scale a device to half its usual size. Using the ”transform” property does not change the resolution of the `iframe` of the emulated device, only the space that the `iframe` occupies. Thus, the layout of the emulated device remains intact even if it is scaled down. If the developer instead wants to change the resolution of a device, they can click on the bottom-right corner of the `iframe` and drag to increase or decrease the resolution.

The devices can be moved around using HTML5 Drag and Drop. However, we cannot just make the devices `draggable` by default. The developer should not be able to move the device if they click on the `iframe` because otherwise interaction with the `iframe` would be difficult. Thus, we chose to only make devices `draggable` if the developer clicks on the header of the device. For this reason, we assign a click handler to each device that checks if the developer clicked on the header of the device. If they did, we make the device `draggable`.

When the developer starts dragging the device, the device ID is assigned as data to the event. However, the device ID is not the only required data. If the developer drags the device to a certain position, we cannot just assign this position to the device. If we did, the top left corner of the device would be at the position where the device was dropped, but the developer probably did not click on the top left corner of the device when they started dragging the device. Thus, we need to compute the offset of the click from the top left corner of the device and adjust the position accordingly when the device is dropped. For example, if the developer clicks on the device 10 pixels from the top and drops the device x pixels from the top of the device area, we need to assign the position x - 10 to the device. For achieving this, we compute the offset when the developer clicks on the header of the device and set the offset as data to the event.

5.3.1 Color Generation

The primary goal of the device colors was to make it easy to distinguish devices, thus their colors should be as distinct as possible. For achieving this, we use the HSL color space[15]. Finding distinct colors is a much simpler task in the HSL color space compared to the RGB color space because only one of the three color components has to be modified. In the HSL color space, colors consist of the three values hue, saturation and lightness. The colors of the devices should all have the same saturation and lightness, thus the only value that needs to be determined is the hue. The hue can have any value between 0 and 360. Since our goal is to have colors that are easily distinguishable, their hues should be as different as possible. A

[15] `https://en.wikipedia.org/wiki/HSL_and_HSV`

simple algorithm can be used for determining the color of the next device:

1. If no colors have been assigned yet, assign the color 0/360.
2. If only one color has been assigned, assign the color 180.
3. If at least two colors have been assigned: Order the assigned colors by their values, compute the maximum distance between any two neighboring colors and assign the color in-between.

5.4 Easy Integration of Real Devices

Real devices can be connected by scanning a QR code or opening a URL. When the URL is opened, the device loads the helper application. Once the helper application is loaded, the application under test is shown in a full-screen `iframe` on the device. If the developer issues any command in the main application, e.g. refreshing the `iframe` of the real device, the command is first sent to the server and then forwarded to the target device.

5.5 Easy Switching of Device Configurations

Device configurations are stored in Local Storage. The Local Storage stores an array containing the names of all saved device configurations. This array is used for the autocomplete functionality and for retrieving the actual device configuration. The actual device configuration is an array of devices. For each device, the name, width, height, device pixel ratio, scaling, layer and position is stored. It does not make any sense to store real devices because they might not be available when the device configuration is loaded and they need to be reconnected manually anyway by opening the appropriate URL, thus only emulated devices are stored in device configurations. If the developer wants to load a device configuration, an ID is requested for each device, then the device is created and its scaling and position are set according to the values in the Local Storage.

5.6 Integration with Debugging Tools

The following subsections describe the implementation of the parts that were adopted from the browser's debugging tools.

5.6.1 Shared JavaScript Console

The following types of logging messages are aggregated in the shared JavaScript console:

- `console.log`
- `console.debug`

- `console.warn`
- `console.info`
- `console.error`
- `console.count`
- `console.dir`
- `console.assert`

In the script that is injected into the application under test, those functions are overwritten by a new function and the original functions are stored. The new function performs the following two actions: First, it sends the content and type of the logging message to our server. Second, it calls the original logging function. Thus, the logging messages are still displayed in the browser console, but are forwarded to our console in addition.

Forwarding JavaScript errors is implemented by assigning an event handler to JavaScript errors using `window.onerror`. Whenever a JavaScript error occurs, the event handler is called. Inside the event handler, the JavaScript error message is sent to our server, together with the stack trace if available.

Commands that are sent to devices are executed by calling `eval` with the command string as argument.

5.6.2 Function Debugging and Inspection

Function debugging and inspection requires the Chrome extension. When the developer opens the Chrome DevTools, the function debugging area displayed. When the developer adds a new function to debug, a command is sent to each activated emulated device. If a device receives the name of a function to debug, it overwrites the function to debug by a new function and stores the original function. The function that overwrites the original function performs the following actions:

1. It highlights the device with a semi-transparent green overlay.
2. It calls the original function and stores the return value.
3. It removes the highlighting from the device.
4. It returns the return value.

After overwriting the original function, the device sends a message back to the main application that it is ready for debugging. The main application then sends a command to the Chrome extension with the name of the variable that stores the original function. The Chrome extension receives the command and the URL of the device the message originated from. The Chrome extension then calls the ”debug” function with the function name and the URL of the frame that represents the device.

If the developer is done debugging a function, they can remove it from the list of debugged functions and the original function is restored inside the script. Then, a command is sent to the DevTools extension, informing it to stop debugging the function.

The functions that are debugged run inside an `iframe` and the URL of the `iframe` needs to be passed as a parameter. However, when a new device is added, the hook for the `iframe` of the new device does not exist yet and the Chrome DevTools need to be closed and re-opened. Similarly, if the URL of a device changes, the hook is lost. Unfortunately, there is no way of automatically re-opening the DevTools. For this reason, a red overlay containing a warning text is shown whenever a URL changes or a new device is added. The warning can be seen in Figure 5.2.

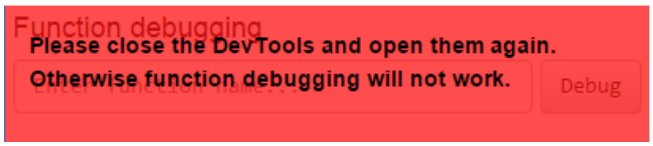


Figure 5.2: Warning that is shown for function debugging

5.6.3 HTML Inspection

HTML inspection also requires the Chrome extension and thus only works on emulated devices. If the button to inspect the HTML is clicked, a command is sent to the DevTools extension, telling it to open the `body` element of the frame with the URL of the device. The HTML inspection view in the Chrome DevTools then jumps directly to the body of the HTML of the device.

5.6.4 Shared CSS Editor

Inside the injected script in the application under test, a stylesheet is created and added to the DOM tree. Whenever the developer adds a new rule to the CSS editor, a command is sent to all active devices and the rule is added to an array containing all CSS rules. Whenever the array with the CSS rules changes, the stylesheet is rewritten. Retrieving the appropriate rule from the stylesheet and modifying it is difficult and time-consuming, simply rewriting the stylesheet is much more convenient, especially since we do not expect developers to add hundreds of rules to the shared CSS editor. If a rule is disabled, it is removed from the stylesheet; if it is enabled again, it is re-added to the stylesheet.

For the CSS autocomplete functionality, we need to get all available CSS properties. The available CSS properties are retrieved using ”document.body.style” which contains all available CSS properties, even if they are not set. The CSS properties of `document.body.style` are then converted to an array of available CSS properties.

5.7 Automatic Connection Management

In the script that is injected into the application under test, a connection function is available. When a device wants to connect to another device, a command is sent to that device. That device then calls the connection function. The connection function returns the URL that has to be loaded to connect to the device. The URL is then loaded on the device that wants to connect.

The connection function has to be implemented by the developer.

5.8 Coordinated Record and Replay

Recording interactions is done by assigning event handlers for all relevant events that we want to record. It is important that no other script receives the events before us, because otherwise they could modify the page or even stop event propagation. For this reason, the event handlers are assigned as soon as the injected script is loaded and the script has to be placed at the top of the page. However, the event handlers do not do anything as long as recording has not started yet. When the developer starts recording, all events are logged to an array. After finishing recording, the array of events is sent to the main application.

Furthermore, all event handlers are capturing and are assigned to the document itself. Since no other element in the DOM hierarchy can be above the document element, the document element is always the first element that receives an event with capturing event handlers. After finishing recording and before sending the event sequence back to the server, all circular structures are removed. All other event properties are kept so the events can be replayed accurately.

The event sequences should be replayable on other devices than the recording device, therefore some way of determining the target of the event is required. Whenever an event handler is triggered, the DOM hierarchy is traveled up from the target of the event until an ID is encountered. Using this ID, we can determine the target element by taking the appropriate child until the target element is reached. From this, we can derive a hierarchy that describes how to reach the target element on a device and replaying the event on another device becomes possible.

When visualizing event sequences, each event requires some space. Thus, events that happen close together in time are grouped together. If events were not grouped, the visualization of the event sequence would not represent the true timing of the events: Even though two events might occur almost simultaneously, they would both require some space when visualized and thus give the impression that some time passed between them. The following algorithm is used for grouping events:

1. Take the first unprocessed event.
2. Take all events that happen within x milliseconds of that event and group them together.
3. Visualize the grouping of events.
4. Go to Step 1.

When the developer starts replaying, the event sequences assigned to a device are sent to that device along with the list of breakpoints. The events that happen before the first breakpoint are then triggered using `setTimeout`. After all events have been triggered, a message is sent to the main application, telling it that a breakpoint has been reached. After the developer chooses to continue, a message is sent to all replaying devices telling them to continue event replaying.

All events are replayed using the properties recorded in the event sequences. Using the hierarchy computed when recording, the target element of the event is determined. For most events, replaying with the properties logged when recording is enough for realistically reproducing the events. However, some events require a bit more work. Replaying key events is possible, but no text is written into input fields. In addition to the key events, we also need to trigger a text event using the correct char code. Scrolling events also do not scroll. Thus, we also log the scrolling position after a scrolling event occurs and manually set it after replaying the event. Furthermore, the backspace key cannot be reproduced with neither text events nor key events. Thus, we manually determine the position of the caret if the backspace key is pressed and delete the last character before the caret when replaying.

5.9 Integration with Polymer and Shadow DOM

Although XDTools works with all web-based cross-device applications, one of the initial goals was to make it work with applications developed with XD-MVC. XD-MVC can either just be used as a JavaScript API or it can be used together with Polymer[16]. Polymer provides developers with the option to use Shadow DOM[17]. Shadow DOM refers to the ability of the browser to include a subtree of DOM elements into the rendering of the document, but not into the main document tree. Thus, it can encapsulate components and prevent them from being reached by traditional tree walking functions such as `childNodes` and `firstChild`. We must assume that some applications implemented with XD-MVC use Polymer and Shadow DOM and XDTools should support those applications as well. However, a few modifications are required for supporting both Polymer and Shadow DOM. First, determining the target of an event can become more difficult because the element might not be accessible from the global scope. Second, debugging functions that are not accessible from the global scope should be possible and we need some way of accessing those functions. Furthermore, we want to be able to add CSS to elements that are not accessible from the global scope.

5.9.1 Determining Event Targets

An element can define Shadow DOM by attaching a shadow root to itself. The Shadow DOM can be accessed using the `shadowRoot` property of the element. Without Shadow DOM, finding the nearest element with an ID was enough for determining the path that leads to the target element of an event. With Shadow DOM, we need to additionally consider all shadow roots that lead from the `body` element to the target element. The `path` property of the

[16] `https://www.polymer-project.org`
[17] `http://www.w3.org/TR/shadow-dom/`

event contains all elements that were visited by the event. We adjusted our algorithm in the following way:

1. Take the lowest (in the DOM hierarchy) element in the `path` that has not yet been processed.
2. Determine how to reach this element from the next higher element in the path:
   (a) Check if the element has a parent node. If not, it must be the hierarchically highest element inside a shadow root. Thus, it can be reached by accessing the `shadowRoot` property of the higher element.
   (b) Check if the element has an ID. If yes, it can be reached using the ID.
   (c) If the element has a parent element but no ID, it can be accessed by accessing the appropriate child of the parent element.
3. Go back to Step 1.

Using this hierarchy, the target element can already be reached, but because of elements with ID, some steps can be skipped. Inside each shadow root, we look for the lowest element with an ID. All elements inside the shadow root but above that element can be skipped and the ID can be accessed directly. If no element has an ID, we have to keep all elements. The same can be done for the path from the `body` element to the highest shadow root and for the path from the lowest shadow root to the target element. Using this algorithm, the shortest path to the target element is computed. For applications that do not contain any Shadow DOM, the algorithm yields the same result as the standard algorithm described earlier. The computed hierarchy can be used to determine the target of the event on other devices by accessing shadow roots instead of IDs or children when appropriate.

This modified algorithm allows us to record and replay events on applications that use Shadow DOM.

