Testing 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

29th September 2015

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

globis

Abstract

Here comes the abstract.

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Contributions 3

1.3 Structure of this Document 4

2 Background and Related Work **5**

2.1 Web Application Testing 6

2.1.1 Browser-Integrated Debugging Tools 6

2.1.2 Record and Replay 7

2.2 Responsive Web Application Testing 9

2.2.1 Web Services 9

2.2.2 Testing with Actual Devices 9

2.2.3 Device Emulation 11

2.2.4 Summary 11

2.3 Cross-Device Application Development Frameworks 12

2.3.1 XD-MVC 12

2.3.2 Connichiwa 12

2.3.3 DireWolf 12

2.3.4 XDStudio 13

2.3.5 Panelrama 14

2.3.6 Polychrome 14

2.3.7 XDSession 15

2.3.8 WatchConnect 15

2.3.9 Weave 15

2.3.10 Summary 16

3 Approach **17**

3.1 Emulation of Multiple Devices 18
3.2 Easy Integration of Real and Emulated Devices 21
3.3 Easy Switching of Device Configurations 22
3.4 Integration with Debugging Tools 23
3.5 Automatic Connection Management 25
3.6 Coordinated Record and Replay 26

4 Implementation **27**
4.1 Architecture 28
4.2 Choice of Technologies 29
4.2.1 Server Side 29
4.2.2 Client Side 30
4.3 Overview of Features 31
4.4 Emulation of Multiple Devices 32
4.4.1 Color Generation 34
4.5 Easy Integration of Real Devices 35
4.6 Easy Switching of Device Configurations 36
4.7 Integration with Debugging Tools 36
4.7.1 Shared JavaScript Console 36
4.7.2 Function Debugging 38
4.7.3 Shared CSS Editor 40
4.8 Automatic Connection Management 41
4.9 Coordinated Record and Replay 42
4.10 Integration with Polymer 45

5 Sample Applications **47**
5.1 XDCinema 48
5.1.1 Implementation 49
5.2 XDYouTube 50
5.2.1 Implementation 52
5.3 Insights 52

6 Evaluation **55**
6.1 Setup 56
6.1.1 Participants 57

6.1.2 Tasks 59
6.1.3 Evaluation Methods 61
6.2 Results 63
6.2.1 XDCinema: Fixing a Bug 63
6.2.2 XDCinema: Implementing a Feature 65
6.2.3 XDYouTube: Fixing a Bug 66
6.2.4 XDYouTube: Implementing a Feature 67
6.2.5 General 69
6.3 Discussion 70

7 Conclusion **79**
7.1 Future Work 80

A Questionnaires **81**

1
Introduction

1.1 Problem Statement

Despite the abundance of devices nowadays, up until recently there was no way of sharing state information and I/O resources between devices. Although many users have access to multiple devices at the same time, e.g. their smartphone and laptop, those devices are mostly used independently. In the last few years, cross-device applications have have started to fill this gap by facilitating the use of multiple devices at once and the sharing of data between them. 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 cross-device applications. Such applications typically run on any device that has access to a modern web browser. Santosa et al. [11] have observed in a field study that users already use multiple devices in parallel in their workflows and that better functional coordination is needed. 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 and 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 application development frameworks have little support for this and either provide no facilities for testing applications at all or only very basic facilities that are focused on specific aspects of the applications.

In contrast, there are already plenty of practical tools for testing and debugging web applications in general and many of them can be accessed directly from modern browsers. Google Chrome[3] in particular, but to some extent also other browsers, provide quite mature tools for debugging JavaScript, HTML and CSS. Unfortunately, those tools are focused on debugging one device at a time, limiting their usefulness for testing cross-device applications where multiple devices are 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 has made it a requirement 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 of such websites; some are already built into modern web browsers while others can be accessed through a website or by installing a program on a desktop PC. In summary, those tools use two different approaches for testing responsive websites: First, different devices can be emulated on a desktop computer. Second, a varied set of actual devices can be used. Google Chrome's Device Mode[4] provides extensive support for emulating devices; apart from simply 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 to facilitate this. Some allow the developer to reload all devices at once, e.g. Adobe Edge Inspect CC[5], while others auto-

[1] http://www.w3.org/2009/dap/
[2] http://www.webrtc.org/
[3] http://www.google.com/chrome/
[4] https://developer.chrome.com/devtools/docs/device-mode
[5] https://www.adobe.com/ch_de/products/edge-inspect.html

matically reload devices when files change, e.g. BrowserSync[6]. Some of those tools even allow developers to simultaneously browse the 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, cross-device applications and web applications targeted at one device at a time 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, which requires the developer to open multiple browser windows, possibly with different user profiles or in incognito mode, as cross-device applications only become useful when multiple devices are involved at once. 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 those 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.

1.2 Contributions

Our project aims to make it easier to test and debug cross-device applications. As a first step, we analyzed existing tools for debugging web applications. This includes tools built directly into browsers as well as external tools. Due to the similarities between cross-device applications and responsive web applications, tools for testing responsive websites are of particular interest. During the analysis, we gathered the limitations of those tools and possible solutions to those limitations regarding cross-device application testing. Furthermore, we also investigated some frameworks for developing cross-device applications. The inspection of those frameworks was done for two reasons: Firstly, we wanted to see if any of the frameworks already provide some useful for cross-device application testing. Second, we wanted to analyze how different frameworks could profit from a tool that facilitates cross-device application testing. Finally, we gathered some requirements for more suitable tools for cross-device application testing based on the limitations of existing tools.

Based on those requirements, we designed and implemented a new set of tools for testing and debugging cross-device applications. Those tools allow testing applications both on real devices and emulated devices. Our tools provide a number of different features: Some features have been adopted from other tools while others are based on some existing features but have been extended to suit the needs of cross-device application testing. Some features that only make sense in a cross-device environment are completely new and not based on any available features.

[6] http://www.browsersync.io/
[7] http://crossbrowsertesting.com/

During the development of our tools, we also implemented two sample cross-device applications using XD-MVC[8], a cross-device application development framework. Those applications should helped us in multiple ways while developing our tools: First, actually using our tools for developing cross-device applications provided some new ideas for crucial features that were still missing from our tools. Second, developing sample applications will helped us improve existing features and revealed some bugs in existing features. Finally, we will also learned something about the usability of our tool.

Eventually, we conducted a user study to evaluate the usefulness and suitability of the tool, comparing to traditional methods of testing web applications. During this study, participants got the opportunity to use our tools for implementing a new feature in a cross-device application and for finding and fixing a bug. This study showed that our tools are indeed considered useful for testing cross-device applications by our participants. Furthermore, some participants also provided some new ideas for features and improvements for our existing features as well as the layout of our tools.

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 a set of tools for testing and debugging cross-device applications.
- Development of two sample cross-device applications.
- User study for evaluating our set of tools.

1.3 Structure of this Document

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

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

In Chapter 3, we describe our approach at achieving our goals and the requirements gathered during the analysis of related work.

In Chapter 4, we will describe the implementation of our set of tools in detail.

In Chapter 5, we will describe the sample applications that we developed and the insights we gained from them.

In Chapter 6, we will describe the user study and explain the results of it.

In Chapter 7, we conclude the work by showing what was achieved and what problems remain, and address possible future work that might be based on our existing set of tools.

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

2

Background and Related Work

2.1 Web Application Testing

2.1.1 Browser-Integrated Debugging Tools

Chrome DevTools

Google Chrome already provides a wide range of features for testing and debugging web applications. First of all, it lets the developer inspect the DOM tree and allows inspection and on-the-fly editing of DOM elements. Furthermore, new CSS rules or properties can be added or existing ones can be modified. 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. In the DevTools, you can see all the 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.

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, the main disadvantage is that they can only be used to debug one device and one website 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 will not be explained 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 access a number of developer tools from within Firefox. 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.

Summary

Internet Explorer, Microsoft Edge, Opera, Safari 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.

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

2.1.2 Record and Replay

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 nondeterministic inputs. Developers can use Timelapse to browser, visualize, and seek within recorded program executions 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.

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 on his 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. Thus, the recording functionality is an integral part of the browser. 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[2].

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

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 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 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.

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 also provides basic 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, 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 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.

Record and replay mechanisms have already been shown to help with debugging in traditional web applications, thus they could certainly also be useful for debugging cross-device applications. In particular, we believe that replaying event sequences on multiple devices simultaneously could help simulating 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 simultaneously a very difficult task. If multiple devices should 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 more on testing on real devices and others focus on emulating devices. In the following subsections we will describe some of those tools.

2.2.1 Web Services

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

BrowserStack

BrowserStack[3] 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. Advantages of BrowserStack are that real iOS devices are used for screenshots and interactions can be tested automatically using Selenium. A major disadvantage is that only emulated Android devices are available, which limits the usefulness of the tool. Furthermore, only one device can be live tested at a time which is a huge disadvantage for cross-device application testing.

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. The usefulness of the 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.

2.2.2 Testing with Actual Devices

Using the following tools, developers can connect and debug real devices.

Remote Preview

Remote Preview[4] allows synchronizing URLs across multiple devices. This allows fast previewing of a website on multiple devices. In cross-device scenarios, it may provide especially

[3] `https://www.browserstack.com/`
[4] `https://github.com/viljamis/Remote-Preview`

useful for quick connecting of devices in applications where devices are connected simply by copying the same URL to all devices. However, the fact that this tool only provides one feature, namely URL synchronizing, limits its usefulness.

BrowserSync

BrowserSync 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 or refresh all devices as well as automatically refresh devices when files are changed. Furthermore, it allows synchronizing interaction 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 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.

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. 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.

Ghostlab

Ghostlab[5] is one of the more mature tools for website testing and provides features similar to Browser Sync. 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

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

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 developers to emulate devices.

Chrome Device Mode

Google Chrome provides extensive support for emulating devices with its Device Mode. If the developer opens the DevTools, they can switch to the Device Mode and emulate any device. The developer can select an arbitrary device from a large 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 maybe touch, Chrome Device Mode allows to emulate many other aspects as well.

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. First of all, there are two different approaches to live testing in the tools described above. The first is to test on real devices, but through a web service. In those tools, live testing is possible on one device at a time, but in cross-device scenarios, multiple devices are typically involved. The second approach is to let the developer connect their own devices and synchronize interactions among all or some devices. 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, but this is already covered by Google Chrome anyways. Something similar to Selenium testing would clearly also be 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.

2.3.1 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.

2.3.2 Connichiwa

Connichiwa is a framework for developing cross-device web applications developed by Schreiner at al. [12]. It runs local web applications on one of the devices without requiring an existing network or internet connection. It has four key goals:

- Integration of existing devices: Devices should be supported without the need to augment them with additional hardware, markers, or tags
- Independence of network infrastructure: No remote server is used, therefore the device neither needs to be in the same network nor have a permanent internet connection
- Usability of its API

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.

2.3.3 DireWolf

DireWolf is a framework for distributed web applications based on widgets. It was developed by Kovachev et al. [6] Widgets can be shared, reused, mashed up and personalized between applications. Splitting the interface into separate widgets and enabling them to exchange information allows the development of customizable web applications. It makes the following contributions:

- They provide a framework for easy browser-based distribution of Web widgets between multiple devices.

- They facilitate extended multi-modal real-time interactions on a federation of personal computing devices
- They provide continuous state-preserving widget migration

DireWolf helps managing a set of devices and handles communication and control of distributed parts of the web application. Local inter-widget communication is used to communicate with widgets running in the same browser context and remote inter-widget communication provides the message-exchange mechanism for widgets located at different devices.

2.3.4 XDStudio

In [9], Nebeling et al. present their web-based GUI builder, XDStudio. 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:

- Use mode: The user can interact normally with the interface loaded into the editor
- Design mode: The user can manipulate the interface directly

XDStudio makes the following contributions:

- It explores two scenarios thatt were developed as interesting use cases of multi-device, distributed user interfaces and used to drive the design and evaluation of XDStudio
- The notion of distribution profiles at the core of XDStudio extends existing context models to multiple devices and users
- It provides cross-device authoring concepts and tools that cater for cases where not all devices and users are available, or where they are even different, at design and run-time
- User study to evaluate XDStudio

Distribution profiles can be specified in terms of involved devices, users and target user interfaces. The user can either select which user interface widgets and other elements of the source to include in the distribution by dragging and dropping them from the source interface to the corresponding target interface or click a button available for each target interface to insert the full version of the source interface as a starting point, and then include/exclude interface elements from the distribution.

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 like expected is still a difficult task.

2.3.5 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. An existing application can be converted to Panelrama using the following steps:

1. Wrap grouping of HTML UI elements within a ”panel” tag
2. Complete a Panelrama definition including by selecting the state information for synchronization and rating each panel for its needs with respect to various device characteristics
3. Modify the business logic such that it accesses the new synchronized state information

2.3.6 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:

- Co-browsing 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
- Maintenance of state and input events on a server to handle common issues with distributed applications such as consistency management, conflict resolution, and undo operations

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. There are three different modes for sharing:

- Explicit sharing: The user decides when the operations should be shared, which operations to share, and whom to share them with
- Explicit sharing: The user decides when the operations should be shared, which operations to share, and whom to share them with
- Implicit sharing: Operations are automatically shared with all connected devices
- Unilateral sharing: One device (the leader) always shares its operations automatically, while the other devices only listen to the operations shared by the leader

2.3.7 XDSession

XDSession, developed by Nebeling et al. [8], is a framework for cross-device application development based on a concept of cross-device sessions which is also useful for logging and debugging. 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/replay of cross-device sessions. A session consists of users, devices, and information. When a device joins a session, it receives the whole or missing part of the data belonging to the selected session. When a device leaves a session, all data of that session that is not shared with any other session is removed from the device.

XDSeession provides a capture and replay mechanism for user interactions and changes to the 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 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.

2.3.8 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.

2.3.9 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's APIs capture affordances of wearable devices and provide mechanisms for distributing output and combining sensing events and user input across multiple 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.

