Is a Framework Enough? Cross-Device Testing and Debugging

Maria Husmann, Nina Heyder, and Moira C. Norrie
Department of Computer Science, ETH Zurich
{husmann,norrie}@inf.ethz.ch, heydern@student.ethz.ch

ABSTRACT

Although numerous cross-device frameworks have been proposed in recent years, we know of only a few cross-device applications that have been implemented and many of these are prototypes to showcase the frameworks. Fewer still have been deployed and are accessible to the public. To go beyond prototyping, applications need to be carefully tested and bugs eliminated but current cross-device frameworks provide little support for this and it therefore remains a challenge. Tied to the lack of real-world applications, there is also a lack of cross-device developers who could be studied to gather requirements for better support. However, based on principles of responsive design, there are many tools available and widely used in practice to support the development of web applications that can be accessed from diverse devices, including ones to facilitate testing and debugging. Inspired by these tools, we have created *XDTools* – a new integrated set of tools for testing and debugging cross-device applications. A preliminary qualitative evaluation with 12 developers produced promising results.

Author Keywords

multi-device; distributed user interfaces; debugging

ACM Classification Keywords

H.5.2. Information Interfaces and Presentation : User Interfaces - Graphical user interfaces

INTRODUCTION

The proliferation of consumer devices such as tablets and smartphones has sparked the creation of a number of cross-device frameworks [15, 5, 8, 22, 25]. The goal of such frameworks is to facilitate the development of applications that make use of, and adapt to, the set of devices at hand. Santosa et al. have observed in a field study [24] that users already avail themselves of multiple devices in their workflows and that better functional coordination between devices is needed. Despite the available frameworks and the identified user needs, we have seen few cross-device applications in the wild.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EICS'16, June 21–24, 2016, Brussels, Belgium
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4322-0/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2933242.2933249

The diversity of devices in terms of screen size, input capabilities and connectivity is also a challenge when developing single-device applications. Developers have been faced with the issue of creating applications that are functional and appealing on all devices. In particular for web applications, responsive designs that adapt to device characteristics (mainly the screen size) are now prevalent and considered good design practice (e.g. [6]). In addition to responsive design frameworks such as Bootstrap[1], a large number of tools have emerged that support the development process. Developers typically work in an iterative process where implementing, testing and debugging are done in turn [13]. Some tools are directly built into modern browsers, some can be downloaded and installed locally, while others can be used as a service online. Two main approaches can be observed: First, emulating a range of devices on a desktop computer and, second, using a varied set of actual devices.

The abundance of tools to support responsive design is an indication that providing frameworks alone is not sufficient to support the development of applications in practice. Supplementing frameworks with tools for testing and debugging facilitates the development process and, in turn, could even spur the creation of quality applications. We decided to investigate what tools could be provided for cross-device application development, with the focus on web-based applications. There are few professional cross-device developers who could be interviewed or observed at work to gather requirements for such tools. Instead, we based our investigations on related work in cross-device development, tools for responsive web design, and our own experience in developing cross-device applications. Advances in web technologies such as Device APIs and WebRTC have laid the foundation for a new generation of web-based cross-device frameworks [27, 1, 12, 19]. Applications built with these frameworks typically run on any device - granted that a modern web browser is available - and require no installation. As these applications are regular web applications, responsive design tools can be used in their development.

While tools for responsive design are a good starting point, they do not cater for cross-device scenarios where multiple devices are used simultaneously and in a coordinated manner. They can be used to compare how an application will look on a tablet or a smartphone, but they have not been built to contrast how it will look on a tablet *and* a smartphone as opposed to two smartphones. If multiple devices are used simultaneously, potentially by multiple users, simply mirroring

[1] http://getbootstrap.com/

interactions from one device to all other devices does not reflect actual usage accurately.

We have also observed the following strategy to emulate multiple devices. Multiple browser windows are opened and, for each window, a different user profile is used to avoid shared data. Alternatively, private windows are used. This is tedious and does not scale well. If a new device is emulated and the number of open windows already matches the number of profiles, a new profile has to be created first. Another issue that is not addressed by responsive design tools is the pairing of devices. Each time an application is reloaded, connections are lost and usually have to be re-established manually which becomes increasingly cumbersome as the number of devices is increased.

We address these issues with XDTools, an integrated set of tools tailored for web-based cross-device development. We extend and adapt concepts used for testing and debugging responsive web applications. Specifically, we support both the emulation of devices and the integration of real devices (Figure 1) and make it easy to switch between different device configurations. Furthermore, we provide automatic connection management to remove tedious re-pairing steps on each iteration. Common debugging tools such as a console have been either rebuilt or integrated. XDTools also includes a record and replay facility for user interaction to automate the testing process and simulate multiple users.