5.9.2 Determining Scopes

There are two things to be considered: First, there may be shadow roots in the application. Second, the developer can define functions for their custom Polymer elements that need to be accessed using the Polymer element. In principle, those functions can be accessed using the appropriate path through the DOM tree, but determining this path is not trivial. Thus, we want to automatically determine all different scopes on which functions can be defined, i.e. all shadow roots and Polymer elements and the path through the DOM tree that leads to those scopes. We implemented the following algorithm for determining the scopes:

1. Add the `body` element to the processing queue.
2. As long as the processing queue is not empty, take the first element and perform the following actions:
   (a) If the element has the property `shadowRoot`, we found a new scope and add it to the list of scopes, together with the path that determines how to reach the scope. Then we add the shadow root of the element to the processing queue.

(b) If the element has the property `is`, it is a Polymer element and we add it to the list of scopes, together with the path. Then we add each child of the element to the processing queue.

(c) If neither of the above is true, we have not found a new scope and we simply add all children of the element to the processing queue.

When the algorithm has finished, the list of scopes is sent to the main application. In the main application, the developer can choose the scope that they want to work with from a drop-down menu.

The selected scope has an influence on the following three components of XDTools:

- Shared JavaScript console: In the shared JavaScript console, all commands in the console are executed in the selected scope.
- Shared CSS editor: In the CSS editor, the CSS rule is added on the currently selected scope. Each scope has its own stylesheet containing the CSS rules for that scope.
- Function debugging: The functions that are debugged are on the selected scope.

In addition, the developer can directly jump into the HTML of each scope by clicking the HTML inspection button and choosing the scope from a drop-down menu.

6 Sample Applications

Over the course of our project, we developed two sample applications, XDCinema and XDYouTube. While developing those applications, we used XDTools to test and debug them. Using XDTools while developing two real cross-device applications helped us gain further ideas for improving XDTools and served as a first evaluation step for XDTools. In the following, we describe the two sample applications and the insights we gained from them.

6.1 XDCinema

As the name suggests, XDCinema is a cross-device cinema application. It allows users to check the cinema listings and view different information about movies and cinemas. It can be used with one to four different devices. On the ”main device”, the user can specify a city and date. They will then see a list of movies that are shown in this city on that date. The application is essentially split into four different views that can be seen in Figure 6.1 and that all show different information:

- The search view (a): In the search view, the movies shown in the selected city on the selected date are shown. For each movie, the cinemas in the selected city that show the movie and the times at which the movie is shown are displayed.
- The location view (b): In the location view, the location of the selected cinema is shown on a map. The location view is shown when the user clicks on a cinema in the search view.
- The movie view (c): In the movie view, information about the selected movie is displayed. It shows an image of the movie, the summary, the genres, the duration and the average rating of the movie. Furthermore, it displays a link to the trailer of the movie and the ticket prices of the movie in the cinemas in the selected city. If a cinema is

selected, the price that corresponds to that cinema is highlighted. The movie view is shown when the user clicks on a movie in the search view.

- The trailer view (d): In the trailer view, the trailer of the selected movie is displayed. The trailer view is shown when the user clicks on the link to the trailer in the movie view.

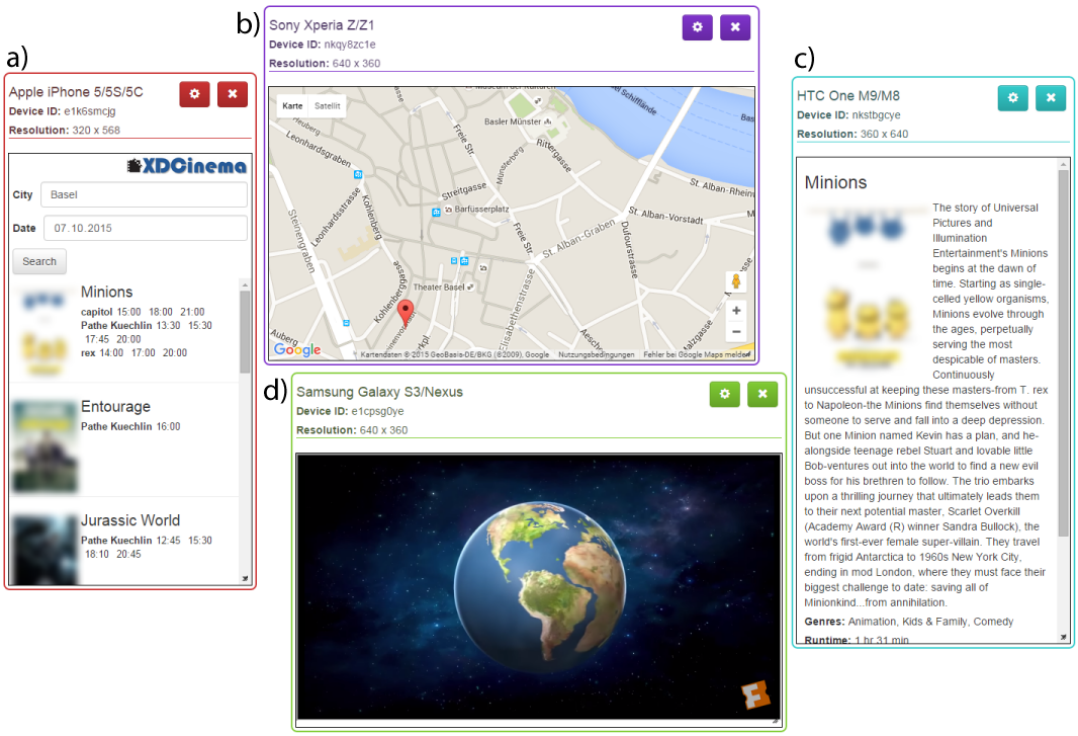


Figure 6.1: Different views of XDCinema

Depending on the number of devices that are connected, the four views are distributed differently. The trailer view is shown when the link to the trailer in the movie view is clicked and is only shown when at least four devices are connected. Otherwise, the link is just opened in a new tab on the device with the movie view. If only one device is connected, the remaining three views are all shown on that device and when the user navigates away from the search view they can get back to the search view using a back button. When two devices are connected, the movie view and the location view are both moved to the second device and the view that is shown depends on where the user last clicked in the search view. If the user clicks on a movie in the search view, the movie view is shown; if they click on a cinema, the location view is shown. If three devices are connected, the three views are distributed among all three devices. If more than four devices are connected, all additional devices do not show anything.

6.1.1 Implementation

XDCinema was implemented using the XD-MVC framework, but without Polymer. Thus, it is just a normal web application that accesses the JavaScript API of XD-MVC. Due to the

difficulty of finding suitable APIs in the cinema and movie domain, the data that is used in the application is statically encoded in a JavaScript file. However, the application could easily be extended to support access to an API instead. The DOM on each device contains all four application views and all except the appropriate application view are hidden depending on the role of the device.

Whenever a new device connects or a device disconnects, the roles are re-distributed and the views are updated. The available roles correspond to the four different application views, thus the role assignment is performed according to the rules of the view distribution described above. The role distribution happens randomly. The selected city, movie and cinema are shared between all connected devices through a shared variable that is synchronized using XD-MVC. When a shared variable is updated, each device performs the appropriate actions depending on its role. The selected city, movie, and cinema can only be changed from the search view. Therefore, all other devices only receive updates but do not send any updates themselves. If only one device is present, it has to hide the search view and location view if the selected movie is changed and it has to hide the search view and the movie view if the selected cinema changes. If the user wants to go back to the search view, it has to hide the location and movie view. If a device has both the movie and the location role, it has to hide the appropriate view when either the selected movie or the selected cinema changes. In cases where three or more devices are connected, the views that have to be hidden do not change and only the information in the view has to be updated if the respective shared variable changes.

6.2 XDYouTube

XDYouTube is a cross-device YouTube application that allows multiple people to watch videos together on one large screen. The application allows users to search for videos on their devices and add them to a queue. The videos in the queue are played one after another on a large screen. The users can also look at the currently playing video and the queued videos or pause and play the current video from their devices. The application is split into two different views, the controller view and the player view. In the player view (see Figure 6.2), the current video is played.

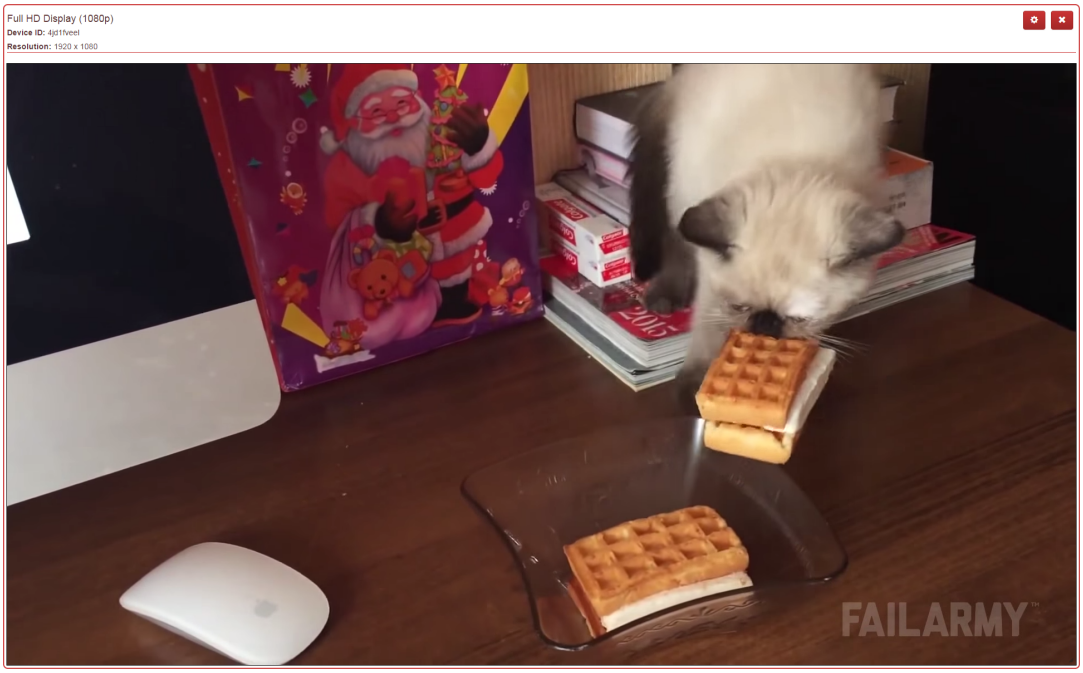


Figure 6.2: Player view of XDYouTube

The controller view consists of two different parts:

- In the first part, the user can search for videos. If the user searches for a video, a list of results is displayed. For each result, the thumbnail, title and description of the video is displayed. The user can click on a button to add the video to the queue. They can also navigate to the next page of search results.
- In the second part, the user sees the title, thumbnail and description of the video that is currently played. They can also pause or continue the video. The user also sees the thumbnails and titles of the videos that are still in the queue.

On large devices, both parts of the controller view are displayed simultaneously. On smaller devices, the first part is only shown in portrait mode (see Figure 6.3. The second part is only shown in landscape mode (see Figure 6.4).