2.3.10 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, many of those frameworks provide no support for testing and debugging the applications that are developed with them. Applications DireWolf, Polychrome and XD-MVC all include no features for testing and run in an unmodified browser. Thus, they could all 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 in 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
Approach

In the previous section, we have already described some features that could also provide useful in a cross-device testing scenario. Those features include:

- Refreshing all devices/Loading a URL on all devices
- Device Emulation
- Testing on Real Devices
- Recording and replaying interactions
- Many features that are already included in browser developer tools

While some of those features like refreshing all devices are already very useful in cross-device testing scenarios and can be adopted without much adjustment, other features like device emulation still have some limitations that make them difficult to use for cross-device testing and have to be extended in some ways. In the following subsections, we will describe the core concepts of our system and how we adapted existing features to suit the needs of cross-device testing.

3.1 Emulation of Multiple Devices

Device emulation is already often used for testing responsive websites. In its simplest form, device emulation can just be done by resizing a browser window so it looks and feels like a smaller device. However, manually resizing a browser window such that it has a resolution that is typical for actual devices is difficult. Furthermore, just emulating the screen size is not enough to realistically emulate a real device: Mobile devices typically use touch interactions and often have poor network connectivity. Those limitations have led to the emergence of more sophisticated tools for device emulation. Advanced device emulation facilities like 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 a large number of existing devices, thus realistic resolutions can be emulated. However, all those tools have one common limitation: They can emulate only one device per browser tab or window. Another limitation of those tools is that even when multiple browser windows are used, those browser windows share their local resources such as local and session storage. This limits the use of those tools for cross-device applications, as cross-device applications usually run on multiple independent devices and thus should not share any local resources. To accurately simulate the use of a cross-device application in real life, some mechanism for preventing devices from sharing local resources is needed. Nowadays, there are several possibilities for doing this: First, different browser profiles can be used in different browser windows. Second, the incognito mode provided by 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 limits the number of devices that can be emulated to a rather small number. Furthermore, all of those solutions require multiple windows that need to be arranged somehow and tasks like creating multiple profiles might be time-consuming and not what the developer wants. This is tedious and frequent switching between browser windows is required. Additionally, if the

developer actually wants to use browser debugging tools, those tools have to be opened in all different windows which requires a lot of space and also limits the number of devices that can realistically be emulated. The screen size of the device that is used for emulating the devices can also be a limitation in other scenarios: If the device has a full HD screen but 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 have to put one window in full screen. This would make switching between devices even more tedious and the consequences of the interactions performed on the full HD device could not be seen in real-time because the developer would have to switch to the other windows first. Also, things like emulating a 4K TV are impossible on a device with a smaller resolution.

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 our system:

- It should be possible to emulate multiple devices in one browser window.
- The emulated devices should not share any local resources.
- The screen size should not be a major limiting factor concerning the number of devices that can be emulated simultaneously.
- The developer should have access to a list containing some existing devices.

Our system provides a solution that fulfills all those requirements. Firstly, the user can add multiple emulated devices simultaneously. The developer can either select the emulated devices from a list of existing devices or create a custom device. The user 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 essentially 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, 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 types of different 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 very large range of devices is covered, as devices in the same category typically have similar properties such as screen size and input modalities. Our list of existing devices includes multiple devices from each of those categories. For the desktop devices, we just provide some typical resolutions of desktop PCs and laptops. The list of 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).

Once the developer has created an emulated device, 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 easy 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 scale down an emulated device and have more space available for devices. If the developers 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. Thus, the developer can continuously increase or decrease the resolution of the device to immediately see how the application looks with different resolutions. Figure 3.1 illustrates the difference between resizing a device and scaling a device using W3School's website[1].

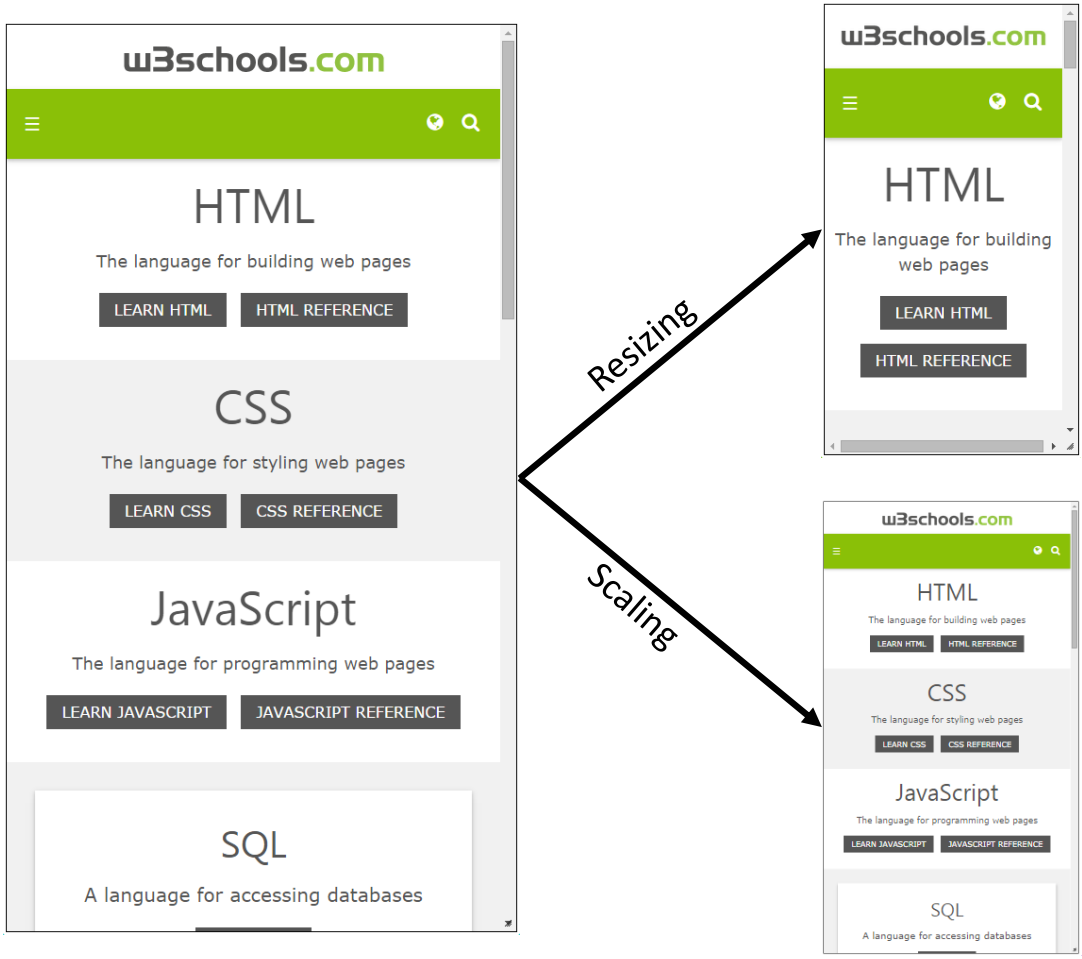

Figure 3.1: Difference between resizing and scaling a website

[1] http://www.w3schools.com/

Finally, our system includes a mechanism that prevents the sharing of local resources between emulated devices. In summary, our system allows the developer to create multiple emulated devices with just a few clicks and to order the devices as needed for the application that the developer wants to test. There is no need for constant switching between multiple browser windows and creating new browser profiles. Also, the developer tools can just be opened for the browser window where all devices are emulated instead of having to open them for each emulated device.

3.2 Easy Integration of Real and Emulated Devices

Emulating devices is a versatile tool for testing cross-device 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, location, and more. However, not every little detail of a real device can be emulated accurately. The following list provides an overview of some of the limitations regardinf 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 realistically on an emulated device.
- Interrupts: While using an application on a real device, the user might be interrupted by things like the arriving 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 uses too much performance for most mobile devices, this might not even be noticed if the developer only test 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 differences between real and emulated devices, this list 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 our system: It should easily be possible to connect real devices to our system. QR codes have become increasingly

popular over the last few years and almost all devices nowadays are equipped with at least one camera. Thus, our system provides a QR code that can be scanned with real devices to connect them to our system. This makes it easy and efficient to connect a large number of real devices to our system. As a backup mechanism, the developer can also type a URL into the browser of a real device to connect it to our system. Once a real device is connected to our system, it is represented with a proxy and can be used in the same way as an emulated device. The developer can also use emulated devices alongside real devices if desired. Thus, our system allows the user 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.

3.3 Easy Switching of Device Configurations

Cross-device applications are typically not always used by the same users and with the same set of devices. 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 using a cross-device application and their characteristics may vary greatly. Depending on those devices, the UI distribution might be different. A cross-device application needs to be able to support all those different device scenarios. However, some cross-device applications are also targeted to specific scenarios, e.g. a presentation room with multiple big screens that are always present, in addition to some mobile devices that are only in the room when their owner is attending a presentation. In such an application, the developer would probably want to emulate the devices that are always present when the application is used whenever they are testing the application. Thus, it is a requirement for our system that multiple different device scenarios can quickly be created.

Our system allows the developer to create a device configuration, save it for later, and then re-use it. A device configuration consists of the following information:

- The number of emulated devices.
- The types of the devices.
- The position and scaling of the devices.

Thus, the developer can easily re-use device configurations without much effort and switch between different device configurations efficiently. Thus, testing a cross-device application in 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 are saved in the configuration, e.g. in the example mentioned above, the developer could create a device scenario where only the big screens are already configured and then dynamically add some mobile devices to create a more realistic scenario.

3.4 Integration with Debugging Tools

Many of the features integrated into the debugging tools of browsers are also immensely useful for testing cross-device applications, some might be even more useful for testing cross-device applications than for testing traditional web applications. However, those tools are typically limited to debugging one device at a time. In our system, multiple devices are emulated in the same browser window. While this simplifies some ways of debugging multiple devices, e.g. because messages logged from all devices are shown in the same console, it makes other things more difficult, such as trying out things in the console by sending commands. 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 devices. Furthermore, even though logging messages from all devices are displayed in the console, there is no way of knowing which device sent the message. This further complicates the debugging of cross-device applications. The same limitation applies to JavaScript errors that are also shown in the console, but can not easily be related to a device. Further limitations of the browser-included debugging tools are 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, because the function is not even 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 in Google Chrome. Remote debugging provides the same debugging tools as the normal DevTools, but it gets even more difficult to debug multiple devices at the same time. The remote debugging of each device is opened in a new window, thus the developer once again has to navigate between multiple windows. Also, getting an overview of all logging messages from all devices and sending commands to multiple devices becomes even more difficult. Despite those limitations, a tight integration with browser debugging tools is clearly desirable: Most of those tools have been around for quite some time and thus 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 thus already familiar with them. From this, we can derive the following requirements for our system:

- Logging messages should all be aggregated in one place and the device the messages originated from should also be easily identifiable.
- JavaScript errors should also be aggregated and be easily identifiable.
- It should be possible to send JavaScript commands to multiple devices at a time.
- It should be easy to inspect the HTML of a specific device.
- It should be possible to add CSS to multiple devices at the same time (see Figure 3.2).
- It should be possible to add breakpoints to multiple devices simultaneously.

- If possible, all of the above requirements should also be applied for real devices.

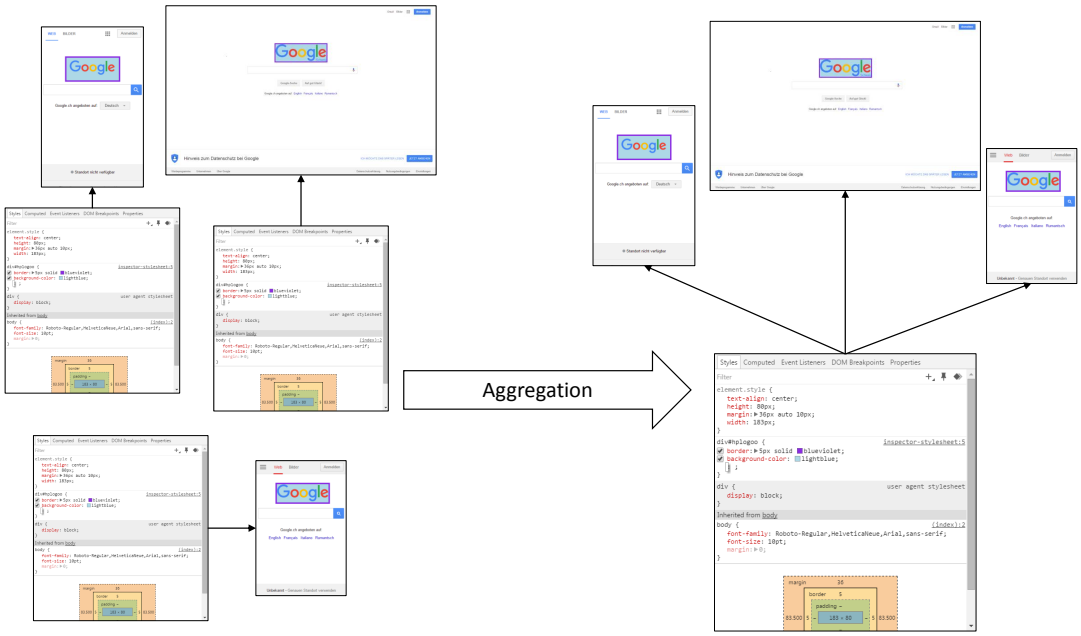

Figure 3.2: Editing CSS once for all devices

Unfortunately, it is not feasible to extend the console of Chrome DevTools due to technical limitations. For this reason, our system includes its own JavaScript console. Each emulated and real device forwards all logging messages and JavaScript errors to our system. This allows us to aggregate all console outputs in our custom console. For easy identifiability, each device gets its own unique color when created and all console outputs are color-coded in the color of the device the output stems from. This color-coding makes it trivial to identify the device a message or error originated from. The large number of messages that are displayed in the console due to the larger number of devices makes it a key requirement that there are some ways of filtering messages. The developer can either filter the messages by type (error, warning, logging message, info), by content, or by device. Our console also allows sending commands to multiple devices. The developer can either send a 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 become trivial. Our custom console works exactly the same for emulated as well as real devices. This aggregation of console outputs from both emulated and real devices at least partially eliminates the need for remote debugging.