XDTools itself is built with web technologies and is mostly browser-agnostic. However, to benefit from the integration with debugging tools, Chrome has to be used. XDTools is also not tied to a specific cross-device framework, and can be used with any web-based cross-device framework. Small configurations may be required only for the automatic connection management.

In the next section, we discuss the background in cross-device development and web application testing and debugging. After examining developer tasks and the requirements for cross-device testing and debugging tools, we provide an overview of our XDTools prototype before presenting details of the architecture and implementation. Finally, we report on an initial evaluation of XDTools with developers and discuss the insights gained.

BACKGROUND

We first analyse tools for testing and debugging web applications with a focus on responsive design, before discussing related work in cross-device frameworks.

Testing and Debugging Web Applications

Applications are normally developed on either a desktop or laptop computer as a keyboard and a certain screen size are crucial for working efficiently. However, when it comes to testing and debugging applications, it is important to also take into account the devices on which they will be used. Different form factors and input modalities influence the user experience and possibly also the program behaviour. For example, a layout that looks good on a large desktop screen may break on the smaller screen of a smartphone. On the other

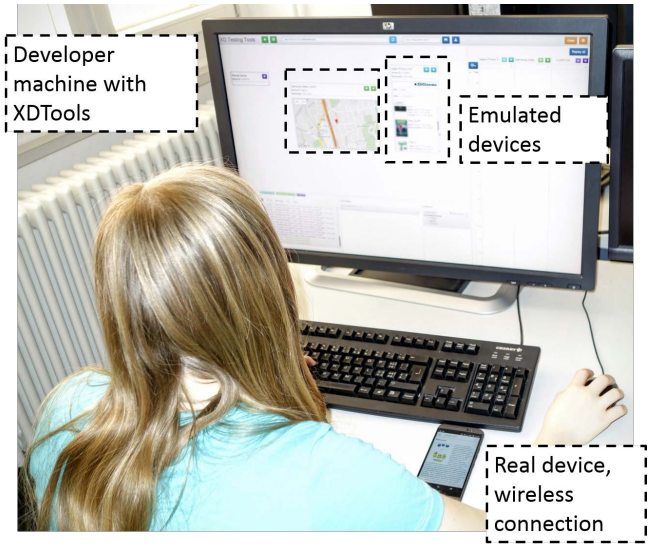

Figure 1. A developer using XDTools. Two devices are emulated in the main XDTools application on the developer screen. A real device is also connected to the system.

hand, the smartphone supports touch events while most desktop computers do not. A handheld device could also have less powerful hardware and slower network connections with the consequence that performance issues may only manifest themselves on particular devices and not on the powerful developer machine. Furthermore, different browser vendors and versions behave differently and implementations that work in one may fail in another.

These issues, while relevant for cross-device applications, are not new and have been addressed. There is tool support for both connecting real devices and emulating devices. When real devices are used, the limited screen real estate of small devices is not suitable for displaying both the user interface (UI) of the application and debugging tools. Thus, tools have been created that allow devices to be debugged remotely from a developer machine.

The remote debugging route is taken by Rivet [16] which provides a browser-agnostic remote debugging mechanism. Among the first tools for remote debugging was Weinre [18] which is aimed at debugging mobile devices that have no other debugging support. Since then, browsers and platforms have started to incorporate support for remote debugging. Typically, remote devices need to enable debugging and have to be connected via a USB cable (e.g. [7]). Smartphone Test Farm[2] addresses scalability issues with this approach and introduces a device inventory. However, USB connections are still required, which may be limiting when testing interactions in a realistic scenario.

When using multiple actual devices, developers have to ensure that all devices load the latest version of the website

[2] http://openstf.io/index.html

when it is changed. Refreshing every device manually becomes increasingly tedious as the number of devices grows. Tools such as BrowserSync[3] and Remote Preview[4] automate this process. Furthermore, BrowserSync can mirror user interactions such as clicking and scrolling between browsers. Similar functionality is provided by Ghostlab[5] and Adobe Edge Inspect CC[6]. Besides tools that can be installed on a developer's machine, there are web services such as BrowserStack[7] and CrossBrowserTesting[8] for testing websites across a number of devices and platforms. Both include a screenshot generation service that renders a given website on a large number of devices.

Another approach for testing and debugging responsive applications is taken with device emulation. Chrome in particular provides extensive support for emulating devices[9]. Common models of mobiles and tablets can be chosen from a list and custom devices can be configured. In addition, the network can be throttled to emulate conditions that occur outside the office and touch input can be simulated. Only one device can be emulated at a time unless multiple windows or tabs are used.