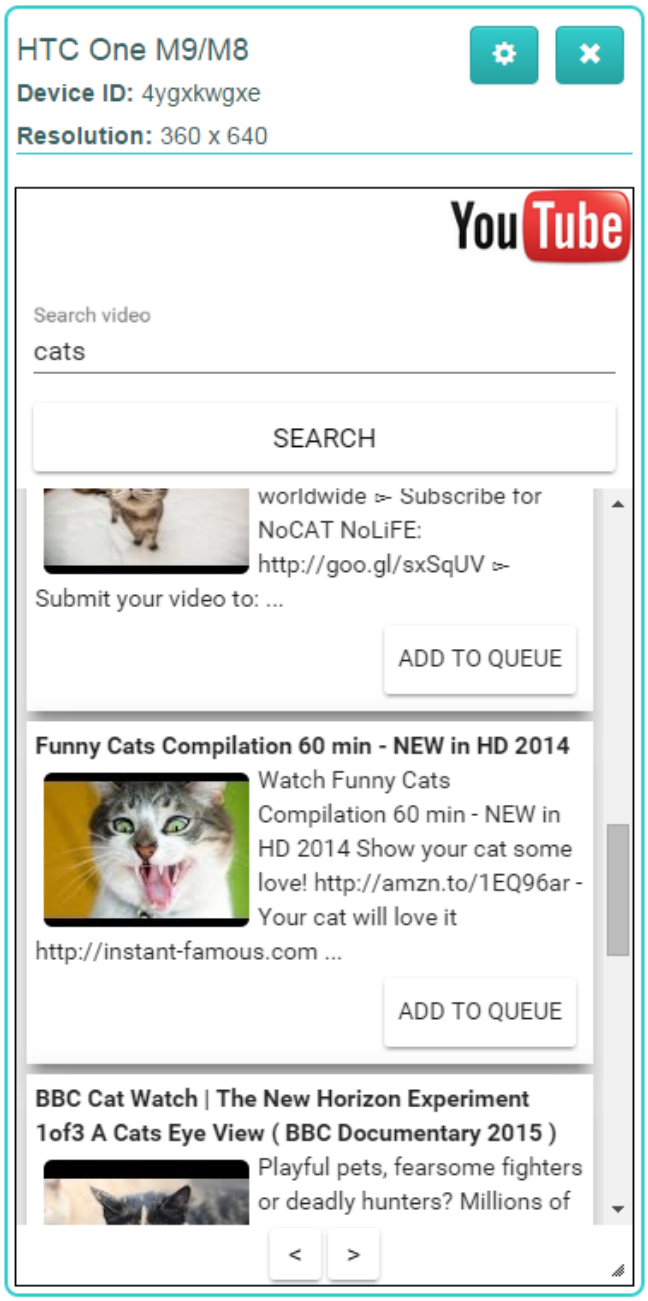


Figure 6.3: First part of the controller view in XDYouTube

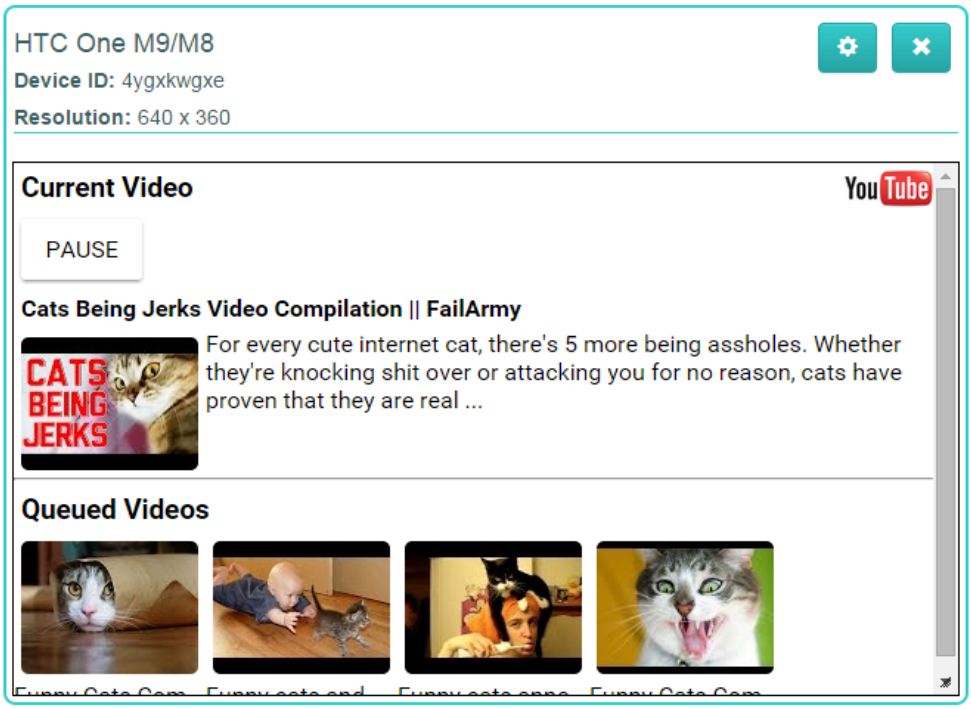


Figure 6.4: Second part of the controller view in XDYouTube

The device with the largest resolution automatically shows the player view. All other devices show the controller view.

6.2.1 Implementation

XDYouTube is implemented with XD-MVC in combination with Polymer. The devices in the application can have three different roles:

- ”player”: This role is assigned to the largest device.
- ”controller”: This role is assigned to all devices except the largest device.
- ”xlarge”: This role is assigned to devices with a width of more than 1000 pixels.

Depending on the role and the orientation of a device, different views are shown. As described above, the device with the role ”player” shows the player view. Devices with the roles ”xlarge” and ”controller” show both parts of the controller view simultaneously, devices that only have the role ”controller” show only one part of the controller view, depending on the orientation. Whenever a new device connects or a device disconnects, the roles and thus also the views are updated to reflect the new device configuration.

6.3 Insights

While developing the sample applications, we used XDTools. However, the implementation of XDTools was not complete yet at that point and some ideas emerged only from developing those applications. Before starting with the development of the sampling applications, the devices had to be connected manually. During the development, we realized that constantly re-connecting devices is one of the most time-consuming tasks while developing cross-device applications. From this, we came up with the idea of automatic connection management. Furthermore, we noticed that we often wanted to look at the JavaScript of the devices. But locating the correct file for a device was a difficult task, and the subdomains generated by our DNS server complicated things even more. Every device has its own subdomain and we first had to look for the subdomain of the device, then locate the subdomain in the large list of domains in the browser debugging tools, and finally navigating to the code that we want to look at. This made it evident that some mechanism for opening files or functions on a specific device was required. Thus, we integrated function debugging into XDTools.

Developing the sample applications also helped with locating and fixing some bugs. It also gave us some ideas for further improvements that are not yet included with XDTools. One example is the positioning of emulated devices: Right now, devices are always added in the top left corner of the device emulation space. Therefore, the developer always has to move the devices away from this location because otherwise the devices would overlap. Some algorithm that automatically looks for a free space where the emulated device can be added would help solve this problem and make creating devices even more efficient.

In general, XDTools already helped a lot while developing the sample applications. Adding multiple emulated devices that do not share any local resources was probably the most useful feature of XDTools. However, the shared JavaScript console also proved helpful while developing the sample applications. We did not use record and replay very often during the development of the applications. In hindsight, we can think of a few situations where it might have been useful, but in that situation, we did not think about using record and replay. Record and replay is a feature that has not often been used previously by developers, and we did not use it before either, thus it might take some time to get used to it and recognize situations where it is useful. Furthermore, our sample applications are rather simple and did not require extensive debugging. However, as far as we can tell from our development experience, XDTools already helps a lot when developing a cross-device application.

7 Evaluation

After implementing XDTools, we conducted a user study. The goal of the study was to evaluate the suitability and quality of XDTools. Thus, we designed multiple tasks that the participants had to complete. As a baseline, the participants also completed some tasks without XDTools, using only the browser debugging tools available in Google Chrome. In the following sections, we will describe the setup of the study, present our results and conclude with a discussion of the results.

7.1 Setup

The study was carried out in a room of the GlobIS group. The participants were sitting in front of a desktop PC with a 30-inch (2560x1600 pixels) and had access to an English (US) keyboard and a mouse. The desktop PC was running Microsoft Windows 7 with Google Chrome Version 45. Participants also had access to two other devices: An Asus Nexus 7 (2012 version) and an HTC M9 Android phone. Furthermore, they were given a tutorial sheet about JavaScript that contained some functions that are useful for DOM modification as well as arrays. Figure 7.1 shows a picture of the setup of the study.

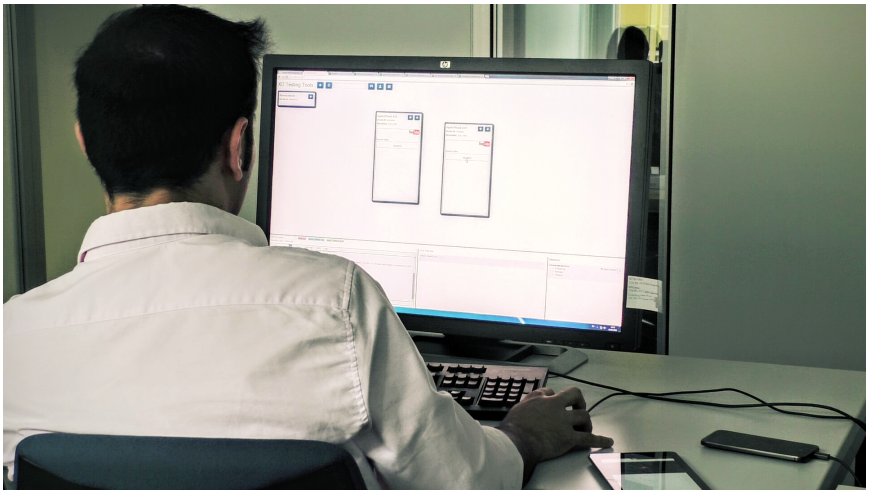


Figure 7.1: Study Setup

During the study, the instructor was sitting next to the participants and was available to answer any questions that occurred. If a participant did not make any progress on a task for some time, the instructor gave some hints to lead the participant in the right direction.

The participants had access to Chrome DevTools for all tasks. Furthermore, we set up multiple profiles on Chrome that the participants could use for emulating multiple devices. The participants could also remote debug the two real devices they had access to. The devices were already connected to the desktop PC by cable and the appropriate tab was opened in the browser. Thus, the participants only had to navigate to the tab and click on ”inspect” to remote debug a device.

During the study, we disabled some features of XDTools that were not required for completing the tasks so the participants would not be overwhelmed with the large amount of features that they had to learn how to use. We disabled the following features:

- Changing the URL: The URL was given by the task anyway and it would have been of no use to change it during the study.
- Saving and loading device configurations: This feature is mainly useful for long-term use of XDTools and was not needed for completing the tasks in the study.
- Record and replay: We think that record and replay takes some time to get used to it and it is also only of limited use for simple tasks like the ones that we used in the study. Furthermore, it might distract participants from the actual task because they might want to try it out even if it does not help them to complete the task.
- Inspecting HTML: The tasks in our study did not require modification or debugging of HTML, thus we deactivated this feature.
- Settings: The settings were disabled for the study because they were implicitly given by us.

All other features could be used by the participants. The following list provides an overview of the features the participants had access to:

- Emulating multiple devices
- Connecting real devices
- Connection features (auto-connect and drop-down list to connect to other devices)
- Shared JavaScript console
- Function debugging
- Shared CSS Editor

This list also represents the features that we wanted to evaluate during the study.

7.1.1 Participants

We recruited 12 participants (2 female) that all were university members in the department of computer science at ETH Zurich. Most participants were either PhD or Master students, but there were also some Bachelor students. The age of the participants ranged from 23 to 33 and the median age was 26. It was required that all participants have at least basic knowledge about front-end web technologies (i.e. HTML, CSS, JavaScript). However, the definition of ”basic” was up to the participants and we did not ask participants to prove their experience before the study. Consequently, we also had some participants that had rather low experience with web technologies. We did not require participants to have any experience with cross-device application development as otherwise it would have been difficult to find enough participants. Nevertheless, two thirds of the participants actually did have some cross-device application development experience.

Previous Experience

We asked all participants about their previous experience with web application development and JavaScript in particular, as well as about their previous experience with responsive web application development and cross-device web application development. Furthermore, we asked them about whether they have used Chrome DevTools before and how they used it. The participants rated their skills in web application development and JavaScript on a 5-point scale (see Figure 7.2) from basic to proficient and also gave the numbers of years in experience (see Figure 7.3) that they had in web application development and JavaScript. Even though only few participants had more than 5 years of experience, all participants rated their skills as rather good.

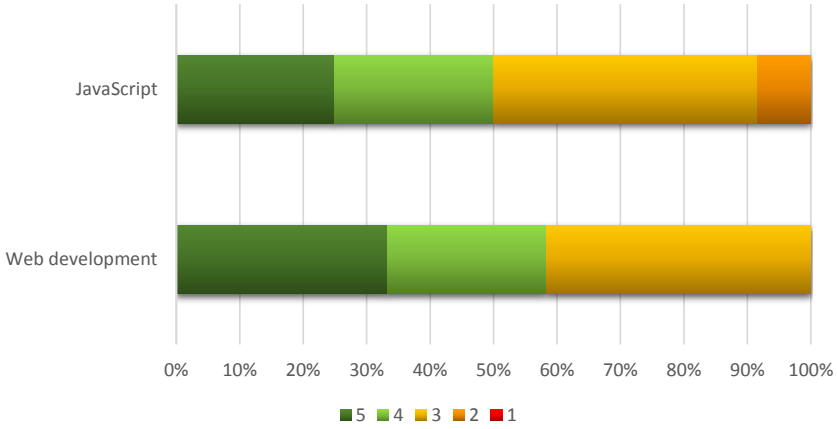


Figure 7.2: Previous experience

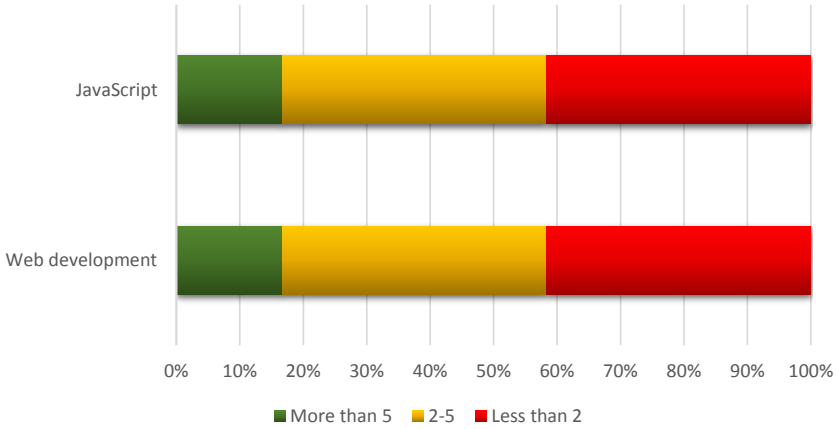


Figure 7.3: Years of experience

Nine out of the twelve participants stated that they already had some experience with developing responsive web applications. All except one used browser tools for emulating devices for testing their applications and all except two used real devices. Only one participant stated that they only used real devices for testing and no browser tools. One participant mentioned that they mainly re-sized browser windows to test their applications.