Modifying the browser-internal CSS editor such that it adds CSS to multiple devices is not feasible, just like with the JavaScript console. Thus, we also included a custom CSS editor

with our system. The CSS editor is designed to feel similar to the CSS editors typically provided by browsers. Thus, the developer can specify a selector and then add some rules that are applied to HTML elements that match the selector. Again, the developer can either select all devices or a subset of devices. The CSS rules are applied to emulated devices as well as real 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 on all those devices. This is considerably less effort than adding rules to all devices individually or editing the CSS file, saving it and reloading all devices multiple times.

Additionally, our system also allows developers to add breakpoints at the beginnings of functions on all devices or a subset of devices. Breakpoints can easily be added by adding the name of a function to a list of functions that the developer wants to debug. The breakpoints are then automatically added to all activated devices. The developer can also easily inspect the function by clicking on a button. Furthermore, as soon as the functions that the developer is debugging are called on any device, the device where the function was called is highlighted. This makes it easy to identify the device that is currently being debugged. Unfortunately, this feature is limited to emulated devices. The JavaScript files of real devices cannot be accessed from within the browser debugging tools without remote debugging, thus adding a breakpoint to a JavaScript function of a real device is impossible from within our system.

Finally, our system allows the developer to directly jump into the HTML of an emulated device just by clicking a button. Clicking this button navigates through the DOM of our system right into the body element of the emulated device. Again, it is technically impossible to implement this feature for real devices, just like with JavaScript debugging.

3.5 Automatic Connection Management

In order to use multiple devices in a cross-device application, those devices need to be paired with each other in some way. The mechanisms for pairing devices differ between different 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 that should be connected. 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 three 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 a device to connect the device to another device by passing the ID of the device the device should be connected to. 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 even more different ways for connecting devices. Thus, it is impossible to derive all ways in which devices could be connected in a cross-device application.

However, if the developer wants to debug a cross-device application, re-connecting the

devices every time a new device configuration is used or possibly even when devices are refreshed is tedious and time-consuming. Thus, it is desirable that devices can be connected automatically. In order to achieve this, our system provides a custom connection function that can be implemented by the developer. Since the developer probably knows how devices are connected in their application, implementing this function should be trivial. Once this function is implemented, a device can be connected to another device by selecting the device to connect to from a drop-down menu. Furthermore, the developer can choose to automatically connect all newly added devices to an existing device session. Thus, connecting devices to each other becomes an easy task that can be achieved with just a few clicks.

3.6 Coordinated Record and Replay

Record and replay has already 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 recording and replaying interactions in web applications. When a bug is encountered during web application testing, it is often difficult to determine the exact steps for reproducing the bugs. Reproducing bugs becomes even more difficult in cross-device scenarios where multiple devices are involved and the interactions performed on one device have some implications on other devices as well. Also, cross-device applications are often used by multiple users at the same time and it is difficult for the 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.

Our system allows the user to start recording interactions on a 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 then be started on all devices simultaneously. This makes it easy to simulate multiple users and devices in a cross-device environment. 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. The developer can also breakpoints at a certain point of time. As soon as all events that occur before this point of time have been replayed, the replaying pauses and the developer can inspect the state of the devices. Thus, if the developer sees that after some events something goes wrong in the application, they can set a breakpoint, replay the events again and see what went wrong. The developer can also add breaks of one second between events to delay all following events. This allows the developer to spend some more time looking at the application before the replaying continues without having to set breakpoints.

4
Implementation

4.1 Architecture

The architecture of our tools is illustrated in Figure 4.1. Our system 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 side applications running on real devices.

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 our system.

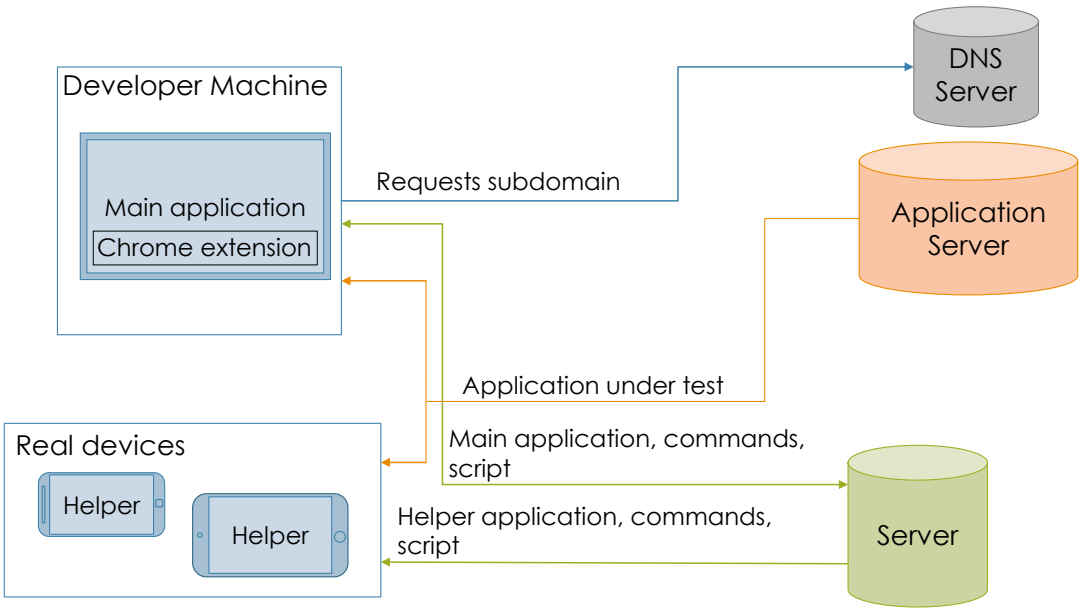

Figure 4.1: Architecture of our tools

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.

Finally, some web server needs to provide the application under test. However, any web server can be used and our system 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.

4.2 Choice of Technologies

Since our set of tools includes desktop devices as well as mobile devices, it should be platform-independent. Thus, our tools are implemented using standard web technologies, i.e. HTML5, CSS3 and JavaScript. This allows our tools 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 our tools. Our tools do not rely on any database, instead HTML5 Local Storage[1] is used for the data that we want to store. Our set of tools only need to store three things:

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

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 and does not require much storage.

4.2.1 Server Side

On the server side, our system use Node.js[2]. Node.js is a JavaScript runtime that uses asynchronous I/O with an event-driven programming model. It uis 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 our application, fast communication between the desktop PC and the connected real devices is required and Node.js fulfills those requirements. Node.js' package ecosystem, npm[3] provides a large number of modules which extend the capabilities of Node.js. We use the following modules in our system:

- Express[4]: Express is a minimal and flexible web application framework that provides a set of features for web and mobile applications. In our system, Express is responsible for serving the content to the clients.
- shortid[5]: shortid generates short URL-friendly unique IDs. shortid generates the device IDs for the emulated devices in our system. As each 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 long-polling. In our system, Socket.io is used for the communication between the server and clients.

[1] `http://www.w3schools.com/html/html5_webstorage.asp`
[2] `https://nodejs.org/en/`
[3] `https://www.npmjs.com/`
[4] `http://expressjs.com/`
[5] `https://github.com/dylang/shortid`
[6] `http://socket.io/`

Socket.io is better suited for our application than AJAX for multiple reasons: First, we need to be able to send push data to clients from the server for 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 our system and AJAX would not fulfill this requirement sufficiently. However, AJAX is used for communicating with the DNS server.

DNS Server

Our system requires a DNS server for registering the subdomains for the emulated devices. It should be easy to dynamically add new subdomains to the DNS server. Furthermore, the DNS server should forward any unknown domains to the standard DNS server. rainbow-dns[7] 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 our application. 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.

4.2.2 Client Side

The client side of our system is implemented using JavaScript with jQuery[8]. jQuery allows easy selection and modification of HTML elements as well as easy event handling, features that are crucial to our system. As an additional helper, we use jQuery UI[9]. jQuery UI is mainly used for the autocomplete functionality and for easy resizing of the individual components of our application. It is also used for resizing the emulated devices to change their resolution. The Twitter Bootstrap[10] framework is used to give our application a nice and uniform look-and-feel. It is also used for facilitating the implementation of things like modal boxes and tabs.

Our system also uses QR codes for connecting real devices to the system. We use the jQuery-based QR Code library jQuery.qrcode[11] 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[12] is used for allowing accurate positioning of devices and timing of event sequences. HTML5 postMessage is used for communication between the testing application in the iframe and our application. Thus, no additional library for communication is needed inside the testing application.

The script that is injected into the application that is being tested is implemented in pure JavaScript. Using libraries like jQuery inside this script could lead to conflicts with other libraries in the applications or different versions of jQuery. Also, loading jQuery into the

[7] https://github.com/asbjornenge/rainbow-dns
[8] https://jquery.com/
[9] https://jqueryui.com/
[10] http://getbootstrap.com/
[11] https://larsjung.de/jquery-qrcode/
[12] http://www.w3schools.com/html/html5_draganddrop.asp

application creates some additional overhead and might not be what the developer of the application wants.

4.3 Overview of Features

In Figure 4.2, the complete interface of our system except for record and replay can be seen.

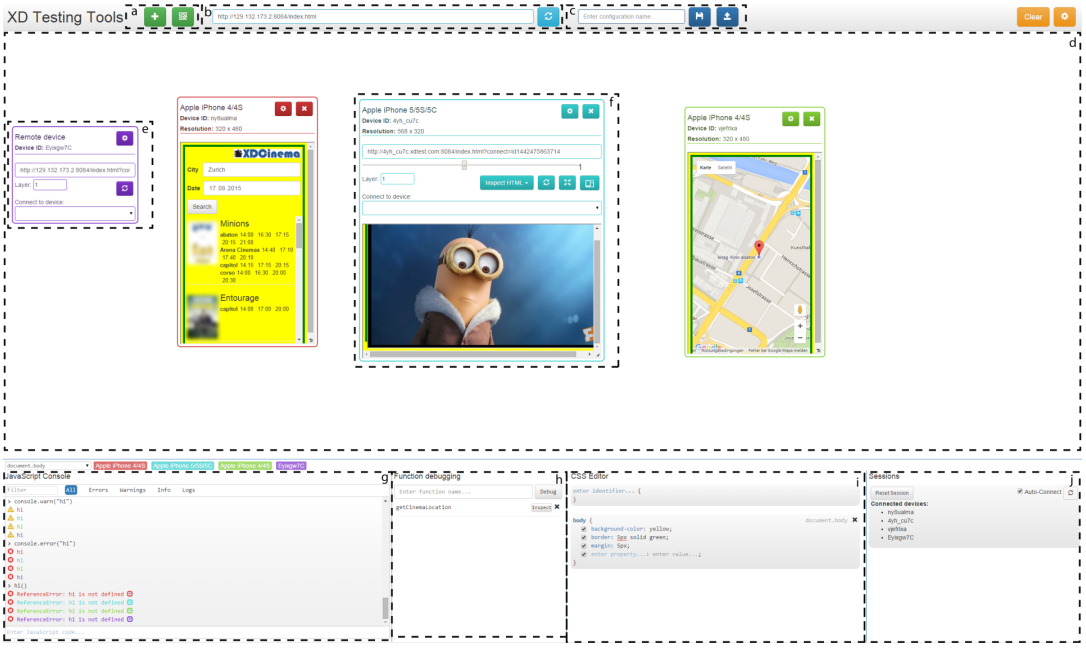

Figure 4.2: The complete interface

The individual features of our system are labeled in the screenshot:

a) Buttons for adding emulated devices and showing the QR code for connecting real devices.

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) An emulated device.

g) The shared JavaScript console.

h) Function debugging.

i) The shared CSS editor.

j) Session management.

Figure 4.3 shows a screenshot of the remote device that is connected to our system in the screenshot shown before.

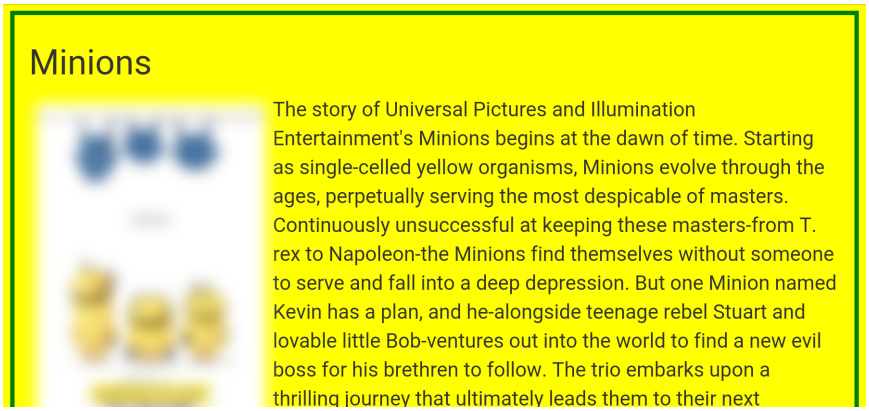

Figure 4.3: The connected remote device

4.4 Emulation of Multiple Devices

Device emulation is realized using iframes: Each emulated device has its own iframe that loads the application that is being tested. However, loading the same domain inside multiple iframes would lead to the sharing of session and persistent data between all emulated devices. We have already described how developers tackle this problem and the limitations of those solutions. For our tools, we created a more scalable and robust solution. When an emulated device is created, it is assigned a unique ID. Based on this ID, a unique subdomain is registered with the local DNS server. All those subdomains will still point to the same application, but as the application is accessed through different domains, no data is 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, our tools could be extended to support emulation of more aspects. In Figure 4.4, an emulated device can be seen.