Selenium[10] is a framework for automating tests of web applications in general. Tests can either be recorded and replayed or scripted. It includes a remote control mechanism (WebDriver) that can be used to send commands to a browser. Selenium Grid supports running tests in parallel on multiple machines. Several web services for testing responsive design build on Selenium (e.g. BrowserStack or SauceLabs[11]).

Capturing and replaying program execution is an established technique for debugging distributed systems and has also found application in web-based systems. Mugshot [17] records nondeterminisms (such as key strokes) on end-user machines and uses them to replay the program execution on a developer machine, thus facilitating the analysis of program failure. FireCrystal [23] also provides a replay mechanism, however, the focus is on helping developers understand the implementation of interactive behaviours. The developer can navigate back and forth on the execution timeline and inspect associated code. Timelapse [3] also provides navigation of recordings and, in addition, integrates with other debugging tools. A user study showed that it was most useful to experts and rather distracted less-skilled developers. All three systems replay one recording of a single device at a time and assume single-device use cases. However, in a multi-user cross-device application, it is possible that certain program states are only reached when multiple users interact simultaneously. Neither system provides adequate support for such scenarios.

In summary, there is extensive support for testing and debugging web applications. However, the state of the art treats devices as being independent of each other. Either devices are used sequentially or completely synchronised in parallel – neither approach fitting cross-device scenarios with coordination among devices.

Cross-device Frameworks and Tools

We start by exploring tool support for building cross-device applications and then go on to describe web-based cross-device frameworks that could benefit from XDTools.

Existing tools for cross-device application development mainly focus on the design and prototyping phase and provide little support for testing. XDStudio [20] is a GUI builder for cross-device applications. The developer can specify distribution profiles via drag and drop of UI elements. The designs can be inspected both on emulated devices and on connected real devices. The designs are interactive, however, there is no specific support for debugging such as access to the console. MultiMasher [10] provides visual tools for creating cross-device mashups from existing websites, but designs can only be tested with real devices connected to the system. The Proximity Toolkit [14] and XDKinect [22] integrate sensors into cross-device applications and enable new interaction modalities. The Proximity Toolkit includes a visual monitoring tool for the tracked entities and environment. XDKinect does not offer any direct tool support, however the associated project Kinect Analysis [21] provides extensive functionality for record and replay of Skeleton data as well as data analysis. Weave [4] is a framework for creating cross-device wearable interaction. The created scripts can be tested on real and emulated devices, however debugging support is limited to a log panel. WatchConnect [9] is a toolkit for prototyping cross-device interactions with smartwatches where support for testing and debugging the prototypes is focused on the machine learning algorithms and the recording of sensor data.

XDSession [19] provides support for testing and debugging applications built on the session concepts that the framework introduces. It provides a capture and replay mechanism for user interactions and changes to the sessions if the framework API is used for the manipulations. A basic device emulation mode is provided that supports the emulation of one device at a time. XDSession supports debugging at a higher level of abstraction, thus it better supports finding problems in interactions rather than source code.

There are a number of web-based cross-device frameworks. DireWolf [12] uses a widget-based approach and focuses on the communication of widgets across devices. Polychrome [1] is a framework tailored for cross-device web visualisation, supporting both the creation of new applications and the extension of legacy applications. XD-MVC[12] is an open-source framework that builds on the MVC architecture. Panelrama [27] is a web-based cross-device framework which supports the automatic distribution of user interface element containers (panels) across devices. The distribution algorithm takes into account device characteristics such as physical size

[3] http://www.browsersync.io/
[4] https://github.com/viljamis/Remote-Preview
[5] http://www.vanamco.com/ghostlab/
[6] http://www.adobe.com/products/edge-inspect.html
[7] https://www.browserstack.com/
[8] http://crossbrowsertesting.com/
[9] https://developer.chrome.com/devtools/docs/device-mode
[10] http://www.seleniumhq.org/
[11] https://saucelabs.com/
[12] https://github.com/mhusm/XD-MVC

or touch capability and panel affinity scores, which are assigned by the developer. At the request of some developers, a tool was built that simulates device configurations and generates previews of the distribution. The tool is not described further and the authors do not mention if the previews are interactive. However, the request for such a tool illustrates the challenge in envisioning the results of automatic distributions for diverse device configurations.

These web-based cross-device frameworks provide little or no support for testing and debugging. As they run in an unmodified browser, they could benefit from our tool support. Even frameworks that require some installation but are based on web technologies could benefit. For example, XDTools could be used with Connichiwa [25], which addresses a drawback of purely web-based systems, namely the need for a network connection. This is done by creating ad-hoc networks and running a webserver on demand. This approach requires the installation of a helper application, however, Connichiwa applications are built with web technologies and also run in a standard browser.