Eight participants already had some experience with cross-device application development. Again, most of them used browser tools for emulating devices and/or real devices for testing their applications. Four of them either used multiple browsers, multiple browser profiles or incognito mode for emulating multiple devices on one device. The fact that the other participants did not use any of those ways of preventing the sharing of local resources indicates that they probably used multiple devices at all times. This could either be because the participants do not know that they exist, because they are inconvenient to use or because the participants simply prefer real devices.

Most of the participants already had experience with Chrome DevTools, only two participants

indicated that they had never used them before and one of them had used the debugging tools of Firefox instead. We asked participants how often they used certain features of Chrome DevTools, in particular Device Mode, HTML and CSS inspection, JavaScript debugging and the console (see Figure 7.4). The participants did not use Device Mode that often, which is no surprise, given that it is a rather new feature. All participants stated that they often use HTML and CSS inspection, thus this seems to be the most popular feature. The console was also used rather often. Surprisingly, JavaScript debugging was not that popular, less than half of the participants stated that they often use it.

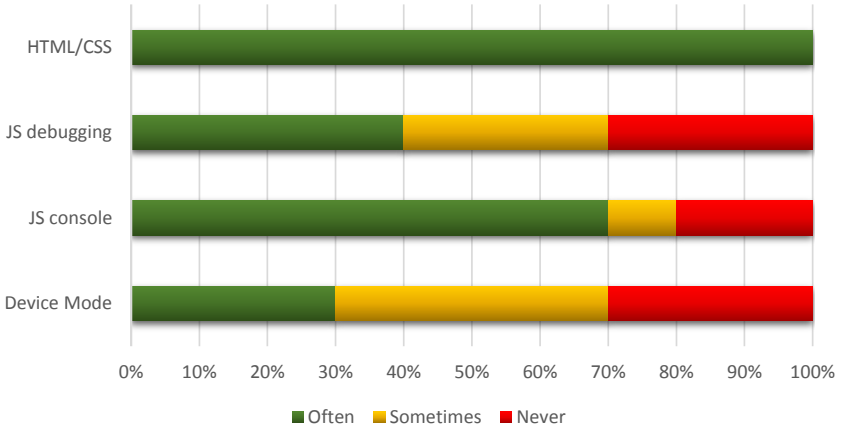


Figure 7.4: Previous experience with Chrome DevTools

7.1.2 Tasks

We used two different applications for the tasks, namely the two sample applications that we described earlier, XDCinema and XDYouTube. For each application, there were two tasks; one was about finding and fixing a bug in the source code, and the other one was about implementing a new feature for the application (the features were also included in our final version of the sample applications). The maximum time for completing the tasks where the participants had to fix a bug was 15 minutes and the maximum time for implementing a feature was 30 minutes. After this time, we aborted the task unless it was clear that the participants would finish within the next 2 to 3 minutes. Each participant had to complete all four of the tasks; the tasks of one application with XDTools, the tasks of the other application without it. The order of the applications as well as whether the first two tasks were with or without XDTools was counter-balanced. The participants completed the debugging task first and then the implementation task. Thus, the participants could learn how to use the application during the debugging task, which we considered easier than the implementation task, and were already familiar with the application for the implementation task. The complete task descriptions that were given to the participants can be seen in B. In the following, we give a summary of the tasks.

XDYouTube

XDYouTube allows users to use their personal devices to search for videos and add them to a queue. The videos from the queue are played one after the other on the largest of the devices. Users can also see the title and description of the currently playing video as well as the videos that are still in the queue by switching their device into landscape mode.

The first task (xdyt-bug) with XDYouTube was to fix a bug concerning the video queue. As soon as one video finishes playing, the next video is dequeued from the queue and starts playing. However, when no video is in the queue, a JavaScript error occurs and causes the next video that is added to the queue not to play. The participants were given a description of the task and then had to reproduce and fix it. The participants were given a JavaScript file with two functions, one that adds a video to the queue and one that loads the next video at the end of a video.

For the second task (xdyt-impl), we asked participants to implement a remote control that can play and pause the current video from the controller devices. The participants had to implement two functions: One is called when the remote control button is clicked (the button as well as the event handler were already implemented), the other one is called when a shared variable that states whether the video is paused or playing is changed. Thus, the participants had to change the shared variable whenever the button was clicked and react accordingly on all devices if the shared variable changes, i.e. they had to pause or play the video on the device that plays the video and they had to change the text of the remote control button from ”Pause” to ”Play” and vice versa on all other devices. Furthermore, they had to change the CSS of the button such that it looked similar to a picture of a button that was given to them. The participants were given a JavaScript file with the two empty functions as well as a CSS file with the empty CSS selector of the button. Furthermore, they had access to a few helper functions that simplified the task.

XDCinema

XDCinema allows users to search for a city and date on one device. The device then shows a list of movies that play in this city on that date as well as the cinemas where the movie is played and the time that the movie starts. If the user clicks on a cinema, a summary of the movie as well as other information about the movie is shown on another device. If the user clicks on a cinema, the location of the cinema is shown on another device.

The first task (xdc-bug) was to fix a bug where the location of most cinemas was displayed wrongly, even though the data from the database was correct. The bug was that in one function, the variable ”j” was used instead of ”i”, which caused a wrong location to be returned. The participants were given a JavaScript file with a few functions related to getting and updating the location on all devices.

In the second task (xdc-impl), the participants first had to complete the implementation of a function that shows the prices of each cinema in the selected city where the movie plays below the description of the cinema. A skeleton for this function was already given where a loop over all cinemas that show the movie was already implemented, the participants only had to fill in the body of the loop. The second part of the task was to highlight the correct

price (the participants were given a CSS class called ”highlighted”) when the user clicks on a cinema in the search view and to improve the CSS for highlighting. In the original version, highlighting used a light grey background color and a white text color which was not very readable. The participants were given a JavaScript file with the two functions as well as CSS file with the initial CSS of the ”highlighted” class. As in the XDYouTube implementation task, the participants also had access to some helper functions.

7.1.3 Evaluation Methods

In total, we used four different methods for evaluating the results of the user study. During the study, the participants had to fill out multiple questionnaires. Most our results are based on those questionnaires, the other methods (video recording, personal feedback, time measuring) for evaluation are only used for clearing any inconsistencies, finding explanations for things that cannot be explained with the questionnaires alone and gathering some additional information that we missed during the study.

Questionnaires

At the beginning of the study, each participant had to fill out a questionnaire about their background information. The results of this questionnaire were already presented earlier. After each task, the participant had to fill out another questionnaire with the following questions:

- It was easy to complete the task with the tools I had access to.
- I felt efficient completing the task with the tools I had access to.
- It was challenging to complete the task with the tools I had access to.
- The tools I had access to were well suited for completing the task.

The questions could be answered on a 5-level Likert scale from ”Strongly Disagree” to ”Strongly Agree”. In the tasks where the participants had access to XDTools, we also asked them to rate the usefulness of the individual features of XDTools on a 5-level Likert scale. The following question was asked about the features: ”How useful did you find the following features for completing the task?”. The participants could answer this question for each feature on a 5-level scale from ”Not useful” to ”Very useful”. They could also choose the option ”Not used” for features that they did not use while completing the task. For each task, there also was a comment field where the user could write down any additional comments that they had about the task or the tools they had access to.

After completing all tasks, the participants had to fill out a final questionnaire where they could answer some questions that compare XDTools to the browser debugging tools that they had access to. The participants could state whether it was easier and felt more efficient to debug and implement with or without XDTools. They could also state whether they preferred implementing and debugging with or without XDTools.

In addition, they also answered some general questions regarding the usability of XDTools, whether they think that XDTools is useful and whether they would use it for implementing and debugging cross-device applications.

Finally, the participants could state which features of XDTools they would use for debugging and implementing cross-device applications and they could also write some comments about XDTools if they wanted to.

The complete set of questions can be seen in the questionnaires in A.

Video Recording

During the study, we used a video camera to record the participants while completing the tasks. This was mainly done to make sure that no important information was lost and for extracting some task completion strategies from the videos.

Personal Feedback

At the end of the study, participants were encouraged to share any comments that they still wanted to mention and to give their general opinion about XDTools. Any comments that the participants had mentioned during the study were also noted.

Time Measuring

For each participant, the time required for completing each task was measured. This was mainly done to detect any major discrepancies between completion times with and without XDTools. However, exact times are not considered relevant for evaluation because they highly depend on the participant and on the hints given by the instructor during the study.

7.2 Results

In the following sections, we will first present the results from the individual tasks. For each task, we will compare how participants answered the questions in the per-task questionnaires with and without access to XDTools.

7.2.1 XDCinema: Fixing a Bug

The results for the task where the participants had to fix a bug in XDCinema can be seen in Figure 7.5. The figure shows the median values for the questions asked after the task with and without XDTools. For the question that asks about how challenging it was to complete the task with the tools the participant had access to, a lower value is better; for all other questions, a higher value is considered better. There is a rather big difference in the median value for the suitability of the tools, while the differences are smaller for the other questions or not there at all. Thus, it seems that even though the participants felt that XDTools is more suited for the task than the standard browser debugging tools, there is no significant difference regarding efficiency or easiness. However, this is not surprising, as the suitability of the tools is generally independent of the task, whereas the efficiency and easiness depend on the task.

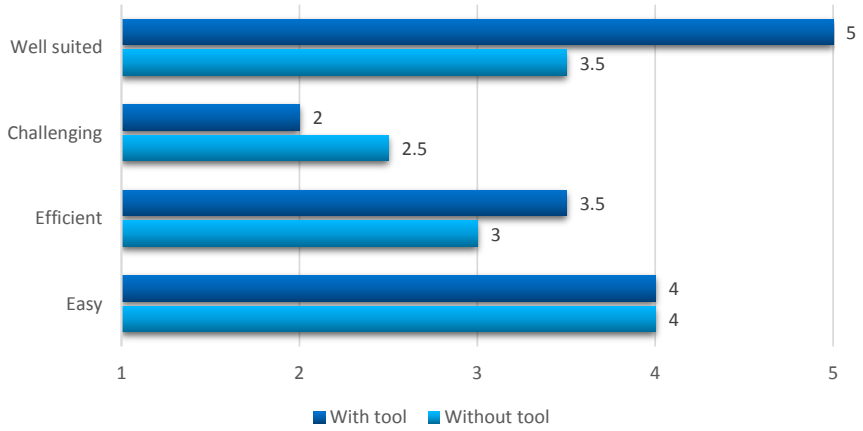


Figure 7.5: XDCinema debugging task - Comparison

Figure 7.6 shows how many participants used the individual features of XDTools and how useful they found them. Obviously, the figure only includes the six participants that had access to XDTools. None of the participants used real devices and all of them used device emulation instead. However, two participants rated device emulation with a 3, which indicates that they found it somewhat useful for the task, but not extremely useful. The other participants all found device emulation very useful. The connection features and function debugging were used by all except one participant and were well appreciated by the participants. The shared JavaScript console was rather unpopular for this task, probably due to the simplicity of the task and also due to the fact that the bug produced no JavaScript errors that would be displayed in the console.

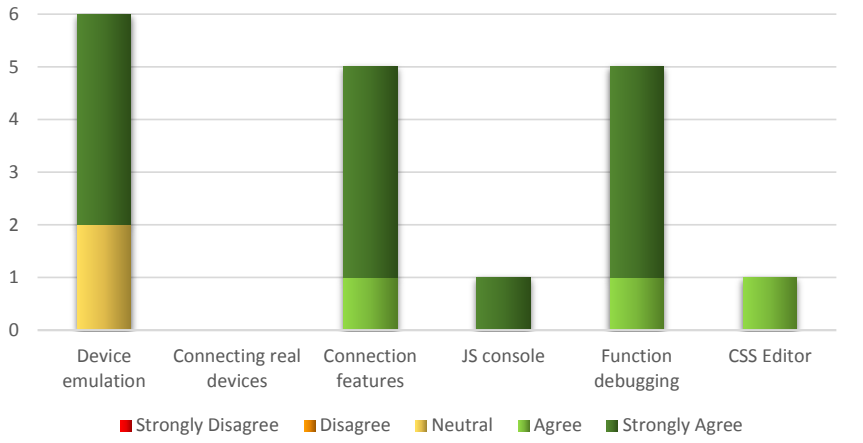


Figure 7.6: XDCinema debugging task - Features used and their Usefulness

7.2.2 XDCinema: Implementing a Feature

In Figure 7.7, the results for the implementation task in XDCinema can be seen. In this task, the difference in suitability is less pronounced than in the XDCinema debugging task,

but the difference for the question about efficiency and easiness is larger. In general, the task was considered as rather easy independent of the tools the participant had access to, although participants that had access to XDTools perceived it as a bit easier. The median value of five with XDTools suggests that participants had no problems completing the task at all when they had access to XDTools. Similar results can be observed concerning efficiency. The difference in median values suggests that the participants felt a bit more efficient with XDTools. Surprisingly, if we compare the average and median completion times for this task, the participants that had access to XDTools were considerably slower (comparing median values, the participants that had access to XDTools were about 9 minutes slower). In fact, this is the only task where the difference in completion times with and without XDTools is noticeable; the completion times for all other tasks are almost equivalent. However, those two facts do not necessarily contradict each other, as there were participants with very different experience levels and due to the low number of participants, such factors can easily have a big influence on completion times. In general, the completion times should not be considered as especially relevant, after all the instructor also gave some hints during the study and this can also distort completion times significantly. However, it would be interesting to see how completion times differ if there is a larger number of participants and if the participants had to complete tasks completely on their own.