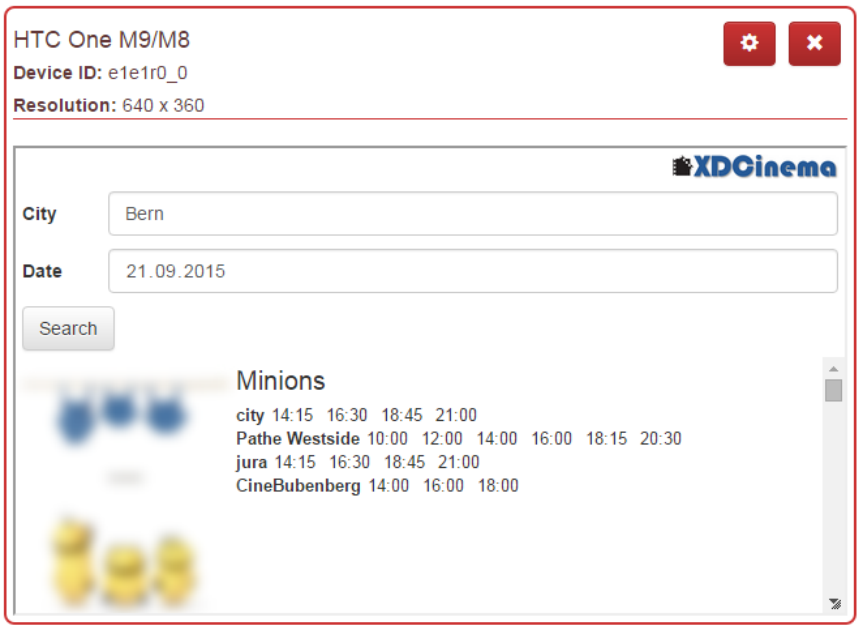

Figure 4.4: An emulated device

Apart from its unique ID, each device also has a unique color. The border of an 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, i.e. the z-index.
- The iframe of 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 it in a dropdown menu.

The device's settings are not constantly used by the developer and showing them at all times would occupy valuable screen space. Thus, the setting menu can be extended and collapsed by the developer by clicking a button. 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 takes up. 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 in the bottom-right corner of the iframe and drag to increase or decrease the resolution. An example of a settings menu can be seen in Figure 4.5.

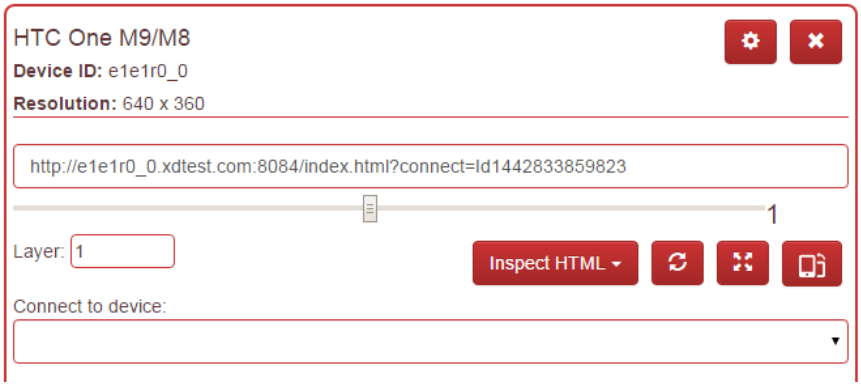

Figure 4.5: Settings menu of an emulated device

In principle, the developer can add as many emulated devices as desired. The only limiting factor is the available screen space, but scaling can be used to solve this issue. The devices can be moved by the developer as desired. Using HTML5's Drag and Drop, our tools allows the developer to click on the header of the device and then drag it to the desired location. Because we require the developer to click on a specific area of the device, we cannot enable dragging by default on the devices. When the developer clicks on the device, we first check if they clicked on the header of the device and enable dragging the device if so. 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 drops the device at the target location, we cannot simply set the position of the device to those coordinates because otherwise the re-positioning would not be accurate enough. For accurate positioning, we also need to consider where exactly on the header the developer clicked, e.g. if the developer clicked 20 pixels to the left and 10 pixels to the bottom of the header, we need to subtract those 10 and 20 pixels from the position where the device is dropped. Otherwise, the device would be shifted by 10/20 pixels. Thus, we also set the offset of the click on the device header as data to the event.

4.4.1 Color Generation

As described above, each device has its own unique color. We use the HSL[13] color space for determining device colors because finding distinct colors is a much simpler task compared to the RGB model. In the HSL color space, colors consist of three values: Hue, saturation and value. The value can also be described as the brightness. The colors of the devices should all have the same saturation and brightness, thus the only value that needs to be determined is the hue. The hue can have any value between 0 and 360. Our goal is to have colors that are easily distinguishable, thus their hues should be as different as possible. A simple algorithm can be used for determining the color of the next device:

[13] https://en.wikipedia.org/wiki/HSL_and_HSV

- If no colors have been assigned yet, assign the color 0/360.
- If only one color has been assigned, assign the color 180.
- If at least two colors have been assigned, compute the maximum distance between two assigned colors and assign the color in between.

4.5 Easy Integration of Real Devices

Real devices can be connected by scanning a QR code or opening the URL. When the URL is opened, the device loads the helper application. Once the helper application is loaded the URL, the application that is being tested is shown in a full-screen iframe on the device. The developer can use the application on the real device, but everything relating to our tools is coordinated through the main application. Therefore, no interface elements are required on the real device. Each connected real device is represented by a proxy within our main application. The proxy of the real device also contains a settings menu, but no iframe. The settings menu is also missing a few things:

- Scaling of the device: There is no need to scale real devices, as no iframe is shown in the main application.
- 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 main application has no access to the HTML of the real device, thus it cannot be inspected.

Figure 4.6 shows the settings menu of a remote device. The settings menu only contains interface elements for setting the URL of the device, the refresh button, the layer input field and the dropdown menu for connecting the device to other devices. If the developer issues any command on the main device, e.g. refreshing the iframe of the real device, the command is first sent to the server and then forwarded to the target device.

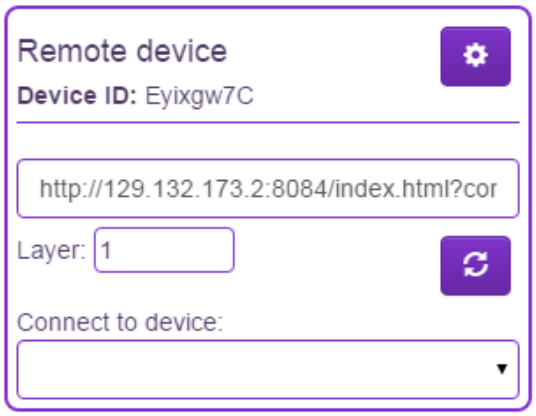

Figure 4.6: Settings menu of a remote device

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.

4.6 Easy Switching of Device Configurations

The developer can save different configurations of emulated devices. The developer can type the name of a device configuration into an input field and then either click a save button to save the current device configuration under that name or click a load button to load a device configuration with that name (if it exists). Autocomplete is used for providing suggestions for existing device configurations that could be loaded (see Figure 4.7).

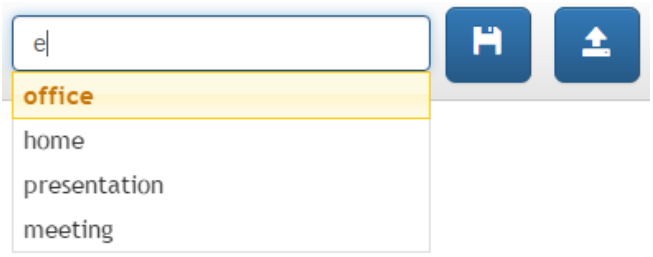

Figure 4.7: Saving/Loading Device Configurations

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 and they need to be re-connected manually anyway, 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.

4.7 Integration with Debugging Tools

Our system includes three different features that are also available in normal browser debugging tools. However, we extended the features to support debugging of multiple devices at the same time.

4.7.1 Shared JavaScript Console

The shared JavaScript console aggregates the console outputs of all devices and allows the developer to send commands to all devices. The following types of logging messages are forwarded:

- console.log
- console.debug
- console.warn
- console.info
- console.error
- console.count
- console.dir
- console.assert

In the script that is injected in the application being tested, those functions are overwritten by a new function and the original functions are stored. The new function first sends the content and type of the logging message to our server and then calls the original logging function. For forwarding JavaScript errors, an event handler is assigned using ”window.onerror” that is called whenever a JavaScript error occurs. The error is then forwarded to our server, together with the stack trace if available.

Depending on the type of message that is sent (error, warning, info, log and all others), a different symbol is used in front of the message being displayed in the console. The message itself is colored in the color of the device that sent it. If a device is deactivated, it does not forward any messages or errors anymore and all existing messages are filtered out. The console can be filtered by the four types of messages mentioned above. All messages that do not have the symbol that represents this type of message are hidden as long as the filtering is active. If the developer types a string to filter by, all messages are checked and if a message does not contain this string, it is hidden.

When a JavaScript error with stack trace is received by the console, the stack trace can be collapsed and extended in the console. The 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. By clicking on an arrow button next to the error message, they can extend the error to show the entire stack trace. Figure 4.8 shows what such a stack trace looks like in our JavaScript console.

Figure 4.8: Stack trace

If the user types a command into the console, the command is sent to all active devices. Inside the script in the application, the JavaScript function ”eval” is called with the command string as argument. The return value of the command is sent back to the server and displayed in the shared JavaScript console.

The screenshot in Figure 4.9 shows the shared JavaScript console including a few examples of how to use it. First, the developer tries to call a function on all devices but makes a typo. Because the function with the typo in the name does not exist, an error message is displayed by all devices. The developer then notices the typo, types the correct function and sees the return values of the function (the coordinates of a cinema). Finally, the developer wants to see which roles have been assigned to the devices. They type the name of the variable that contains the roles and the value of this variable on all devices is shown in the console.

Figure 4.9: Shared JavaScript console

The JavaScript console also has a history function similar to Chrome DevTools. Whenever the user types a command, the command is added to an array containing the command history. By using the arrow keys, the developer can navigate through this array and call previously called command again.

4.7.2 Function Debugging

For function debugging, the Chrome extension is required. The developer has access to an input field where they can type the name of the function they want to debug. When the developer adds a function to debug, a command is sent to each activated emulated device. As soon as a device receives the name of a function to debug, it overwrites the function by another function and stores the original function. The function that overwrites the original function first highlights the device with a semi-transparent green overlay, then calls the original function and stores the return value, then it removes the highlighting from the device and returns the return value. After the function is overwritten, it sends a message back to the server that it is ready for debugging. The server then sends a command to the Chrome ex-

tension with the name of the original function. The Chrome extension receives the command and the URL of the device the message originated from. The Chrome extension then adds the function for debugging using the function name and the URL of the frame that represents the device as parameters.

When the debugged function is now called on one of the devices that the developer wants to debug, the overwriting function is called first and highlights the device. Then, the original function is called and interrupted right at the beginning so the developer can debug the function. After finishing debugging, the function call returns and the device can be unhighlighted. Figure 4.10 shows a screenshot while a function is debugged. In the screenshot, the highlighted device, the DevTools with the debugged function opened and the list of debugged functions can be seen.

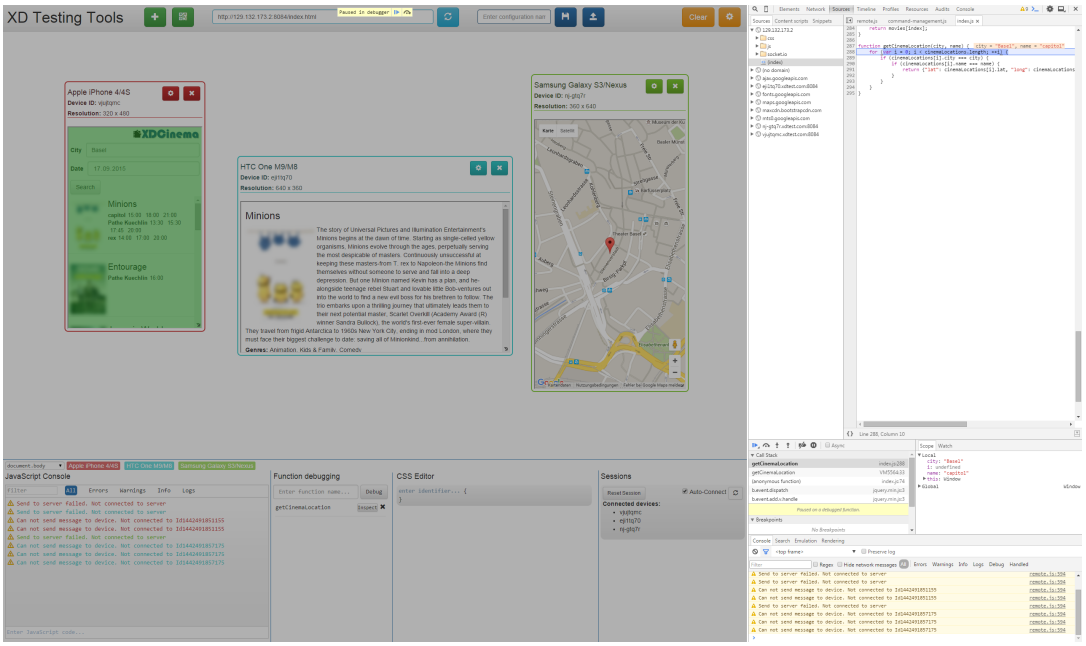

Figure 4.10: The complete interface while debugging a function

If the developer is done debugging a function, they can remove it from the list of debugged functions and the original function is restored inside the script. Additionally, the developer can also just inspect the function without it being called by clicking on a button in the list of debugged functions. Then, the function is opened on a random device.

Apart from function debugging, the Chrome extension is also used for inspecting the HTML of the device. The developer can click on a button in the settings menu of an emulated device to jump directly to the body of the HTML of the device inside the Chrome DevTools. When the developer clicks on the button, a command is sent to the server and then forwarded to the DevTools. The DevTools call the function ”inspect” that is part of the DevTools API with the URL of the frame of the device and ”document.body” as arguments.

4.7.3 Shared CSS Editor

For the CSS editor, a stylesheet is created and added to the document inside the injected script. If the developer adds a new rule to the CSS editor, a command is sent to all devices and the rule is added to the stylesheet. The rules are also stored in an array of CSS rules and the stylesheet is rewritten every time something changes for simplicity. The developer can enable and disable rules. If a rule is disabled, it is removed from the stylesheet; if it is enabled again, it is re-added to the stylesheet.

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 4.11.