DEVELOPER TASKS

In this section, we describe the tasks that developers carry out when developing cross-device applications and identify challenges specific to cross-device development based on the five basic programming tasks identified by Shneiderman and Mayer in [26]: composition, comprehension, debugging, modification, and learning. We omit learning as it is out of scope and describe no specific modification tasks as modification *"requires skills gained in composition, comprehension, and debugging"* [26]. To give an idea of concrete tasks, we will use a cross-device video player (like the application presented in [27]) as a running example. The video player adapts to the devices at hand and consists of the video stream, playback controls, and a search interface.

Composition

Composition entails understanding the problem, devising and implementing a plan to solve it, and finally checking the solution [26]. The first three steps have been addressed by cross-device design and authoring tools and frameworks introduced in the previous section. We focus on the last step, checking the solution, and distinguish between distribution, functional, visual, and performance checks: Does the program distribute across devices as expected? Does it behave as it should? Does it look as it should? Is it fast enough? While the last three checks are not specific to cross-device applications, they are possibly more challenging when multiple devices are involved.

Distribution Checks

Most cross-device frameworks offer a means for the developer to specify how an application should be distributed across devices, for example declaratively using affinity scores [27] or imperatively by selecting device types [4]. Either way, the developer needs to verify that the distribution is as expected as this may be difficult to predict purely by looking at the code (illustrated by the request for a preview tool in [27]). For example, in the video application, the largest device should always show the video stream, while smaller devices like mobiles show the playback controls.

Functional Checks

Programmers need to verify that the program behaviour is correct *despite* the distribution. In our example, when the play button is pressed, the video should start playing on whichever device has the video stream. This could be the same device, if only a single device is present, or another device.

Visual Checks

The challenge of making a UI look good on a variety of devices is exacerbated in cross-device applications where the possible combinations of devices increase the design space and result in even more configurations that need to be checked. For example, when only a tablet is used, the device shows the video stream as well as the playback controls. When a mobile is connected, the controls are moved there. Thus the design of the tablet UI needs to be checked with and without a mobile connected.

Performance Checks

While we do not address rigorous, quantified performance analysis, a programmer will typically check informally if there are any obvious issues in performance. Running an application on multiple rather than a single device introduces a communication overhead that may impact performance. Furthermore, the diverse nature of the devices encountered in cross-device scenarios needs to be accounted for. Not every device is as powerful as the machine that a developer may be using. If an application supports multiple users, simultaneous interactions on multiple devices may also affect performance.

Comprehension

It is common that a developer has to work with software written by another person, requiring that they have some understanding of the code. This understanding can be acquired by reading the code, but also by stepping through it with a debugger or by examining the program output [13]. In cross-device applications, the distribution adds complexity. As the program adapts to the devices at hand, stepping through the code with a debugger on a single device may not have the same result as when doing the same with multiple devices. At the same time, coordinating debuggers on multiple devices manually is challenging and the process would need to be repeated to examine different device configurations.

Debugging

A developer may encounter a bug in one of the checks described above or receive a bug report. Being able to reproduce the bug is a crucial first step [2]. In a cross-device application, it is possible for a bug to manifest itself only in certain configurations, so reproduction can be challenging if the bug report omits such details. For example, the video player application has a button to toggle play and pause. A bug report could state that the state of the button does not match the state of the video. When the developer checks this using a single device that shows playback controls, they do not observe the bug and the button behaves correctly. Only when they add a second device with controls, do they realise that the state of

the button is inconsistent across multiple devices with playback controls because it is not properly synchronised.

Once the developer manages to reproduce the bug, they typically try to answer why something did or did not happen [11]. To this end, they can step through the program using a debugger and observe the runtime state. In a cross-device application this process involves multiple debuggers and, if it is not obvious which device is affected, breakpoints have to be repeatedly set on each device.

REQUIREMENTS

Based on the analysis of the developer tasks, existing tools for responsive design and general web development as well as existing cross-device frameworks and scenarios, we have identified the following requirements for tool support.

Emulation of Multiple Devices

Device emulation in current browsers has been tailored towards sequential use. For example, test first on an (emulated) smartphone and afterwards on a tablet. However, in a cross-device scenario, multiple devices are used in parallel and it is desirable that they can be emulated simultaneously. While it is possible to emulate multiple devices on a developer machine, browsers have not been built for this scenario. Thus, workarounds such as creating multiple user profiles and opening multiple windows have to be employed. A tool for testing cross-device applications should make it easy to emulate multiple devices at the same time.