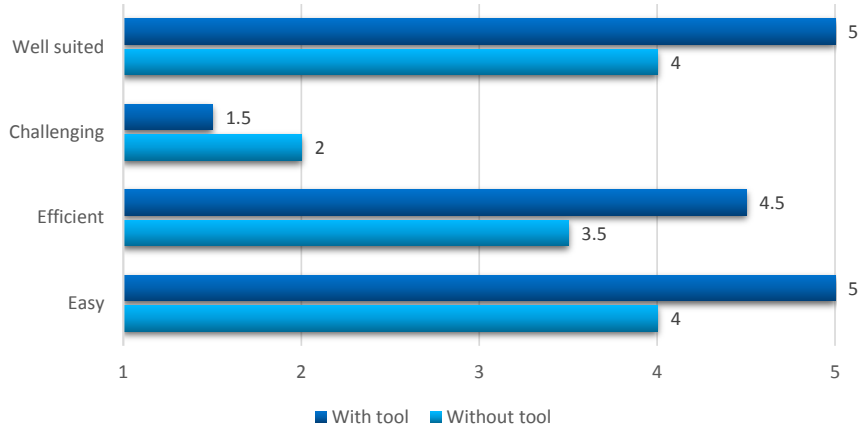


Figure 7.7: XDCinema implementation task - Comparison

Figure 7.8 again shows the use and the ratings of the individual features. Again, no participant used the real devices. All participants used device emulation and the connection features and in contrast to the debugging task in XDCinema, device emulation was rated as very useful by all participants. Function debugging was used a bit less than in the debugging task, but the shared JavaScript console was used much more. It makes sense that function debugging is used more when fixing a bug; if one implements a feature and it works immediately when testing it, there is no need to debug a function, but if one has to fix a bug, there obviously must be a bug in a function and thus it makes much more sense to debug functions. The shared JavaScript console was rarely used to send commands and most participants did not use logging for solving the task, but many participants had some syntax errors when first testing the feature and noticed the error messages in the console. Finally, the CSS editor was also used by some participants in this task. However, some participants completed the CSS

part of the task using only the CSS file. This may be because they did not think of the CSS editor at this specific moment, or because they know CSS so well that they can just write everything down immediately, or also because they do not consider the CSS editor useful for this task.

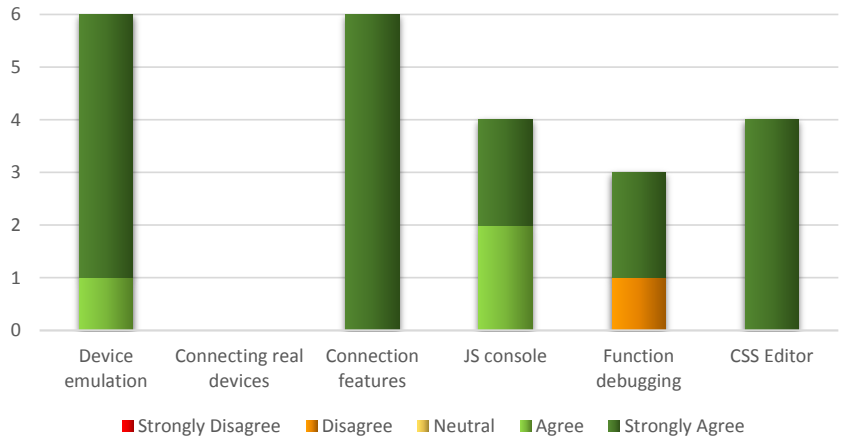


Figure 7.8: XDCinema implementation task - Features used and their usefulness

7.2.3 XDYouTube: Fixing a Bug

Figure 7.9 shows the results for the debugging task in XDYouTube. The difference in suitability between XDTools and the usual browser tools was most significant in this task with a difference of 2. The difference in efficiency is also rather large. However, the task was rated as almost equally easy and challenging with and without XDTools despite the large differences in the other questions. Thus, XDTools seems to mainly influence the efficiency for this task.

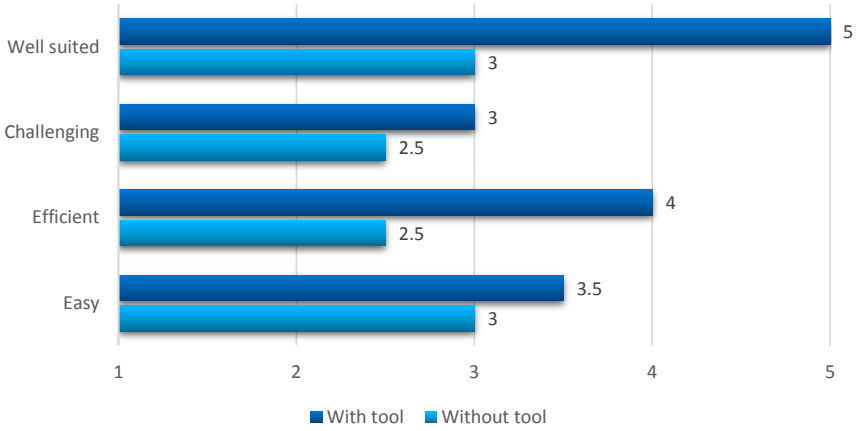


Figure 7.9: XDYouTube debugging task - Comparison

In Figure 7.10, the use and ratings of the individual features can be seen. All of the parti-

cipants used device emulation and connection features and rated them as useful. Function debugging was also used by almost all participants, probably because it was difficult to reproduce the bug and the participants wanted to see what was going on in the functions. About half the participants used the shared JavaScript console, mainly to see the error produced in the function that caused the bug. One participant connected the Nexus 7 to XDTools and liked the feature, but no statement about the general usefulness of the feature can be made from just one participant.

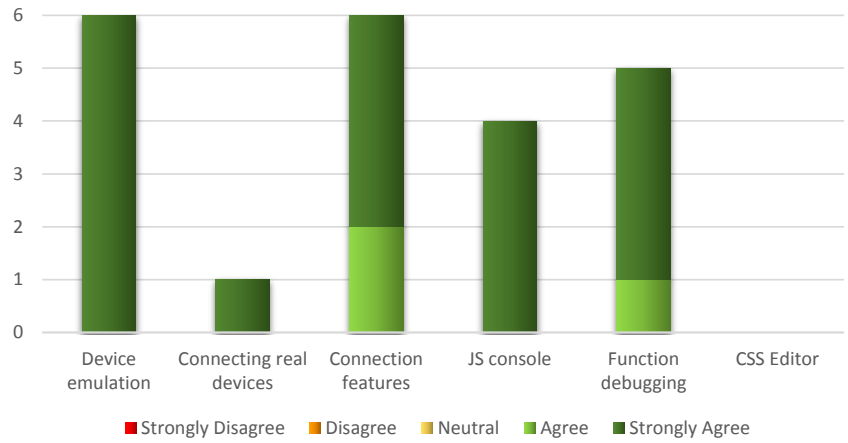


Figure 7.10: XDYouTube debugging task - Features used

7.2.4 XDYouTube: Implementing a Feature

Figure 7.11 shows the results for the implementation task in XDYouTube. In this task, all questions were clearly rated in favor of XDTools. While the participants answered the questions in a rather neutral way when they did not have access to XDTools, they clearly stated that the task was easy to complete and felt efficient to complete with XDTools. This task differs from the others a bit, because in all other tasks the difference in median values was rather insignificant for some questions, whereas the difference is at least 1 in this task for every question. Thus, it seems that XDTools makes the most difference for this task regarding all aspects.

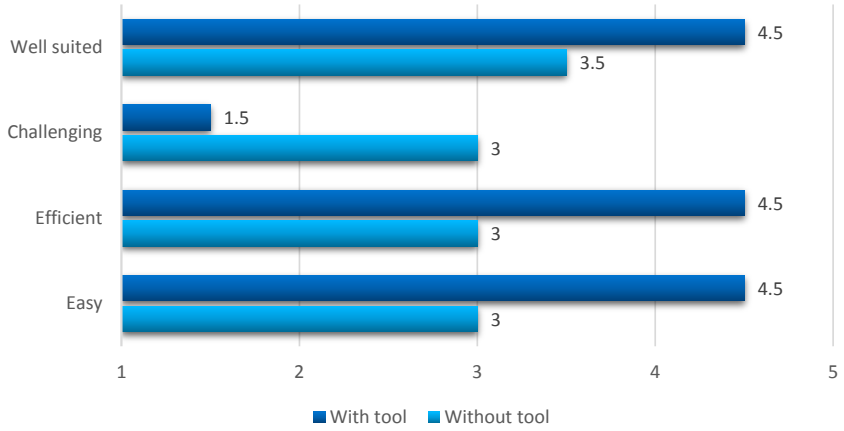


Figure 7.11: XDYouTube implementing task - Comparison

In Figure 7.12, the use and ratings of the individual features can be seen. Once again, device emulation and the connection features were used by every participant. The shared JavaScript console and CSS editor were about equally popular and rated as very useful except for one participant found the console only partially useful. Function debugging was used by half the participants and rated as very useful.

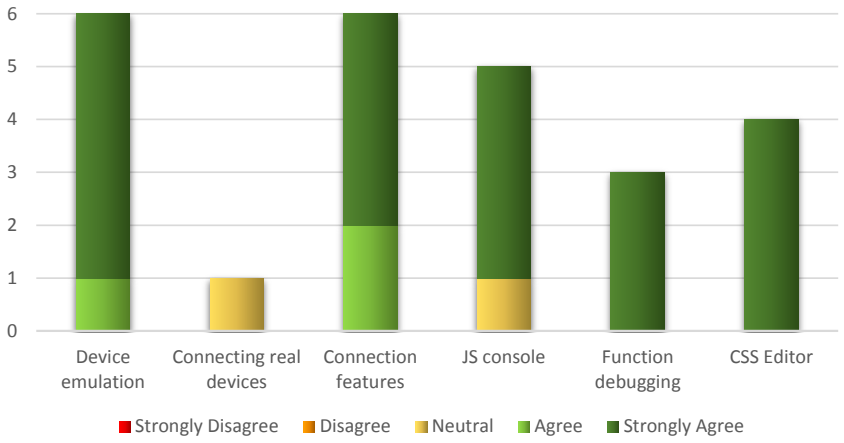


Figure 7.12: XDYouTube implementing task - Features used

7.3 Discussion

Figure 7.13 shows that about three quarters of all participants considered implementing a feature easier with XDTools. Only one participant found it easier to implement a feature without XDTools. However, in principle, this option is redundant as the participants still have access to the browser debugging tools even when they have access to XDTools. This essentiall means that about one quarter of the participants did not see any gain in easiness from using XDTools. The same applies to the question about whether implementing a feature

feels more efficient with XDTools (see Figure 7.14); this figure shows exactly the same results as the figure about easiness. However, all except one participant preferred implementing a feature with XDTools (see Figure 7.15). Thus, it seems that participants like to have access to XDTools even if they cannot directly relate it to a decrease in difficulty or to an increase in efficiency.