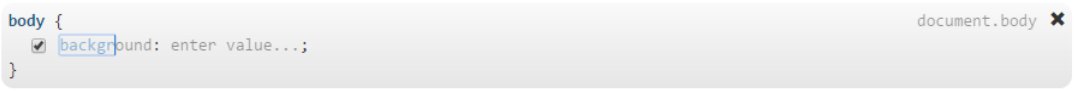

Figure 4.11: Autocomplete in the CSS editor

The available CSS properties are retrieved using ”document.body.style” which contains all available CSS properties, even if they are not set.

Figure 4.12 shows a screenshot of the CSS editor in action. In this screenshot, some CSS rules are added to the body element of all devices.

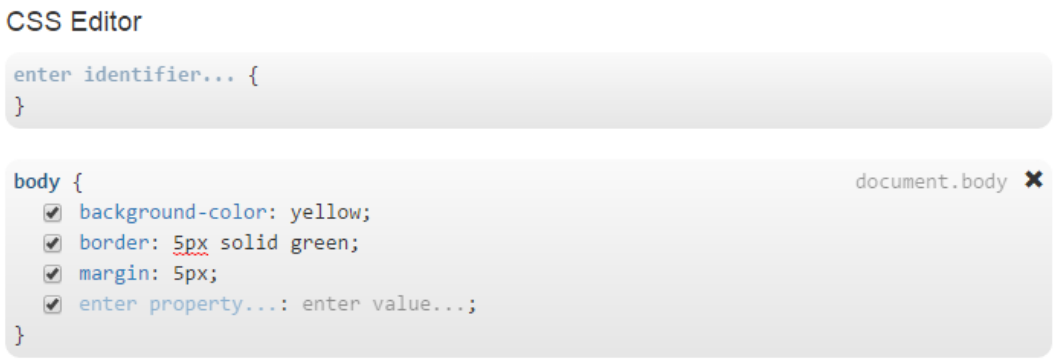

Figure 4.12: Shared CSS Editor

In Figure 4.13, the effects of the CSS shown in the screenshot above are shown.

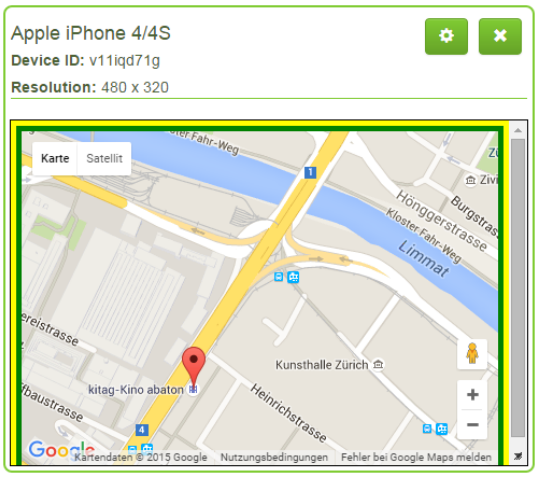

Figure 4.13: CSS applied to emulated device

4.8 Automatic Connection Management

The devices in our system can be connected in two ways. They 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 turned on by default. If auto-connect is on, all newly created and connected devices will 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.

All current sessions are displayed at the bottom of the page of our application. Each session displays all connected devices (see Figure 4.14). The developer can also refresh all devices in a session and reset a session. Resetting a session assigns new IDs to all devices and thus erases the data of the devices.

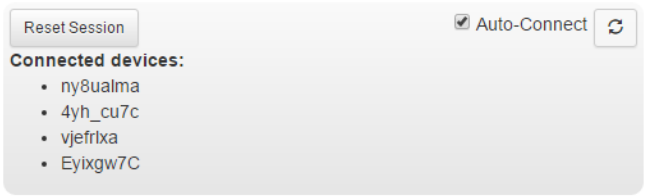

Figure 4.14: A device session

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

is then sent back to the device that wants to connect and can be loaded to connect the device.

4.9 Coordinated Record and Replay

As soon as the script is loaded, event handlers are assigned for all relevant events that we want to record. 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 server and then to the main application. The event handlers are already assigned when loading the script because otherwise other scripts would get a chance to assign event handlers and thus receive events before our script. Furthermore, the event handlers are assigned to the document itself and capturing, thus no other element can receive an event first. This is important because otherwise the page could be modified before we receive the event or the event could even be stopped from propagating. Before sending event sequences 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 devices, 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 reached. From this ID, we can reach the target element by taking the correct 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.

After the main application receives an event sequence, the sequence is visualized. Drag and drop can be used to adjust the timing of the event sequences on the devices. Event sequences can also be moved to other devices using drag and drop. Event sequences can be saved to local storage for later use. Because all events require some space when visualized, events that happen close together in time are grouped together. Event sequences can also be split by clicking between two groupings of events.

Furthermore, the developer can add breakpoints to the replaying timing. By clicking at a certain point in time, a breakpoint is set at that time. When the developer starts replaying, the event sequences assigned to a device are sent to the device along with the list of breakpoints. The events that happen before the first breakpoint are then triggered using ”setTimeout”. If a breakpoint is reached, a message is sent to the main application and the breakpoint that was reached is highlighted. After the developer chooses to continue, a message is sent to all replaying devices telling them to continue event replaying until the next breakpoint is reached. After the last breakpoint has been reached, the event replaying continues until all events have been replayed.

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 replay 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.

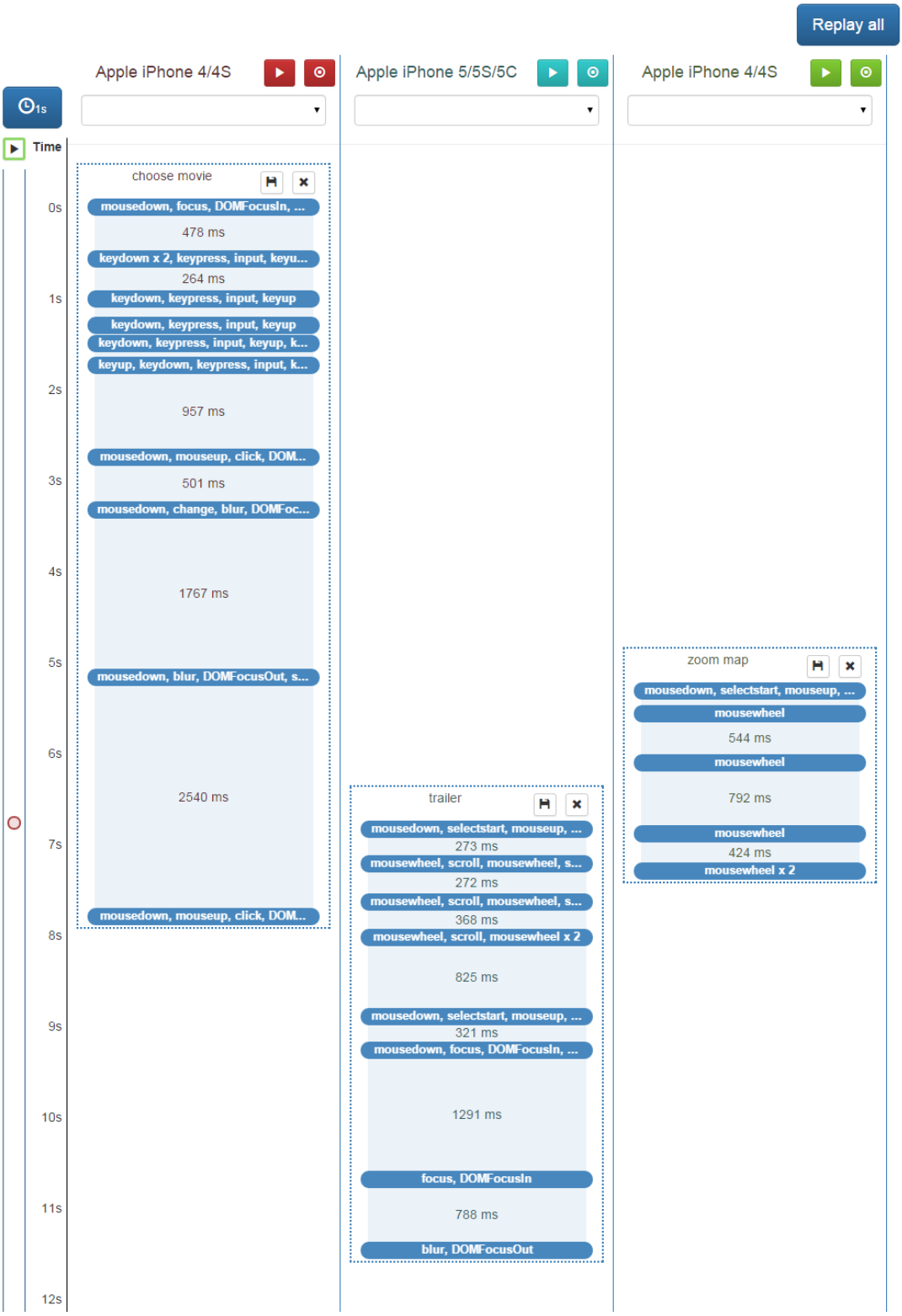

Figure 4.15: Record and Replay

4.10 Integration with Polymer

Shadow DOM event hierarchy Layers

5
Sample Applications

Over the course of our project, we developed two sample applications, XDCinema and XDYouTube. We will now describe those two applications in detail.

5.1 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 5.1 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 movie view (b): 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 location view (c): 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 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.

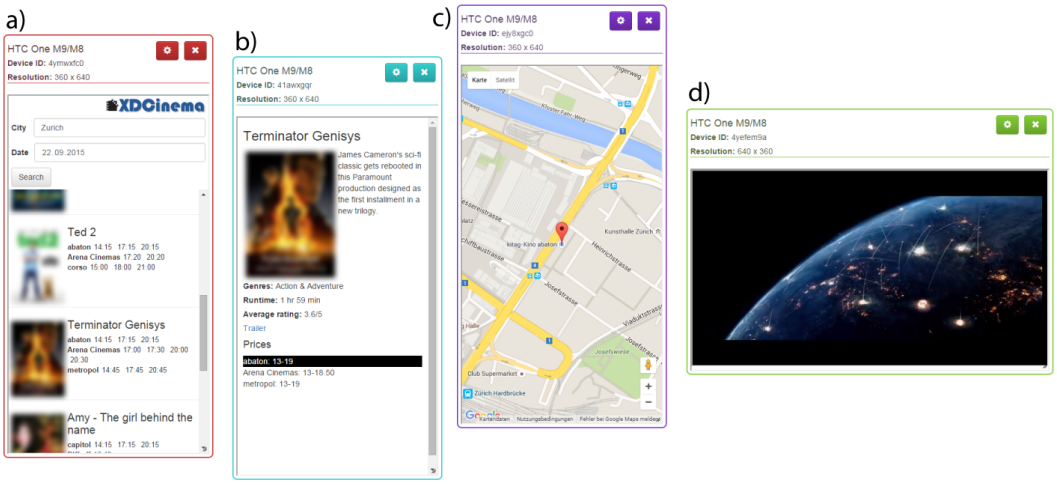

Figure 5.1: Different views of XDCinema

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.

5.1.1 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 distibution happens according to the IDs of the device, but the first device that is connected keeps its search role unless it is disconnected. 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.

5.2 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 5.2), the current video is played.

Figure 5.2: 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 5.3. The second part is only shown in landscape mode (see Figure 5.4).

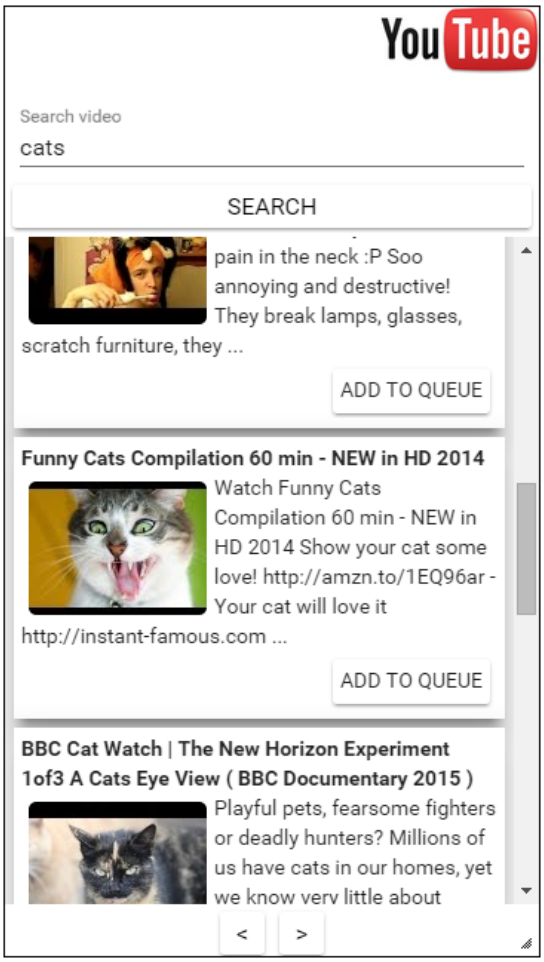

Figure 5.3: First part of the controller view in XDYouTube

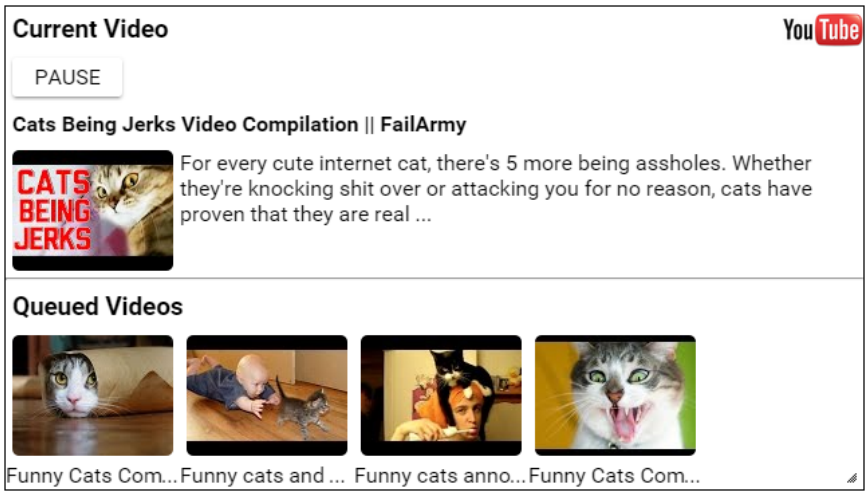

Figure 5.4: Second part of the controller view in XDYouTube

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

5.2.1 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.