Integration of Real Devices

While emulating devices is convenient, it cannot replace testing with real devices. Emulators rarely emulate every single aspect of a device and characteristics such as form factors and haptics cannot be emulated. Performance emulation is typically limited to network throttling. Thus it is crucial that applications are tested at least occasionally on real devices. If the behaviour is not as expected, it is desirable that debugging can be started right away incorporating these devices. While existing tools for testing responsive applications do provide some support, it is generally rather static (e.g. requires connecting USB cables) and not a good fit for rapid changes in device configurations which are common in cross-device scenarios.

We suggest that there is also benefit in using emulated and real devices in combination as is illustrated in the following scenario. We use again the video player as an example of the varied device configurations that cross-device applications typically support. Expected configurations include a single PC, a TV with a PC, and a phone with a TV and a PC. When the phone interface is being tested, using a real phone occasionally ensures that the UI looks and feels good when using touch input. However, a TV may not be readily available and, when the focus is on the phone, an emulated TV could be used in the test.

Switching of Device Configurations

As illustrated by the video streaming application, a developer may want to test and debug an application in various device scenarios. For example, an application might update its UI distribution when a tablet is connected. In the case of the video streaming application, it will move playback controls to the tablet if one is available. The state of the art does not consider such device groupings. Little effort should be required to test and debug various device configurations, and switching between scenarios should be easy and quick.

Integration of Debugging Tools

Modern browsers provide a whole range of tools to support debugging. Source code can be inspected, execution stopped with breakpoints, and program state inspected step by step. JavaScript consoles allow the inspection of logs and the ad-hoc execution of code. With CSS editors, applied style rules can be inspected, disabled, or updated. Such tools are very useful and should be adapted to cope with simultaneous use of multiple devices.

Automatic Connection Management

When an application is run on multiple devices and a change is made to the implementation, each device has to reload the new version. Manually reloading each version is tedious and tools such as BrowserSync automate this process. However, cross-device applications typically also include a pairing step in which devices are connected to each other. Each time the application is reloaded, the connection will be lost. While automatic reloading is helpful, it still leaves the devices unpaired and re-pairing the devices on every refresh is tedious. This issue is specific to cross-device applications and thus not addressed by tools for responsive web applications. It could be alleviated by automatically reconnecting previously paired devices.

Coordinated Record and Replay

Recording and replaying tests allows parts of the testing process to be automated and avoids repetitive work, for example when testing for regressions. A test of a responsive application could, for example, include entering text in a field and then clicking a submit button. All user interactions occur on the same device. The test could then be replayed on multiple devices and the results compared. In a cross-device application, user interactions can comprise multiple devices. Furthermore, some cross-device applications support multiple users interacting simultaneously. It is not practical for a developer to gather several people each time they need to test an application. Thus, record and replay functionality should be tailored towards multiple devices and should support simulating multiple users.

XDTOOLS

Based on the requirements, we have designed and implemented an integrated set of tools, called XDTools, that can be loaded into a browser. Our general goals were to reduce effect of the fragmentation and complexity that is inherent to cross-device applications while building on established concepts and tools used in testing and debugging single-device applications.