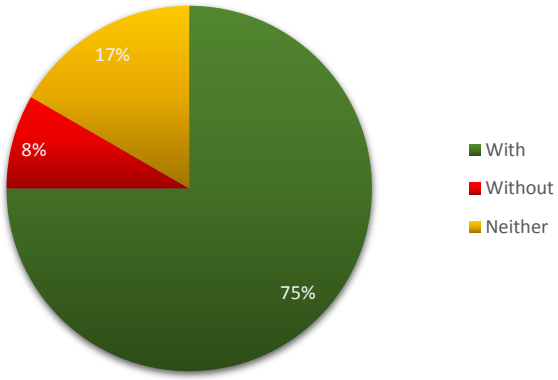


Figure 7.13: Easiness of implementing a feature

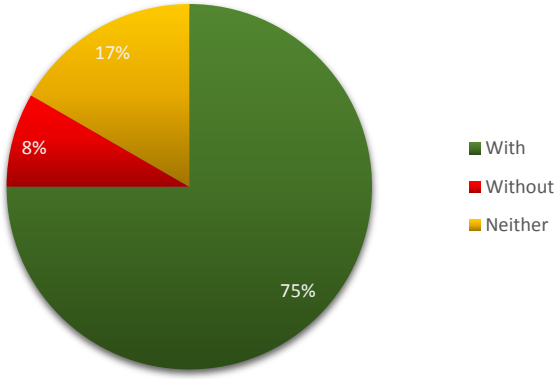


Figure 7.14: Efficiency of implementing a feature

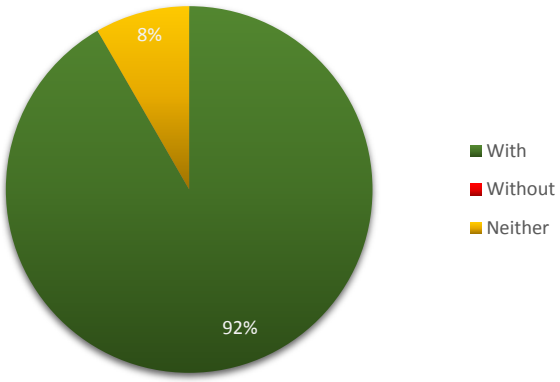


Figure 7.15: Preference for implementing a feature

Figure 7.16 shows that most participants found it easier to debug a cross-device application with XDTools. The results get even more obvious if we look at Figure 7.17 and Figure 7.18. Those two figures show that all participants felt more efficient when debugging with XDTools and also preferred debugging with XDTools.

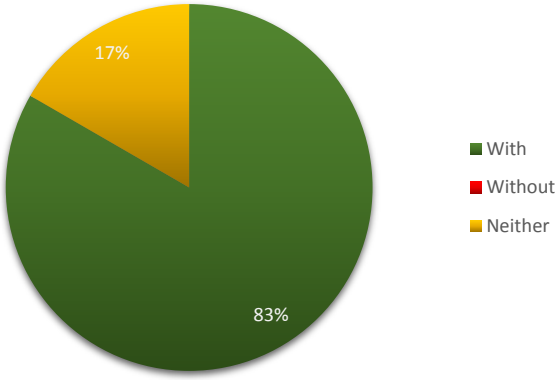


Figure 7.16: Easiness of debugging

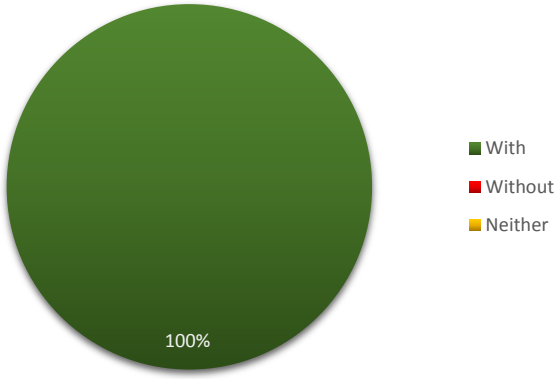


Figure 7.17: Efficiency of debugging

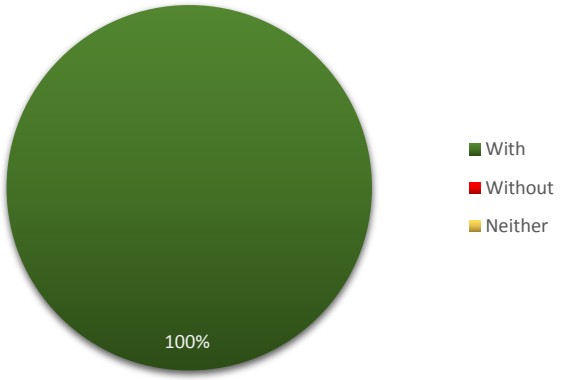


Figure 7.18: Preference for debugging

We also asked the participants if they would use XDTools for debugging and implementing cross-device applications and if they think that XDTools would be useful for cross-device application testing. The results can be seen in Figure 7.19. Almost all participants think that XDTools would be very useful for implementing as well as debugging cross-device applications and the remaining few also think that they would be useful. All participants would use XDTools for implementing cross-device applications and all except one participant would use them for debugging cross-device applications. A few more participants stated that they strongly agree with the statement that they would use XDTools for debugging than for implementing. This is consistent with our previous results, where all participants preferred debugging with XDTools but about one quarter did not prefer implementing with XDTools.

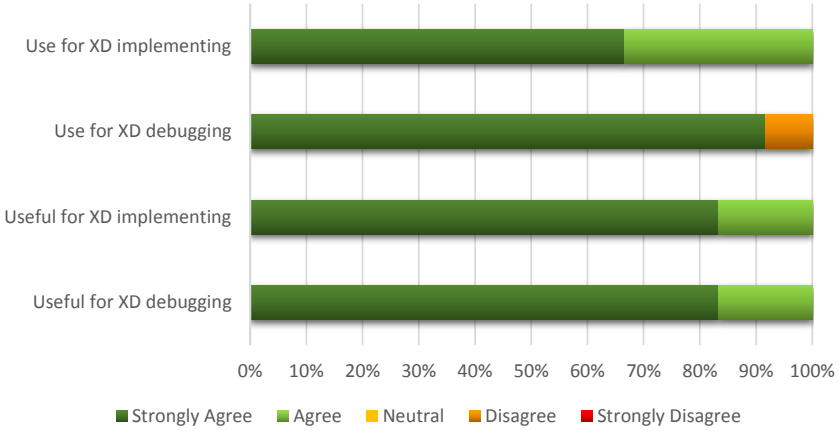


Figure 7.19: Usefulness of XDTools

Finally, the results for the general questions about XDTools can be seen in Figure 7.20. The figure shows that XDTools was perceived as easy to learn despite the many different features and the participants also felt rather confident using XDTools. However, some participants rated the question about confidence in a neutral way and most other participants felt only somewhat confident, but not very confident. This indicates that even though XDTools is considered easy to learn, it may still take participants some time to get used to it. XDTools are not

considered as unnecessarily complex by most of the participants. The results of those three questions indicate that the interface of XDTools is generally well-structured and it would be easy for developers to get used to XDTools.

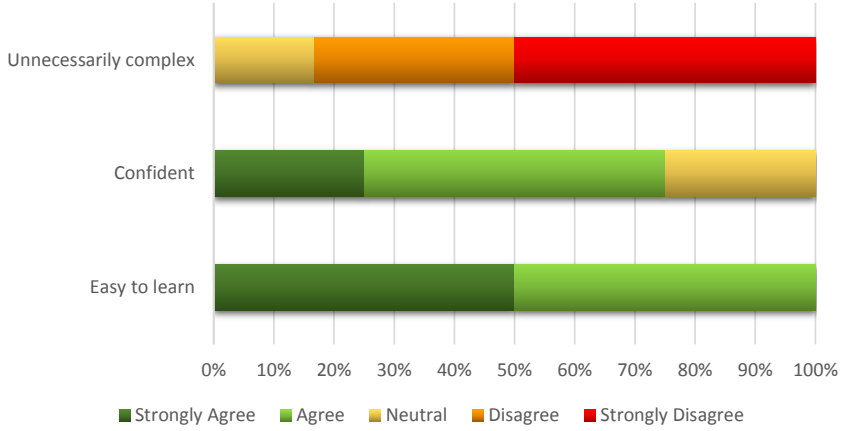


Figure 7.20: General evaluation of XDTools

Figure 7.21 shows which features the participants would use for implementing and debugging cross-device applications. In general, participants would rather use emulated devices than real devices for both implementing and debugging. This is understandable as most things can be done just as well with emulated devices as with real devices. Some participants mentioned that they would not use real devices during implementing, but that they would test their application on real devices after finishing implementing to make sure the application works fine on them. The connection features are almost unavoidable to use, thus it is surprising that some participants state that they would not use them. However, for some parts of debugging and implementing, one device might be sufficient for testing and no connection features would be required. One participant mentioned that the connection features seem very natural and that there is no point in asking about their usefulness because it is obvious that they are useful. This indicates that the feature was indeed greatly appreciated by some participants. The shared JavaScript console is equally popular for debugging and implementing and would be used by almost all participants, thus it seems to be a very popular feature as well. Function debugging is more popular for debugging than for implementing. This corresponds to the actual results of the study and has been elaborated before. Apart from connecting real devices, the shared CSS editor is the least popular. This may be due to the fact that browsers already have quite mature CSS editors and it might be possible to test CSS on one device at a time in many cases. Also, the CSS parts of our tasks were rather simple, thus the real value of such a feature might not be obvious to the participants. While things like changing the background color of a button can easily be done on just one device, more challenging CSS problems like positioning elements require more effort and look much different on different devices. Furthermore, cross-device applications do not always show the same things on all devices and applying CSS to all devices is of no use if the corresponding elements are only shown on one device anyways.

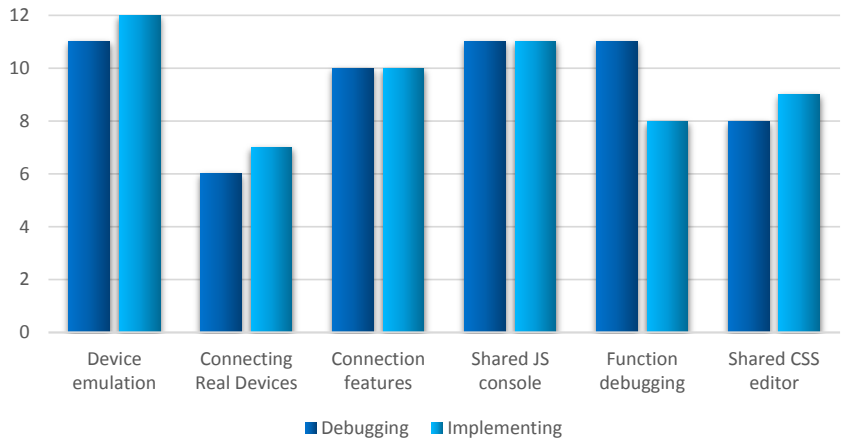


Figure 7.21: Features that the participants would use

During the study, some participants mentioned that they found device emulation very useful because they can have all devices in one place instead of having to manage different browser windows and profiles. Reloading all devices at once was also considered as very useful. With multiple browser windows, each window has to be reloaded individually which makes this a much more tedious task. It was also appreciated that the devices are still connected after reloading, even though for some applications this is also the case with multiple browser windows. The participants really liked function debugging, the only criticism was that it does not show on which device a function is called. In general, the output aggregation of the shared JavaScript console was considered more useful than sending commands t multiple devices. One participant even mentioned that they would not use it to send commands, but the aggregation is very useful.

In general, the feedback during the study showed that participants really liked XDTools. One participant stated that they think that the tool is very useful and they wish they had access to it when they were working on a cross-device application. Another participant mentioned that ”The existing features are already very impressive and work well. They really help the developer working on cross-device applications.”

Record and replay was disabled for the user study, but one participant that had attended a presentation about XDTools before, mentioned that they would find it immensely useful. They consider it as a powerful feature that could be very helpful for replicating bugs in an application and for regression testing. Often, it is not clear how to reach a bug and being able to record the set of interactions that lead to the bug and then replay them can simplify this process.

Aborted Tasks

Due to the time limits that we set, we also had to abort some tasks. There were five cases where we had to abort a task, four while participants had access to XDTools, and one where the participant did not have access to XDTools. There was one participant that finished all tasks except the XDCinema debugging task. The participant had access to XDTools for this

task, but we do not consider this relevant for this specific case, as the time required for completing this task can vary greatly depending on how fast the participant notices the switched variable. Also, the participant had completed all other tasks successfully. Furthermore, there was one participant for whom we had to abort both implementation tasks (one with XDTools and one without) and one participant where we had to abort both XDCinema tasks (both with XDTools). Both those participants had very low experience in web application development and had problems completing basic things like placing brackets correctly around an if-statement and creating a new line in HTML. This indicates that those participants had very low programming experience in general and it is not surprising that they failed to complete some tasks. The participants also had problems completing the other tasks, but eventually managed to finish the tasks with a lot of help from the instructor. Thus, we do not think that the fact that most of the aborted tasks were carried out with XDTools says anything relevant about XDTools.

Devices Used

Overall, the number of devices used did not vary greatly depending on the participant and on the condition the participant was in. Although our tasks were designed to require at least two devices in general, it was possible to complete the XDCinema debugging task with only one device, but the task required less device interaction when multiple devices were involved. Figure 7.22 shows how many emulated devices the participants used on average with and without XDTools. The number of emulated devices used is slightly higher with XDTools for all tasks. One reason for this difference could be that it is easier to add a new device with XDTools. However, the difference is rather small and further investigation is needed to see if this is really the case. For the XDCinema debugging task, only two out of six participants that did not have access to XDTools used more than one device. With XDTools, four participants used more than one device. This also indicates that participants prefer to use more devices when they have access to XDTools. For the XDCinema implementation task, all participant used exactly three devices (when combining real and emulated devices) which is not surprising because the minimum number of devices required was three and additional devices did not provide any value.