Depneding 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.

5.3 Insights

While developing the sample applications, we used our cross-device testing tools. However, the implementation of the tools 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 our tools.

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 the tools. 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.

6
Evaluation

After implementing our tools, we conducted a user study with our tools. The goal of the study was to evaluate the suitability and quality of our tools. Thus, we designed multiple tasks that the participants had to complete with our tools. Because we need something to compare our results to, the participants also completed some tasks without our tools, using only the browser 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 our results.

6.1 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 screen with a 2560x1600 resolution and had access to an English (US) keyboard and a mouse. The desktop PC was running Microsoft Windows 7 with Google Chrome Version 45. They also had access to two real devices: An Asus Nexus 7 (2012 version) Android tablet 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 6.1 shows a picture of the setup of the study.

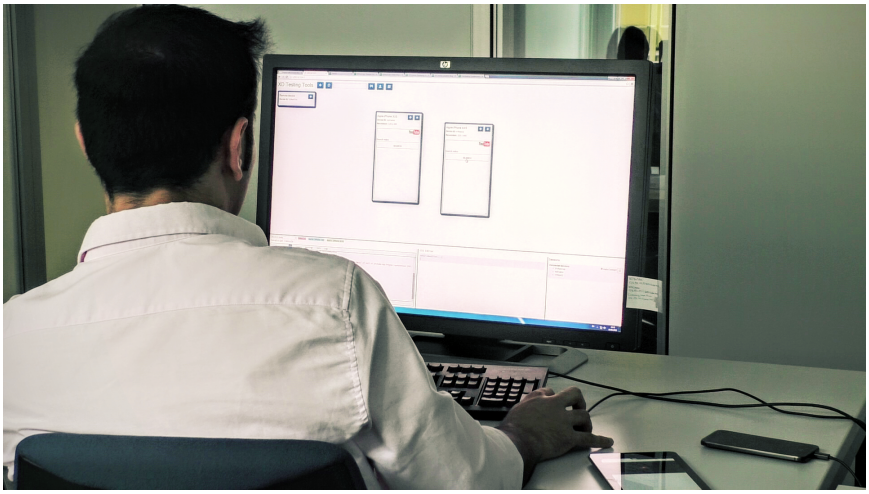

Figure 6.1: Study Setup

During the study, the instructor was sitting next to the participants and was available to answer any questions that occurred during the study. If a participant did not make any progress on a task for some time, the instructor also 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 already 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 our tools 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 our tools and is not needed for completing the tasks in the study.
- Record/Replay: We think that record/replay takes some time to get used to it and it is also only of limited use for simple tasks like the ones 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 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 want to evaluate during the study.

6.1.1 Participants

We recruited 12 participants 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. 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. The age of the participants ranged from 23 to 33 and the median age was 26.

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 applications and cross-device web applications. Furthermore, we asked them about whether they have used Chrome DevTools before and how they used them. The participants rated their skills in web application development and JavaScript on a 5-point Likert scale (see Figure 6.2) from basic to proficient and also gave the numbers of years in experience (see Figure 6.3) that they had in web application development and JavaScript.

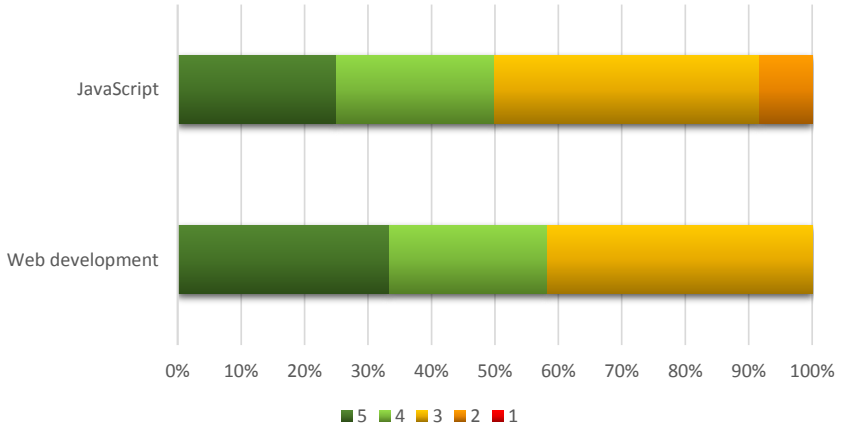

Figure 6.2: Previous experience

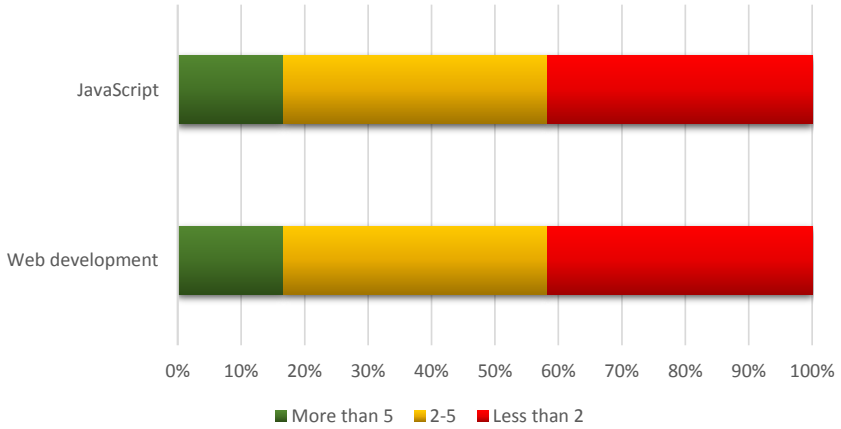

Figure 6.3: Years of experience

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 modes to emulate multiple devices on one device. The fact that the other participants did not use any of those tools indicates that they probably used multiple devices at all times. This could either be because they do not know that they exist, because they are inconvenient or because they 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 developer 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 6.4). Not all participants had used Device Mode, 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.

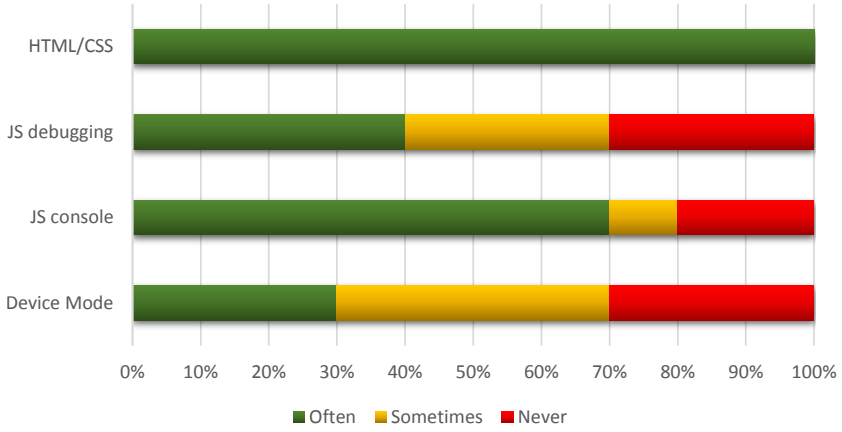

Figure 6.4: Previous experience with Chrome DevTools

6.1.2 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 code, and the other one was about implementing a new feature. The maximum time for the tasks where the participants fixed 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 our tools, the tasks of the other application without them. The order of the applications as well as whether the first two tasks were with or without the tools was random. 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 was

considered easier than the implementation task, and were already familiar with the application for the second task.

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 then 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 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 users 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, 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 was called when the remote control button was clicked (the button as well as the event handler were already implemented), the other was 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 to 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:

- pauseVideo(): Pauses the video
- unpauseVideo(): Unpauses the video
- isVideoPlaying(): Returns true if any video is currently active (even if it is paused)
- setPausedState(state): Sets the state of the shared variable
- getPausedState(): Returns the state of the shared variable (true if paused, false if playing)
- isPlayer(): Returns true if the device is responsible for playing the video and false if not
- isController(): Returns true if the device is responsible for controlling the video (add videos to queue, etc.) and false otherwise

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 was to fix a bug where the location of most cinemas was displayed wrongly, even though the information in the database is correct. The bug was that in one function, ”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, the participants first had to complete the implementation of a function that shows the prices of each cinema 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. The participants had access to the following helper function:

- getCinemaPrice(cinemaName, city): Returns the price range of a cinema in a city

6.1.3 Evaluation Methods

In total, we used four different methods for evaluating the results of our user study. During the study, the participants had to fill out multiple questionnaires. Most our results are based on the questionnaires, the other methods for evaluation are only for clearing any inconsistencies, finding explanations for some things and maybe 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 our tools, we also asked them to rate the usefulness of the individual features of our tools 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 Likert 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 our tools to the usual Chrome browser tools. They had to answer the following questions:

- Did you find it easier to debug with or without the tool?
- Did you feel more efficient debugging with or without the tool?
- Did you prefer debugging with or without the tool?
- Did you find it easier to implement a feature with or without the tool?
- Did you feel more efficient implementing a feature with or without the tool?
- Did you prefer implementing a feature with or without the tool?

For those questions, the participants could either say that they preferred our tool, the normal browser tools, or that they did not have a preference.

In addition, they also answered some general questions about our tool.

- It was easy to learn how to use the tool.
- I felt confident using the tool.
- The tool was unnecessarily complex.
- The tool would be useful for debugging cross-device applications.
- The tool would be useful for implementing cross-device applications.
- I would use the tool for debugging cross-device applications.
- I would use the tool for implementing cross-device applications.

Those questions could again be answered on a 5-level Likert scale from ”Strongly Disagree” to ”Strongly Agree”.

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

Video Recording

In addition to letting participants fill out questionnaires, we also 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 so some strategies for solving tasks could be extracted 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 opinion about the tool. Any comments that the participants had given 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 the tool. 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.

6.2 Results

In the following sections, we will first present the results from the individual tasks and then the more general results. For each task, we will compare how people answered the questions in the per-task questionnaires with and without our tools.

6.2.1 XDCinema: Fixing a Bug

The results for the task where the participants had to fix a bug in XDCinema can be seen in Figure 6.5. The figure shows the median values for the questions asked after the task with and without our tools. 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. The participants felt only slightly more efficient when completing the task with our tools and considered the task as slightly less challenging. In the question about the easiness of the task, exactly the same median value can be observed both with and without our tools. One reason for the bigger difference in the suitability could be that unlike easiness and efficiency, suitability does not really depend on the task itself. In other words, if a task is difficult, the easiness will be rated lower regardless of whether our tools are used or not and if a task is time-consuming, the efficiency will be rated lower. On the other hand, the suitability of the tools for the task does not change with the difficulty or duration of the task. However, it would still be desirable that our tools make debugging cross-device applications easier and more efficient instead of just being more suited. For this task, it seems that our tools mainly

made the task more enjoyable, but did not have any other effects. This could also be due to the fact that the bug is rather trivial and debugging does not necessarily help much anyway.

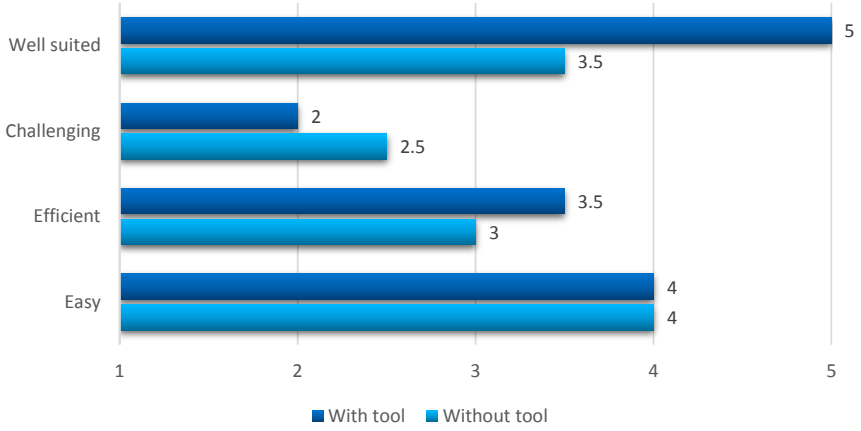

Figure 6.5: XDCinema debugging task - Comparison

Figure 6.6 shows how many participants used the individual features of our tools and how useful they found them. Obviously, the figure only includes the participants that had access to our tools. 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. This could again be due to the fact that the bug they had to fix is actually rather trivial and could maybe even be solved faster by just looking at the code. The connection features and function debugging were used by all except one participant and were very appreciated by the participants. The shared JavaScript console was rather unpopular for this task, probably also due to the simplicity of the task and due to the fact that the bug produced no JavaScript errors that would be displayed in the console.

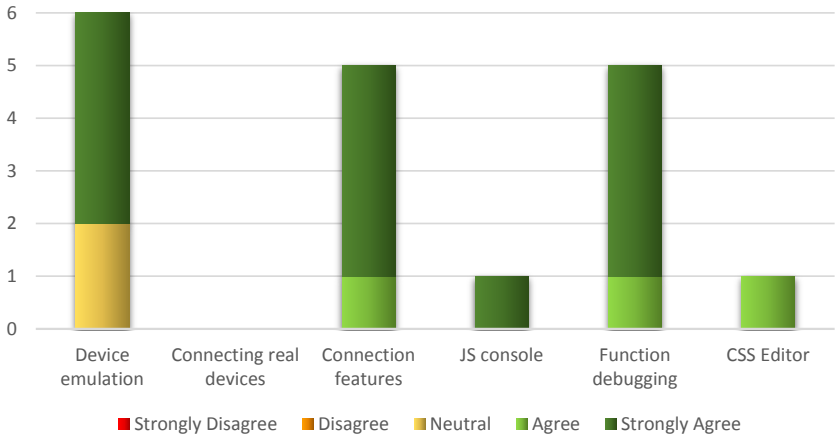

Figure 6.6: XDCinema debugging task - Features used

6.2.2 XDCinema: Implementing a Feature

In Figure 6.7, the results for the implementation task in XDCinema can be seen. Again, the figure shows the median values. 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 the tools perceived it as even easier. The median value of five with our tools suggests that participants had no problems completing the task at all when they had access to our tools. Similar results can be observed concerning efficiency. The difference in median values suggests that the participants felt quite a bit more efficient with our tools. Surprisingly, if we compare the average and median completion times for this task, the participants that had access to our tools were considerably slower (comparing median values, the participants that had access to our tools were about 9 minutes slower). In fact, this is the only task where the difference in completion times with and without our tools 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 as it is random which participants have access to our tools and which not and the number of participants is rather low, it is possible that almost all participants with low experience fall into the same category. 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.