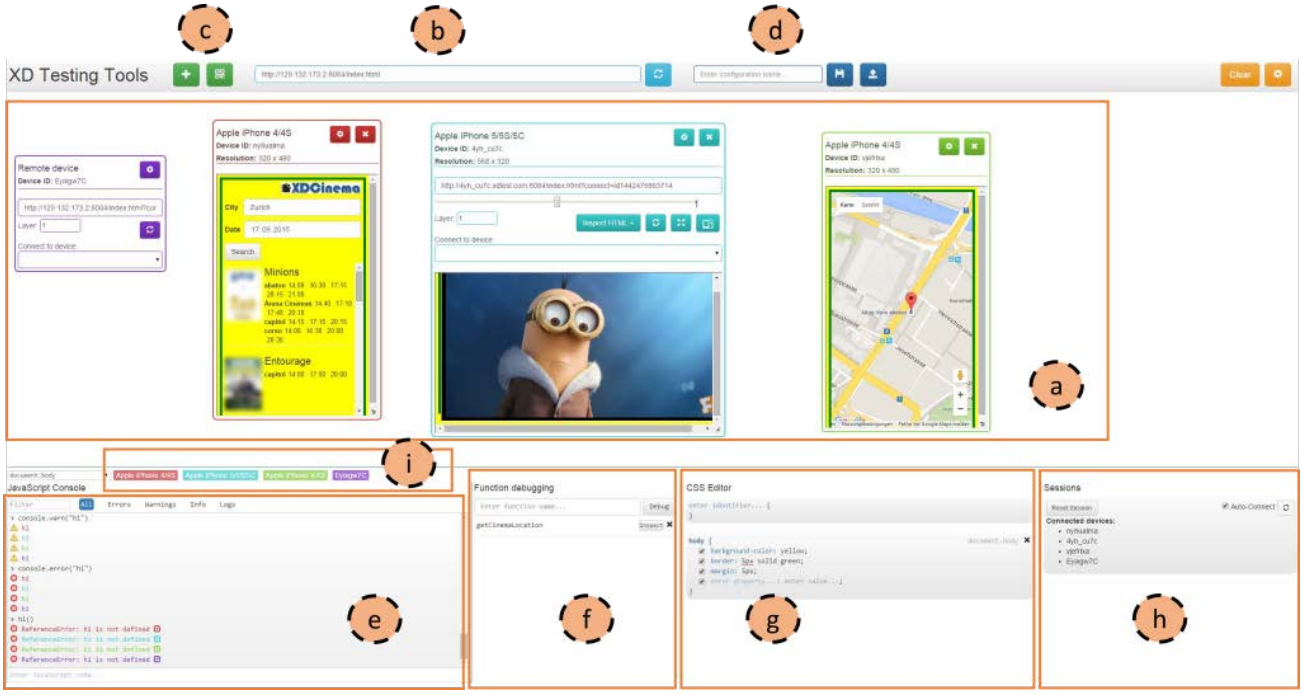

Figure 2. The main application of XDTools with one real device connected and three emulated devices. A cinema application has been loaded into the system.

XDTools consists of a main application that has been built for developer machines with large screens and a helper application for the integration of real devices. Figure 2 provides an overview of the main application. It is centred around a set of emulated devices ⓐ that take up most of the available screen real estate. At the top, a toolbar provides functionality for specifying the application under test ⓑ, adding new emulated devices ⓒ as well as saving and loading device configurations ⓓ. The bottom is reserved for debugging tools, such as the JavaScript console ⓔ, function debugging ⓕ, the CSS editor ⓖ and connection management ⓗ. Each device can be enabled or disabled for debugging ⓘ. Not visible in this screenshot is the record and replay tool which would reside next to the emulated devices on the right side of the screen. The debugging tools can also be configured in size or hidden if not required. In this section, we describe how XDTools addresses each requirement.

Emulation of Multiple Devices

In contrast to device emulation in browsers where only a single device can be emulated per tab, XDTools supports the emulation of multiple devices simultaneously. Thus, at a glance, the developer can see an entire configuration of devices and, when interacting with a device, immediately sees the reaction of the other devices. The devices can be moved and resized by direct manipulation on the screen. Each device is assigned a colour to facilitate coordination and recognition in the debugging tools. A toggled menu (Fig 3) provides additional information about a device and functionality to switch between portrait and landscape modes. The system ensures that each device has its own separate local resources such as

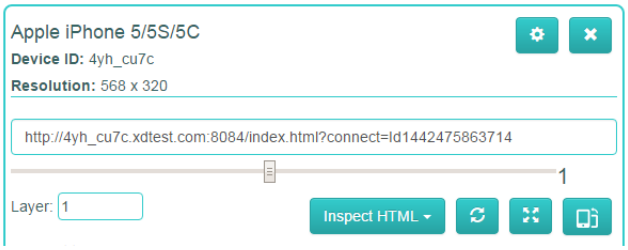

Figure 3. Configuring an emulated device.

local and session storage, unlike the case when multiple tabs are open. It is thus a better imitation of having different devices and allows, for example, multiple user accounts to be logged into an application, which normally requires different browser profiles. Like existing responsive design tools, the system is preconfigured with a list of common device types, but the developer is free to customise them, for example by changing the screen resolution.

Integration of Real Devices

We decided against using cables to integrate real devices. Cables do not scale well and it is cumbersome to connect and disconnect devices physically in order to switch device configurations. Rather, real devices can easily be integrated by either scanning a QR code on the main device or via a URL. This will load a helper application that connects the device to the system. The helper application does not have any UI elements in order not to interfere with the application under