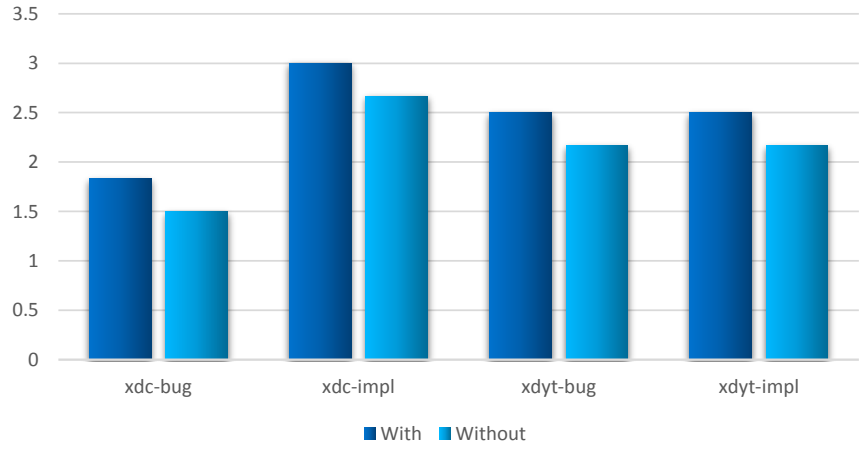


Figure 7.22: Average number of emulated devices used

The real devices were only rarely used overall. In the XDYouTube implementation and debugging tasks, one participant per condition used one real device. In the XDCinema implementation task, one participant that did not have access to XDTools used two real devices. Normally, most of the tasks required for implementing cross-device applications can be performed just as well on emulated devices. Testing on real devices is typically limited to points where major implementation steps have been completed or when a developer wants to see how interactions feel or work on actual touch devices. Therefore, it is no surprise that real devices were not used frequently during our study. The number of devices used during the tasks shows that although participants sometimes stick to the minimum number of devices required, they also like to be able to add more devices for convenience of further testing. In the XDYouTube implementation task, participants often added three devices so they could put one in landscape mode and one in portrait mode. This allowed them to add videos to the queue and test their implementation of the remote control button without switching orientation in-between.

In terms of screen sizes used, participants mostly used low-resolution mobile phones and sometimes tablets when carrying out tasks with XDTools. The devices were in portrait mode at almost all times, except for when they were required to be in landscape mode by the application. This is probably due to the dimensions of the area where devices can be placed which makes it more convenient to have devices in portrait mode. Furthermore, most predefined devices are in portrait mode by default and participants did not switch their orientation unless required to. Without access to XDTools, participants also often used smaller devices, but sometimes they had one large full-screen browser window and other smaller browser windows that they switched to when required. Participants that used three devices often let one device fill the left half of the screen and let the other two devices fill one quarter of the screen each, which put those devices in landscape mode.

Debugging and Implementation Strategies

In both debugging tasks and both conditions, about half the participants started by looking at the code and about half started by trying to reproduce the bug. Most of the participants

switched between the code and the application multiple times while carrying out the tasks, although some looked at the code directly in the DevTools. Almost all participants used the DevTools at some point for completing the task, and most of the participants that had access to XDTools used function debugging. The debugging task in XDCinema and XDYouTube had some fundamental differences: The main difficulty for the XDYouTube debugging task was reproducing the bug, once the bug was reproduced, the cause of the bug was pretty obvious given the JavaScript error the bug produced. In contrast, reproducing the bug in the XDCinema debugging task was trivial, but finding out what causes the bug was more difficult. In XDYouTube, participants usually spent quite a lot of time trying to reproduce the bug before they switched to the DevTools or function debugging. Only after participants failed at reproducing the bug, they used tools for debugging in an attempt to find the bug this way. In XDCinema, participants switched to the DevTools or function debugging much faster because they reproduced the bug almost immediately. Consequently, in XDYouTube, much more time is spent switching between the code and application because the participants try out different ways of finding the bug. In XDCinema, the debugging process is much more streamlined because after reproducing the bug, participants mostly stick to either looking at the code or using tools for debugging.

In the implementation task, some participants started by looking for the HTML element that contained the prices or the remote control button first, while others started implementing right away. All participants implemented everything required for the XDYouTube implementation task at once and only then started testing it. Only the CSS was usually added after making sure that the button works. On one hand, it makes sense to implement the complete remote control functionality at once because if only part of it was implemented, it would be difficult to see if the implemented function actually works. On the other hand, we would have expected that at least some participants implemented part of the functionality and used logging mechanisms to see if it works. In the XDCinema implementation task, the task was clearly separated into to parts and the first part did not require the second part to work. Thus, all except one participant implemented the first part, tested it, and then moved on to the second part. However, almost all participants had to modify their first part a little bit while implementing the second part because of the different requirements. The CSS part of the task was again implemented after completing everything else by most participants.

In summary, no real difference in implementation and debugging strategies can be seen between the two conditions of our study. The only difference is that some parts of the debugging process are made more efficient or easier because the developer has access to additional tools. Thus, developers should be able to use the same strategies for debugging and implementing features as they usually would and augment them with additional features provided by XDTools where it is useful.

Feature Requests

During the study, some participants suggested some improvements to existing features that would help them even more. Multiple participants suggested that function debugging should show on which device a function is called. We were able to successfully implement this feature after completing the user study. Also, one participant said that auto-connect should be enabled by default because there rarely is a situation where a developer would not want to

connect the devices. Over the course of the study, it became obvious that this feature would be useful as many participants forgot to enable auto-connect before adding more devices and thus either manually connected them or removed them, enabled auto-connect, and re-added the devices. Furthermore, it was suggested that the border of the device should be in the device's unique color so devices can more easily be recognized. Those two features were also included into XDTools after the study. Also, it became clear during the study that it was not at all obvious when function debugging was working and when the participants had to re-open the DevTools. Due to this, we now show a warning when the developer should re-open the DevTools to make sure that function debugging works. Also, we now display a warning message when the user wants to reload the page because some participants accidentally reloaded the page instead of the devices and then lost all their devices. Some additional feature suggestions include:

- CSS value suggestions, e.g. suggest "1px solid black" when the developer sets the border property
- More integration with the DevTools
- Opening the DevTools automatically when the developer wants to debug a function
- Auto-complete or similar for functions in function debugging
- Combining a source code editor with XDTools

Due to time constraints and in some cases also feasibility constraints we were not able to implement those feature requests (yet). Also, while some of those requests like CSS value suggestions would clearly be useful, others like combining a source code editor with XDTools require more research to determine if they are actually desired by developers and would provide some additional benefits.

8 Conclusion

In this master's thesis, we have analyzed how existing tools support the testing and debugging of cross-device web applications. We found that there are many tools available for testing web applications in general and also responsive web applications, but all those tools have significant disadvantages when it comes to cross-device application testing. There are significant differences between cross-device applications and traditional web applications: One of the biggest differences is that multiple different devices that show different content are typically involved in a cross-device scenario. Those differences make it difficult to debug cross-device applications using tools aimed at testing traditional applications that usually only involve one device at a time. We also analyzed existing cross-device application development frameworks and found that they provide little to no support for testing and especially debugging the applications. However, many of those frameworks could benefit from a tool that helps debug the applications developed with them.

With the limitations discovered during our analysis in mind, we specified a set of requirement that a cross-device application testing and debugging tool has to fulfill. Using those requirements, we implemented XDTools, a system that facilitates the testing and debugging of cross-device applications. XDTools is implemented as a web application and is thus largely independent of the system it is running on. XDTools includes concepts that have already been seen in responsive web application testing tools as well as concepts that are well-known from browser debugging tools and only required extension to suit the needs of cross-device applications. XDTools also features a record and replay tool. Record and replay has already proven to be useful for reproducing bugs in traditional web applications and it provides additional value for cross-device applications where multiple users can be simulated with such a tool. Finally, our system also includes some features targeted specifically at cross-device applications, such as an automatic connection tool.

Using our system, we also implemented two sample applications that provided useful insights for further improvements to our system and also served as basis for our user study.

Finally, we conducted a user study to evaluate our system. The participants of our study had to complete multiple tasks where they had to either fix a bug or implement a new feature, either using our system or only using traditional browser debugging tools. The results of the user study show that our system indeed helps with debugging cross-device applications and we received some enthusiastic feedback from participants that already had some previous experience with cross-device applications and struggled with testing them.

Overall, we think that our system already provides a number of useful features for cross-device application testing and debugging cross-device applications and adds a lot of value to the existing web-based cross-device application development frameworks. Testing cross-device applications is an important step on the way to a more widespread adaptation of cross-device applications and we hope that it will help advance the field further.

8.1 Future Work

We will now present a few ideas for future work that could improve XDTools.

8.1.1 Extended Record and Replay

So far, record and replay only records user interactions. However, there are a lot of other factors that influence the behavior of a web application. In the future, record and replay could be extended to support recording of other non-deterministic factors. Those factors could for example include:

- Dates
- Random numbers
- Server responses
- Interrupts by `setTimeout` and `setInterval`

Recording additional data could lead to more accurate replaying of events and could help reproduce some bugs that could not be reproduced so far. Furthermore, the replaying of events could be improved: So far, if data is recorded on one device and then replayed on another device, the coordinates of things like click events are kept. The same applies for scrolling where the scrolling position is set manually. However, if devices have different resolutions, this could lead to inaccurate replaying of events. Ideally, the events would be adjusted to the target resolution before replaying.

8.1.2 Extended Device Emulation

The device emulation in XDTools only emulates the resolution of the devices. However, the resolution is not the only difference between a desktop device and mobile devices. More things could be considered when emulating devices for a more realistic experience:

- Touch interactions
- Network conditions
- Location
- Input from sensors
- Hardware performance

Although an emulated device will never be completely equivalent to a real device, including more factors into device emulation can enable more sophisticated testing on emulated devices. Emulating things like location and network conditions can even reveal issues that are not necessarily found when testing on real devices. Testing on real devices typically still happens inside the office, a somewhat artificial setting. In the office, network conditions will typically be good and sensor readings like location do not change much. If those things can be emulated, testing applications in the office can become more realistic and useful.

8.1.3 Tighter Integration with Browser

It would be very useful if XDTools could be integrated more with the debugging tools provided by browsers and also with the browser itself. Even though we actually implemented a Chrome DevTools extension, the access to the debugging tools is still rather limited. Tighter integration with the debugging tools could further improve the debugging capabilities, especially concerning JavaScript debugging: Within XDTools, it was only possible to add breakpoints at the beginning of functions and functions that are not globally accessible cannot be debugged at all. In the ideal case, it would be possible to set a breakpoint on one device and automatically transfer it to other devices if desired.

Furthermore, our shared JavaScript console and CSS editor provide less features than the console and CSS editor included directly in the browser. If our aggregation features would be integrated directly into the browser debugging tools, more extensive debugging would be possible.

If XDTools would be integrated directly into the browser, the DNS server would not be required anymore. Instead, the browser could just make sure that different devices do not share data. With everything from XDTools included directly into the browser, no installation of additional components would be required anymore and debugging cross-device applications would be possible directly from the browser's debugging tools.

8.1.4 Long-Term Study

Although our user study yielded promising results, it obviously was an artificial setting: The tasks were rather short and the participants were not familiar with the application they worked with. Thus, they also spent some getting to know the application and understanding the code. Also, the participants only had access to a subset of the code of the entire application. In the real world, developers typically deal with a much larger amount of code. Additionally, the instructor was always sitting next to the participants and a camera was recording the

participants. This might influence the behavior of the participants. Finally, some features were disabled for the user study and could not be evaluated yet.

It would be very useful if a long-term study could be conducted to see how developers use XDTools in the wild. Giving developers that are in the process of developing a cross-device application access to XDTools during the entire course of development would help further evaluate XDTools. The developers would be in their natural "habitat" and could use XDTools in a way that best fits their needs. The developers could provide feedback during the development of the application and the actual usage of XDTools could be recorded and analyzed to gain further insights. After finishing development of their application, developers could be interviewed, giving them a chance to tell us what they think of XDTools and provide suggestions for further improvements. Conducting such a long-term user study could be very helpful when releasing future versions of XDTools. As a first step for further evaluation and improvement of XDTools, we plan to make XDTools available on Github[1] in the near future. Hopefully, some developers can already benefit from XDTools if it is available to the public and we might also receive some valuable feedback.

[1] `https://github.com/`

A
Questionnaires

General Questions

* Required

Year of birth *

Gender *

- ○ Male
- ○ Female

How do you rate your web application development skills? *

	1	2	3	4	5	
Basic	○	○	○	○	○	Proficient

How many years of experience do you have in web application development? *

- ○ Less than 2 years
- ○ 2-5 years
- ○ More than 5 years

Do you already have some experience with developing responsive web applications? *

- ○ Yes
- ○ No

If yes, how did you test the design of the responsive web application(s)?

- ☐ Browser tools for emulating devices (e.g. Chrome Device Mode)
- ☐ Real devices
- ☐ Other:

Do you already have some experience with cross-device application development? *

- ○ Yes
- ○ No

If yes, how did you test the cross-device web application(s)?