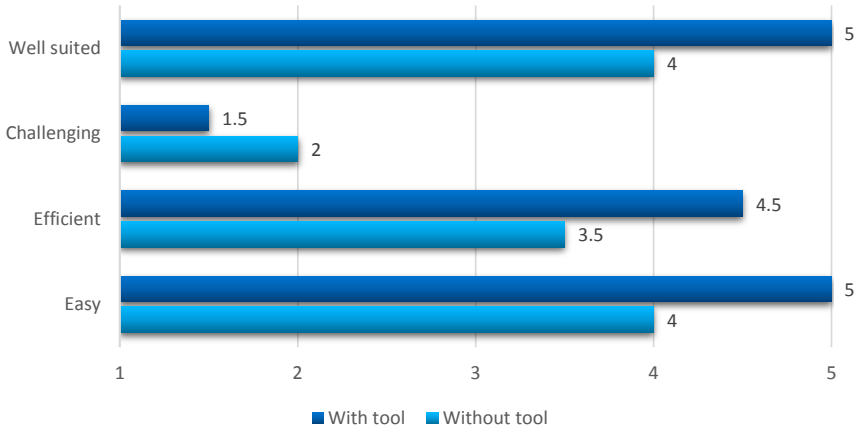

Figure 6.7: XDCinema implementation task - Comparison

Figure 6.8 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. This also explains why the console was used more in the implementation task than in the debugging task: Generally, console outputs are very useful for debugging, but in this specific bug, there were no errors in the console in contrast to the implementation task, where syntax errors were shown in the console. Finally, the CSS editor was also used by some participants in this task. However, some 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.

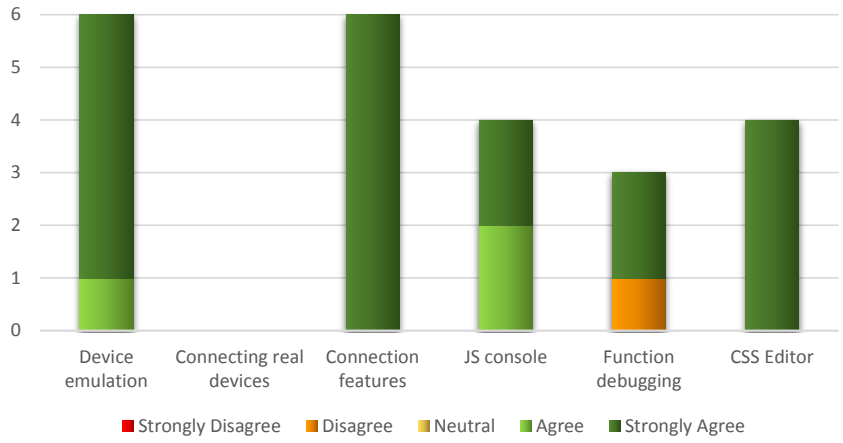

Figure 6.8: XDCinema implementation task - Features used

6.2.3 XDYouTube: Fixing a Bug

Figure 6.9 shows the results for the debugging task in XDYouTube. The difference in suitability between our tools and the usual browser tools was most significant in this task with a difference of 2. The difference in efficiency is also rather large. Surprisingly, the task was rated as almost equally easy and challenging with and without our tools despite the large differences in the other questions. It seems that for this task, our tools did not make the task any easier to solve, but the participants felt more efficient when completing it. During the study, we noticed that most participants had problems reproducing the bug and almost all participants required some hints and finished the task more or less around the time limit. This could explain why the task was perceived as equally difficult with and without our tools: The participants did not really find the bug without help anyway, independent of whether they had access to our tools, so it makes sense that they would consider the task as difficult in general.

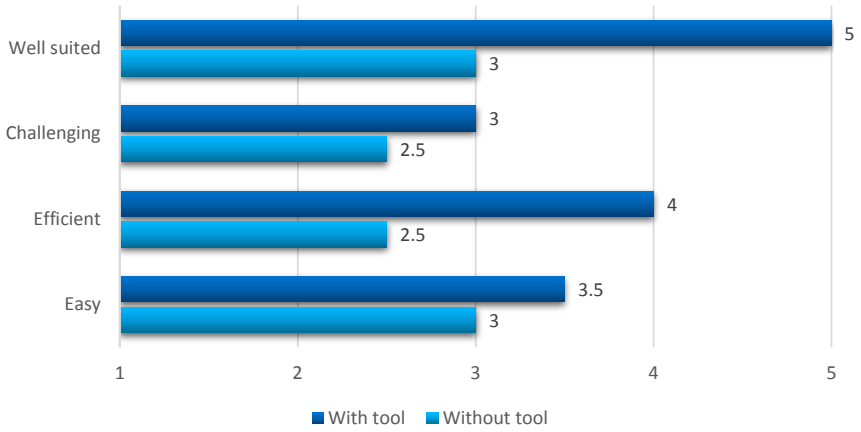

Figure 6.9: XDYouTube debugging task - Comparison

In Figure 6.10; the use and ratings of the individual features can be seen. All of the participants 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 our tools and liked the feature, but no statement about the general usefulness of the feature can be made from just one participant.

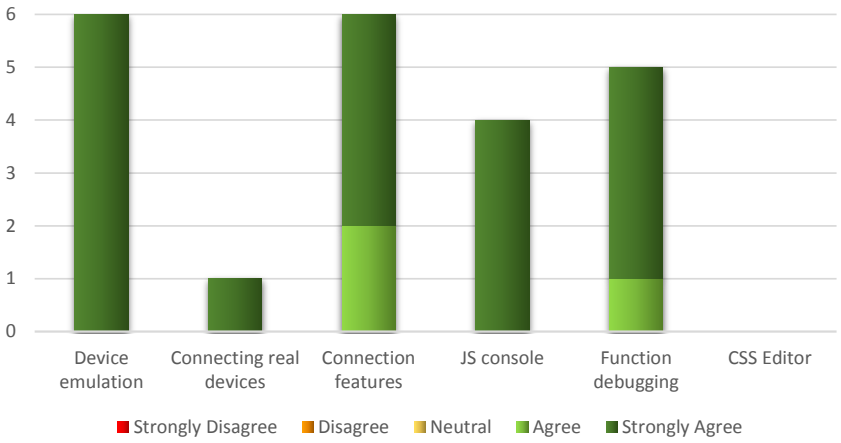

Figure 6.10: XDYouTube debugging task - Features used

6.2.4 XDYouTube: Implementing a Feature

Figure 6.11 shows the results for the implementation task in XDYouTube. In this task, all questions were clearly rated in favor of our tools. While the participant answered the questions in a rather neutral way when they did not have access to our tools, they clearly

stated that the task was easy to complete and felt efficient to complete with our tools. This task differs from the others a bit, because all other tasks had questions where the difference in median was 0.5 or 0, whereas the difference is at least 1 in this task for every question. Especially in the questions concerning how challenging and how easy it was to complete a task, a big difference to the other three tasks can be seen. For the other three tasks, the difference between how challenging the task was was 0.5 for all tasks, and ranged from 0 to 1 regarding how easy it was to complete the task. In this task, the difference was 1.5 for both questions. Thus, this seems to be the only task where our tools made the task much easier to complete. One participant mentioned that they think that the XDYouTube tasks were better suited for the study because they emphasize the cross-device scenario more. If the other participants felt the same way about the task, this could be a possible explanation why they profited the most from our tools in this task.

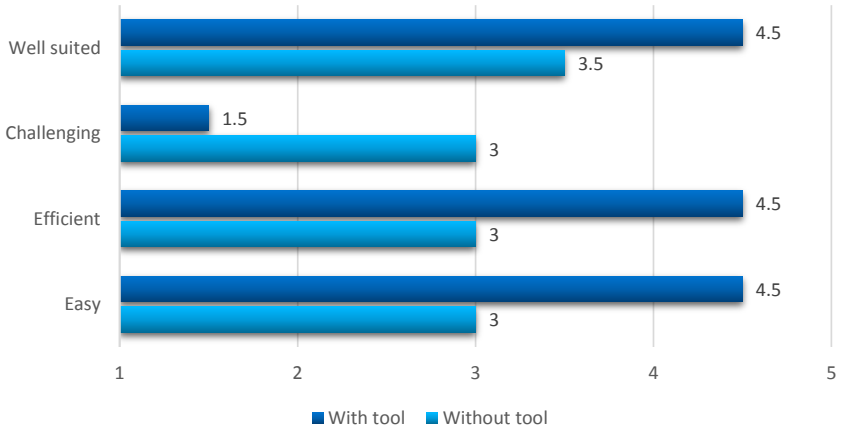

Figure 6.11: XDYouTube implementing task - Comparison

In Figure 6.12; 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 that had a neutral opinion on the console. Function debugging was rarely used for this task. This is rather surprising because many participants had a bug where they had switched playing and pausing the video at the beginning and this could probably have been solved easily by debugging the function. However, most participants just got stuck at the bug and required some hints to fix it instead of debugging their functions.

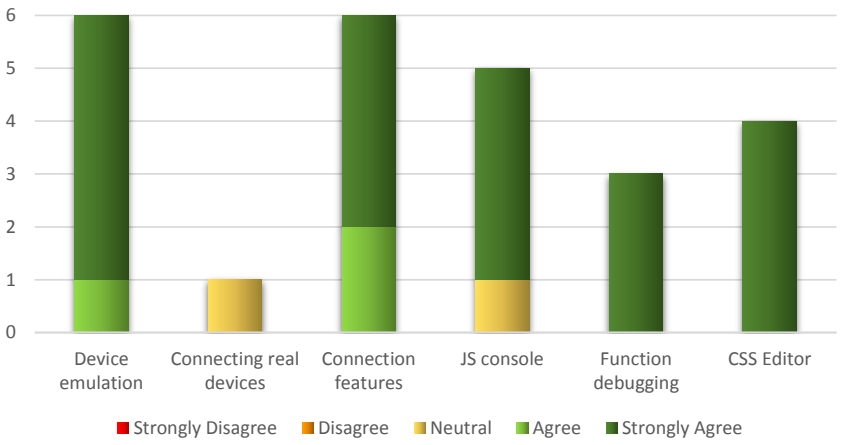

Figure 6.12: XDYouTube implementing task - Features used

6.2.5 General

In general, our tools were rated as very well suited for all tasks with a median value of 5 for each task except the XDYouTube implementation task where the median value is 4.5. In contrast, the usefulness of the usual browser tools was rated with median values between 3 and 4, this indicates that our tools are indeed better suited for implementing and debugging cross-device applications. The efficiency was also rated as significantly better for all tasks except the debugging task in XDCinema, where the difference was only 0.5. This indicates that our tools make the process of testing cross-device applications more efficient. The difference in easiness seems to depend partially on the type of task; in the tasks where the participants had to implement a feature, completing the task with our tools was clearly considered easier, whereas the difference was smaller for the tasks where the participants had to fix a bug. Surprisingly, the results for the question about how challenging it was to complete the task do not necessarily relate to the results of the question about easiness. For all tasks except the XDYouTube implementation task, the differences are very minor, thus our tools do not seem to have much influence on how challenging a task is perceived.

Without our tools, it looks like participant considered the XDYouTube tasks as more difficult than the XDCinema tasks. This generally corresponds to what participants said during the study. While most of them did not make any comments about the difficulty of the task, those who did said that the XDYouTube tasks were more difficult. However, when using our tools, the picture looks a bit different. With access to our tools, it seems that the debugging tasks are considered harder than the implementation tasks. In principle, this is the opposite of what we wanted to achieve, as the debugging tasks were designed as ”introduction” tasks. However, the completion times show that the participants were not actually faster when completing the implementation tasks than when completing the debugging tasks, therefore the different results in the questionnaires can be considered as okay. Surprisingly, the difference in median values for the question where we asked how well suited the tools were for the tasks is larger in the debugging tasks than in the implementation tasks. The difference is 1 for both implementation tasks and 1.5 and 2 for the debugging tasks. Thus, the participants that had access to our tools considered the debugging tasks as more difficult than the implementation

tasks, but still felt that our tools were much better suited than the usual browser tools. So far, those results make sense: If the task is more difficult, the user can profit more from our tools. However, what remains to be explained is why the participants that did not have access to our tools found the XDYouTube tasks more difficult than the XDCinema task. The explanation lies in the XDYouTube implementation task: As mentioned before, participants seem to profit the most of our tools in this tasks. Consequently, our tools make the task so much easier that it is no longer considered one of the more difficult tasks by the participants that have access to our tools. However, the XDCinema bug was not really perceived as easier to complete with our tools, thus it is one of the more challenging tasks for the participants that had access to our tools, but not for the participants that did not have access to our tools.

6.3 Discussion

Figure 6.13 shows that about three quarters of all participants considered implementing a feature easier with our tools. Only one participant found it easier to implement a feature without our tools. However, in principle, this option is redundant as the participants still have access to the browser tools even when they have access to our tools. Thus the relevant conclusion is that about one quarter of the participants did not see any gain in easiness from using our tools. The same applies to the question about whether implementing a feature feels more efficient with our tools (see Figure 6.14); this figure shows exactly the same results as the figure about easiness. However, all except one participant preferred implementing a feature with our tools (see Figure 6.15). Thus, it seems that participants like to have access to our tools even if they cannot directly relate them to a decrease in difficulty or to an increase in efficiency.