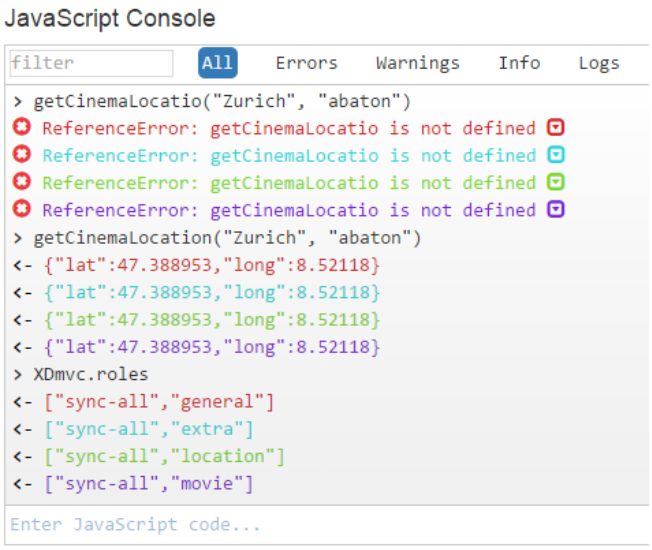

Figure 4. Aggregated consoles.

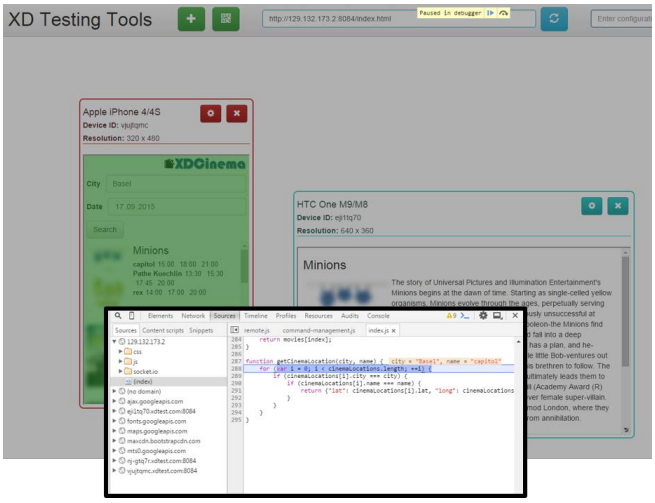

Figure 5. Function debugging. The device hitting the breakpoint is highlighted in green.

test and to provide a realistic testing environment. Devices are represented in the main application with a proxy, making it easy to see at a glance which devices are connected. They can be used just like the emulated devices for debugging, record and replay, and connection management. They are also assigned a colour for easier recognition.

Switching of Device Configurations

With XDTools, different device scenarios can easily be tested. Adding a new device only takes a couple of clicks and commonly used device configurations can be saved and restored. Thus switching between device configurations can be done quickly.

Integration of Debugging Tools

We rebuilt or integrated some of the debugging tools that browsers commonly offer as they have proven to be useful to developers. While in a browser, they can be used for a single device at a time but we offer an integrated solution for multiple devices. By default, all devices are selected and will be available in the tools, however they can easily be unselected to focus on a single device.

JavaScript Console

A JavaScript console aggregates the logs of all devices. Rather than having to open the console on all devices, this provides a single place for the developer to check for output and errors. The logs are colour coded in the device colours to facilitate the identification of the device that is responsible for printing the log. Imitating the behaviour of a typical browser console, our console can also be used to execute JavaScript at runtime on all selected devices, including connected real devices. This could be used to test functionality ad-hoc, inspect the state of an application, or experiment with an implementation. Figure 4 shows example usage. First, a function is called that is not defined and an error message is shown. When the correct function name is used, each device prints the output (a location object) to the console. Here the developer can see that all devices print the same location. The application under test in the example uses different roles for devices and, when the roles are printed, the developer can inspect the roles assigned to each device. Without XDTools, the developer would need to execute the function on each device separately and aggregate the output manually.

CSS Editor

A CSS editor allows new rules to be added to selected devices, facilitating experimentation with new styles without having to reload the application. Again, the developer can select which devices should apply the rules. For example, a developer could easily apply a rule to a tablet device and then create one for a mobile device.

Source Code Inspection and Function Debugging

The source code can be accessed and inspected and it is possible to debug functions (Figure 5) on emulated devices. If any device calls the function that is under inspection, execution will be stopped and the device will be highlighted, making it easy to spot the device that hit the breakpoint. The developer can then step through the function and inspect the current state (e.g. local variables) in the debugger provided by the browser. Without XDTools, the developer would need to set the breakpoint for each device individually and would need to find the correct copy of the source each time.