- ☐ Browser tools for emulating devices (e.g. Chrome Device Mode)
- ☐ Real devices
- ☐ Multiple browser profiles

☐ Other:

How do you rate your JavaScript skills? *

	1	2	3	4	5	
Basic	○	○	○	○	○	Proficient

How many years of experience do you have in JavaScript? *

- ○ Less than 2 years
- ○ 2-5 years
- ○ More than 5 years

Do you already have some experience with Chrome Developer Tools? *

- ○ Yes
- ○ No

If yes, how often have you used the following features of Chrome Developer Tools?

	Never	Sometimes	Often
Device Mode	○	○	○
JavaScript Console	○	○	○
JavaScript Debugging	○	○	○
HTML/CSS Inspection	○	○	○

Submit

Never submit passwords through Google Forms.

XDCinema: Displaying and Highlighting Prices

* Required

It was easy to complete the task with the tools I had access to. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I felt efficient completing the task with the tools I had access to. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

It was challenging to complete the task with the tools I had access to. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

The tools I had access to were well suited for completing the task. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

How useful did you find the following features for completing the task? *

	1: Not useful	2	3	4	5: Very useful	Not used
Device emulation	○	○	○	○	○	○
Connecting of real devices	○	○	○	○	○	○
Connection features	○	○	○	○	○	○
Shared JavaScript console	○	○	○	○	○	○
Function debugging	○	○	○	○	○	○
Shared CSS editor	○	○	○	○	○	○

Comments

Submit

Never submit passwords through Google Forms.

XDCinema: Displaying and Highlighting Prices

* Required

It was easy to complete the task with the tools I had access to. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I felt efficient completing the task with the tools I had access to. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

It was challenging to complete the task with the tools I had access to. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

The tools I had access to were well suited for completing the task. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

Comments

Submit

Never submit passwords through Google Forms.

Concluding Questionnaire

* Required

Did you find it easier to debug with or without the tool? *

- With the tool
- Without the tool
- Neither

Did you feel more efficient debugging with or without the tool? *

- With the tool
- Without the tool
- Neither

Did you prefer debugging with or without the tool? *

- With the tool
- Without the tool
- Neither

Did you find it easier to implement a feature with or without the tool? *

- With the tool
- Without the tool
- Neither

Did you feel more efficient implementing a feature with or without the tool? *

- With the tool
- Without the tool
- Neither

Did you prefer implementing a feature with or without the tool? *

- With the tool
- Without the tool
- Neither

It was easy to learn how to use the tool. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I felt confident using the tool.

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

The tool was unnecessarily complex.

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

The tool would be useful for debugging cross-device applications. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

The tool would be useful for implementing cross-device applications. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

I would use the tool for debugging cross-device applications. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agreee

I would use the tool for implementing cross-device applications. *

	1	2	3	4	5	
Strongly disagree	○	○	○	○	○	Strongly agree

Which of the following features would you use for debugging cross-device applications? *

- ☐ Device emulation
- ☐ Connecting of real devices
- ☐ Connection features
- ☐ Shared JavaScript console
- ☐ Function debugging
- ☐ Shared CSS editor

Which of the following features would you use for implementing cross-device applications? *

- [ ] Device emulation
- [ ] Connecting of real devices
- [ ] Connection features
- [ ] Shared JavaScript console
- [ ] Function debugging
- [ ] Shared CSS editor

Do you have any comments about the existing features of our tool?

Do you have any suggestions for additional features that would be useful for our tool?

Do you have any other comments?

Submit

Never submit passwords through Google Forms.

B

Tasks

XDYouTube: Video Playing Bug

Unfortunately, your friends discovered a bug in your cross-device YouTube application, XDYouTube. In some cases, if they want to add a video to the queue, nothing happens. You promise your friend to have a look at the problem as soon as possible.

1. Fixing the Bug

Detailed description: When searching for a video, you get a list of videos as a result. For each video, there is a button that can be clicked to add the video to the queue. Videos from the queue are played one after another. If no video is in the queue and a video is added to the queue, the video is played immediately. The consequences of the bug are that even though you can still search for videos, nothing happens if you click the button to add the video to the queue.

Your task is to fix the bug. First, try to reproduce the bug described above. If you successfully manage to reproduce the bug, try if you can find the cause of the bug and fix it. You are given a file with one function; the bug must be somewhere in that function.

XDYouTube: Remote Control

You often use the cross-device YouTube app XDYouTube with your friends to watch videos together. Everyone can already queue new videos from their device and see the current video as well as the queued videos. However, everyone needs a break from watching videos sometimes, and you would prefer if you could pause the current video remotely from your mobile device instead of going to the screen that shows the video and pausing the video from there. As you developed the app yourself with a few friends, you decide to implement this feature.

1. Implementing Remote Control Functionality

If you look at the landscape view of a controller device, you see an overview of the current video and the queued videos. Above the current video, we already added a button with the id “pauseButton” that should be used for the remote control functionality. We also have a shared variable that states whether the video is currently playing or paused. Each device can have the role “controller” and/or “player”. Devices with the role “player” are responsible for playing the videos and devices with the role “controller” are responsible for queuing/controlling videos. Your task is to implement to functions:

- pauseClicked(): This function is called whenever a user clicks the play/pause button. If no video is currently playing, the function should not do anything. If video is playing (paused or not paused), it should change the state of the shared variable.
- updatePaused(): This function is called on all devices whenever the state of the shared variable changes. On devices that have the role “controller”, it should change the text of the control button to “PAUSE” or “PLAY” depending on the current state of the video. On devices that have the role “player”, it should play/pause the video. Remember that a single device can have both roles.

You have access to the following helper functions:

- that.pauseVideo(): pauses the video
- that.unpauseVideo(): unpauses the video
- that.isVideoPlaying(): returns if any video is currently active (even if it is paused)
- that.setPausedState(state): sets the state of the shared variable
- that.getPausedState(): returns the state of the shared variable (true -> video is paused, false -> video is playing)
- that.isPlayer(): returns “true” if the device has the role “player”
- that.isController(): returns “true” if the device has the role “controller”

2. Styling the Remote Control Button

Your very talented friend has designed a more fashionable button for your remote control functionality. He sent you an image of how the button should and asked you to integrate it into the application. Your task is to add the appropriate CSS such that the remote control button looks similar to the button in the image below (it does not need to be exactly the same). Additionally, you want to make the button responsive so it looks good on all kinds of devices and some of your friends use your application on their low-resolution smartphones while others use it on their Full HD laptops. The button needs to look (somewhat) good on resolutions up to 1920x1080.

XDCinema: Location Bug

As one of the developers of XDCinema, your job is also to fix bugs that were submitted by users. One user noticed a particularly strange bug: When clicking on a cinema, the location of the cinema that is displayed is completely wrong. Since this severely impacts the usefulness of the application, you decide to fix the bug right away.

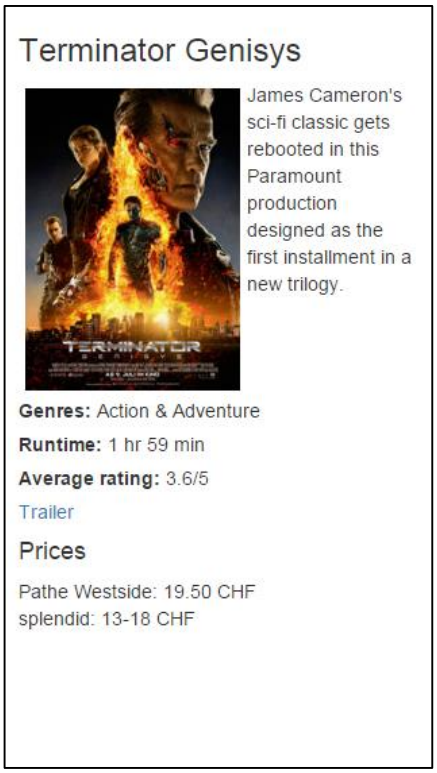
1. Fixing the Bug

Detailed description: When searching for movies in a city, you can click on a cinema to display the location of the cinema on a map. However, the location that is displayed is wrong for most locations even though the data from the database is correct.

Your task is to find the cause of the bug and fix it. You are given a file with a few functions; the bug must be in one of those functions.

XDCinema: Displaying and Highlighting Prices

Different cinemas have different prices. Before someone decides which cinema to visit, they might like to compare prices so they can save money. Instead of visiting the website of each cinema and navigating between multiple pages to find the page that displays the prices, they would prefer to have an overview of all prices in his cross-device cinema application, XDCinema. They send a request to the developers of the app and you, as a developer, decide to implement the feature.



1. Displaying Prices

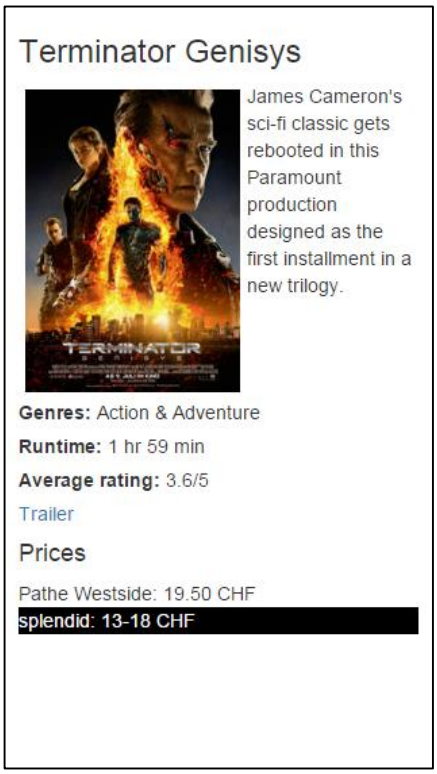
Whenever the user clicks on a movie, you want to display the prices of all cinemas that show the movie in the city that the user has currently selected. This price overview should be displayed below the other information about the movie, similar to the screenshot to the left.

Your task is to implement the function “displayPrices(city, movie)”. This function should update the price overview whenever the user clicks on a movie.

The function is already called at the appropriate locations in the existing code, thus you only have to implement it. You can display all prices inside an existing “div”-element with the id “prices”.

You have access to the following helper functions:

- getCinemas(movie, city): This function returns a list of names of all cinemas that show a certain movie in a certain city.
- getCinemaPrice(cinemaName, city): This function returns the ticket price of a certain cinema in a certain city.



2. Highlighting Prices

To make it even easier for the user to locate the correct price, we want to highlight the price of the appropriate cinema whenever the user selects a new cinema, similar to the following screenshot to the left.

Your task is to implement the function “highlightPrice(cinema, city)”. The function should highlight the price of a certain cinema in a certain city. Again, the function is already called at the appropriate locations and you only have to implement it.

List of Figures

List of Tables

Acknowledgements

TODO

Bibliography

[1] Silviu Andrica and George Candea. Warr: A tool for high-fidelity web application record and replay. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 403–410. IEEE, 2011.

[2] Sriram Karthik Badam and Niklas Elmqvist. Polychrome: A cross-device framework for collaborative web visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, pages 109–118. ACM, 2014.

[3] Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, pages 473–484. ACM, 2013.

[4] Pei-Yu Peggy Chi and Yang Li. Weave: Scripting cross-device wearable interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3923–3932. ACM, 2015.

[5] Steven Houben and Nicolai Marquardt. Watchconnect: A toolkit for prototyping smartwatch-centric cross-device applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1247–1256. ACM, 2015.

[6] Dejan Kovachev, Dominik Renzel, Petru Nicolaescu, and Ralf Klamma. Direwolf-distributing and migrating user interfaces for widget-based web applications. In *Web engineering*, pages 99–113. Springer, 2013.

[7] James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, volume 10, pages 159–174, 2010.

[8] Michael Nebeling, Maria Husmann, Christoph Zimmerli, Giulio Valente, and Moira C Norrie. Xdsession: integrated development and testing of cross-device applications. *Proc. EICS*, 2015.

[9] Michael Nebeling, Theano Mintsi, Maria Husmann, and Moira Norrie. Interactive development of cross-device user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2793–2802. ACM, 2014.

[10] Stephen Oney and Brad Myers. Firecrystal: Understanding interactive behaviors in dynamic web pages. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, pages 105–108. IEEE, 2009.

[11] Stephanie Santosa and Daniel Wigdor. A field study of multi-device workflows in distributed workspaces. In *Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing*, pages 63–72. ACM, 2013.

[12] Mario Schreiner, Roman Rädle, Hans-Christian Jetter, and Harald Reiterer. Connichiwa: A framework for cross-device web applications. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2163–2168. ACM, 2015.

[13] Jishuo Yang and Daniel Wigdor. Panelrama: enabling easy specification of cross-device web applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2783–2792. ACM, 2014.

[14] Asim Yildiz, Bariş Aktemur, and Hasan Sözer. Rumadai: A plug-in to record and replay client-side events of web sites with dynamic content. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 88–89. IEEE, 2012.