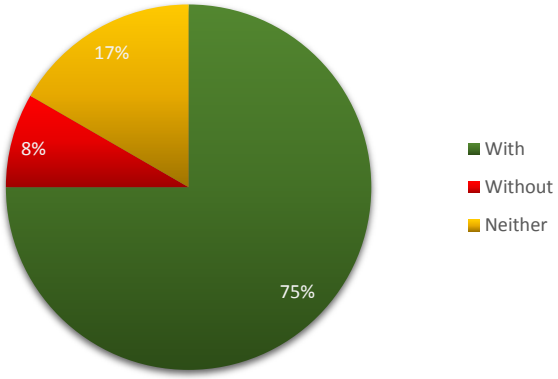

Figure 6.13: Easiness of implementing a feature

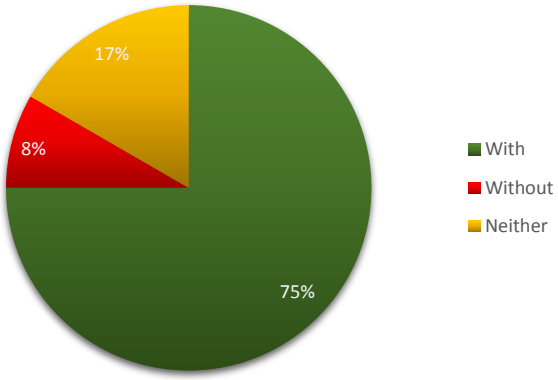

Figure 6.14: Efficiency of implementing a feature

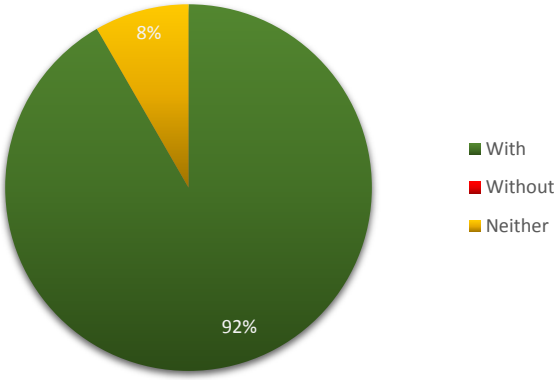

Figure 6.15: Preference for implementing a feature

Figure 6.16 shows that most participants found it easier to debug a cross-device application with our tools. The results get even more obvious if we look at Figure 6.17 and Figure **??**. Those two figures show that all participants felt more efficient when debugging with our tools and also preferred debugging with our tools.

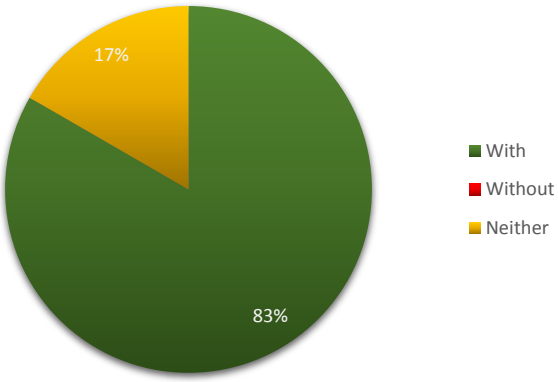

Figure 6.16: Easiness of debugging

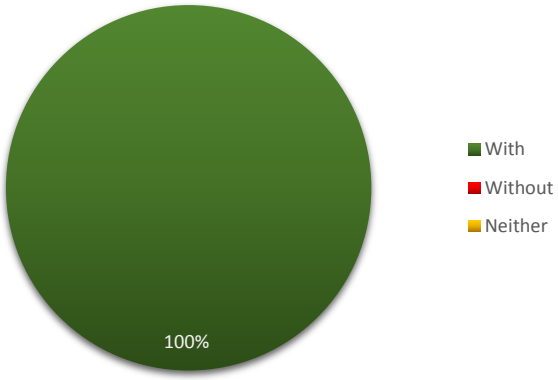

Figure 6.17: Efficiency of debugging

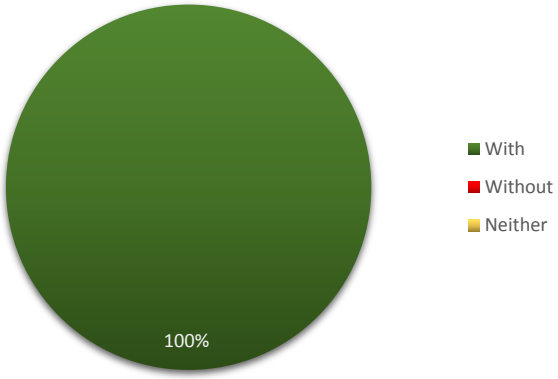

Figure 6.18: Preference for debugging

We also asked the participants if they would use our tools for debugging and implementing cross-device applications and if they think that our tools would be useful for cross-device application testing. The results can be seen in figure 6.19. Almost all participants think that our tools 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 our tools 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 our tools for debugging than for implementing. This is consistent with our previous results, where all participants preferred debugging with our tools but about one quarter did not prefer implementing with our tools.

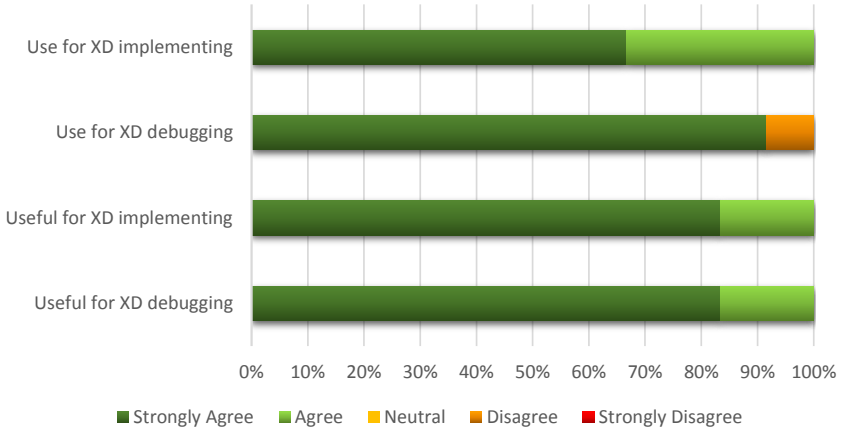

Figure 6.19: Usefulness of our tools

Finally, you can see the results for the general questions about our tools in Figure 6.20. The figure shows that our tools were perceived as easy to learn despite the many different features and the participants also felt rather confident using our tools. 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 our tools are considered easy to learn, it may still take participants some time to get used to them. Our tools are not considered as unnecessarily complex by almost all participants. The results of those three questions indicate that the interface of our tools is generally well-structured and it would be easy for developers to get used to our tools.

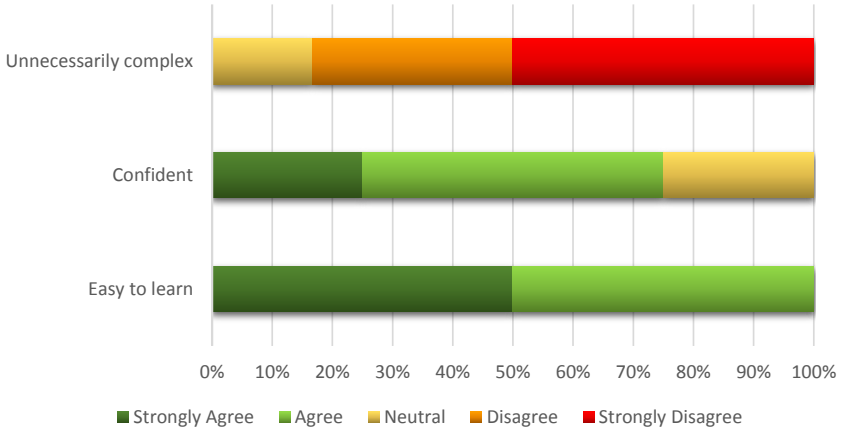

Figure 6.20: General evaluation of our tools

Figure 6.21 shows which features the participants would use for implementing and debugging cross-device applications. In general, people would rather use emulated devices than connecting 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 really 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 complex 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.

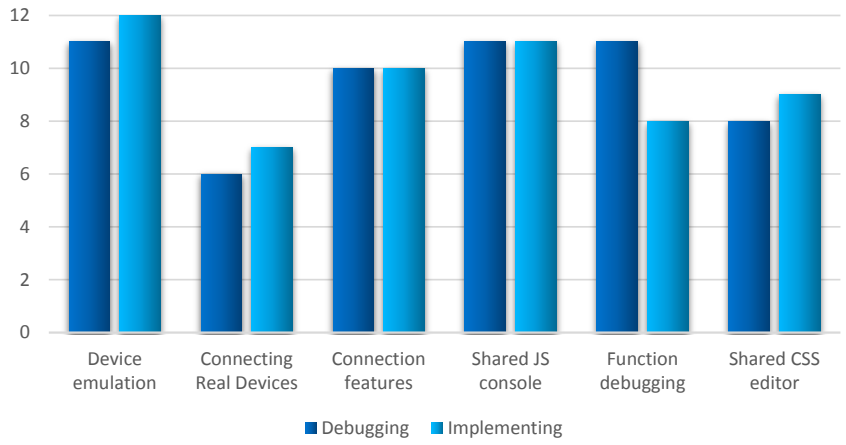

Figure 6.21: Features that the participants would use

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 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, though this is also the case with multiple browser windows. The participants really liked function debugging, the only critique was that it does not show on which device a function is called which was fixed after the study. In general, the output aggregation of the shared JavaScript console was considered more useful than sending commands. One participant even mentioned that he 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 the tool. 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/replay was disabled for the user study, but one participant that had attended a presentation about our tools 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 the task, four while participants had access to our tools, and one where the participant did not have access to our tools. There was one participant that finished all task except the XDCinema debugging task. The participant had access to our tools 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 task successfully. Furthermore, there was one participant for whom we had to abort both implementation tasks (one with our tools and one without) and one participant where we had to abort both XDCinema tasks (both with our tools). 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 our tools says anything relevant about our tools. It was simply bad luck that most of the aborted tasks were carried out in this condition.

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 6.22 shows how many emulated devices the participants used on average with and without our tools. The number of emulated devices used is slightly higher with our tools for all tasks. One reason for this difference could be that it is easier to add a new device with our tools. 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 our tools used more than one device. With our tools, four participants used more than one device. This also indicates that participants prefer to use more devices when

they have access to our tools. 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.

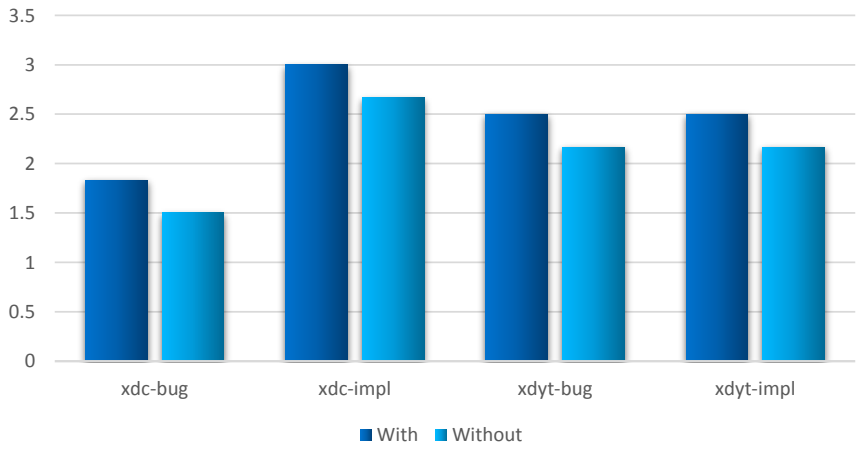

Figure 6.22: Average number of emulated devices used

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 our tools used two real devices, further supporting our thesis that adding emulated devices is more tedious without access to our tools. 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 our tools. 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 our tools, participants also often used smaller devices, but sometimes they had one large full-screen browser window and other smaller browser windows that they just 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 them in landscape mode.

Debugging/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 our tools 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 is 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 in 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 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 because 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 our tools 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 many participants accidentally reload the page instead of the devices and then lose 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 our tools

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 our tools require more research to determine if they are actually desired by developers and would provide some additional benefits.

7
Conclusion

7.1 Future Work

Extended record and replay: Record things like dates, server responses, random numbers, ...

Extended device emulation: Network, location, touch, ...

Tighter integration with debugging tools or browser

Long-term study: Let participants use and evaluate tools over longer period of time

A
Questionnaires

List of Figures

3.1 Difference between resizing and scaling a website 20
3.2 Editing CSS once for all devices 24

4.1 Architecture of our tools 28
4.2 The complete interface 31
4.3 The connected remote device 32
4.4 An emulated device 33
4.5 Settings menu of an emulated device 34
4.6 Settings menu of a remote device 35
4.7 Saving/Loading Device Configurations 36
4.8 Stack trace 37
4.9 Shared JavaScript console 38
4.10 The complete interface while debugging a function 39
4.11 Autocomplete in the CSS editor 40
4.12 Shared CSS Editor 40
4.13 CSS applied to emulated device 41
4.14 A device session 41
4.15 Record and Replay 44

5.1 Different views of XDCinema 49
5.2 Player view of XDYouTube 50
5.3 First part of the controller view in XDYouTube 51
5.4 Second part of the controller view in XDYouTube 52

6.1 Study Setup 56
6.2 Previous experience 58
6.3 Years of experience 58
6.4 Previous experience with Chrome DevTools 59

6.5 XDCinema debugging task - Comparison 64
6.6 XDCinema debugging task - Features used 64
6.7 XDCinema implementation task - Comparison 65
6.8 XDCinema implementation task - Features used 66
6.9 XDYouTube debugging task - Comparison 67
6.10 XDYouTube debugging task - Features used 67
6.11 XDYouTube implementing task - Comparison 68
6.12 XDYouTube implementing task - Features used 69
6.13 Easiness of implementing a feature 70
6.14 Efficiency of implementing a feature 71
6.15 Preference for implementing a feature 71
6.16 Easiness of debugging 71
6.17 Efficiency of debugging 72
6.18 Preference for debugging 72
6.19 Usefulness of our tools 73
6.20 General evaluation of our tools 73
6.21 Features that the participants would use 74
6.22 Average number of emulated devices used 76

List of Tables

Acknowledgements

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.