Automatic Connection Management

In addition to refreshing all devices (real or emulated) simultaneously, XDTools provides an option that will auto-connect previously connected devices. Thus the time consuming repairing step at each reload can be avoided. Furthermore, the system can be configured to automatically pair newly added devices to the existing device set. From our experience, it is a common use case to add a device and pair it immediately with the other devices in the system. At the same time, there

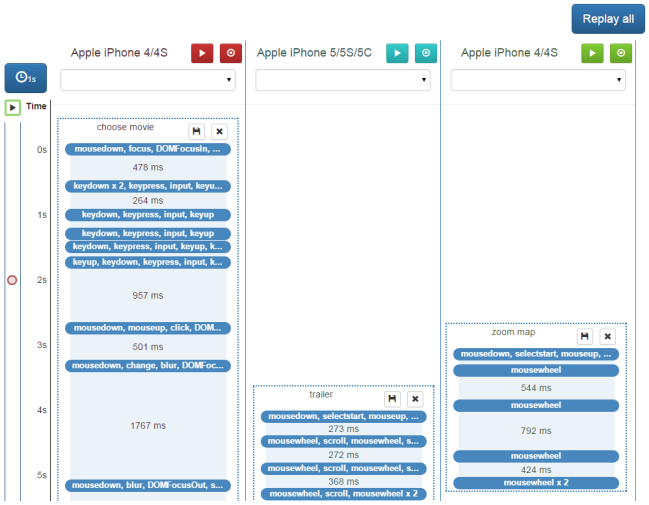

Figure 6. Record and replay.

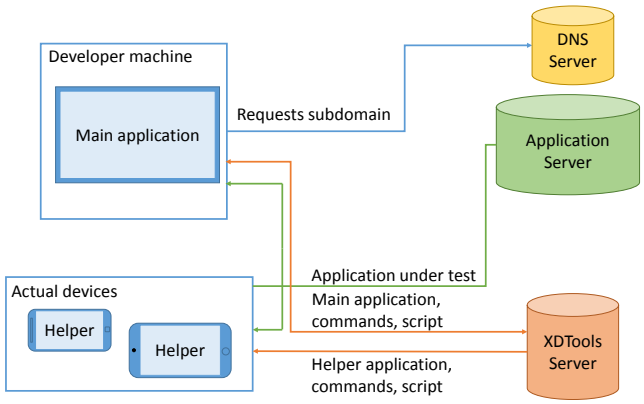

Figure 7. Architecture.

are cases where this is not desired, for example, when the developer wants to explicitly test the pairing step.

Coordinated Record and Replay

XDTools can record user interactions on emulated devices and replay them on both real and emulated devices. Figure 6 shows the replay interface. There is a column for each device that represents a timeline. Recordings can be copied, cut and moved from one device to another, either as a whole or only parts. We thus mirror the characteristic of cross-device applications of distributing UI elements across devices, possibly with replication. A developer could, for example, record an interaction on a single device, then add another device, cut up the interactions to mirror the UI distribution and then replay across both devices to check the functionality.

The replays can be coordinated so that a recording on one device will start when another one has finished on another device. Replays can be started individually on each device or globally for all devices. This flexible replay mechanism allows a developer to simulate multiple users, for example by replaying recordings on multiple devices simultaneously. In addition, breakpoints can be set to stop replaying and inspect the program state.

ARCHITECTURE AND IMPLEMENTATION

The architecture of XDTools is illustrated in Figure 7. It consists of the following parts. There is a main application that runs on the developer device and provides the UI for XDTools. This application is web-based, runs in any browser and is delivered by a server that also coordinates communication with the actual devices. The actual devices load a small helper application. The helper application coordinates the reloading of new versions, the replay of recorded actions and the automatic connection to other devices. Communication between actual devices and the main application is routed via the server and implemented in Socket.io[13].

[13] http://socket.io

A DNS server must be installed locally on the developer machine to create different subdomains for the emulated devices. Finally, the applications under test need to be served, however, any web server can be used and XDTools places no restrictions. The applications only need to be injected with a small script that is hosted on the XDTools server. We now go on to describe interesting and challenging aspects of the implementation.

The helper application for connecting real devices and most of the main application are implemented as pure web applications and therefore do not depend on a specific browser vendor. However, for a tight integration with the debugger, extending a browser was inevitable. We chose to implement a Chrome extension that is used for the code inspection and function debugging. This allows XDTools to control the debugger using the remote debugging protocol[14]. Since Chrome extensions cannot be installed on mobile devices, this approach only allows the setting of breakpoints and stepping through the code on emulated devices, and not on real devices.

In contrast, access to the JavaScript console and integration with the CSS editor is supported also for real devices through the injected script. The script overwrites the default logging functions and forwards all logs to the console of the main application where they will be printed to the aggregated console. In addition, it will execute JavaScript commands that are typed into the console in the application under test.

Device emulation is implemented based on an iframe for each emulated device that loads the application. However, when a domain is accessed multiple times inside the same browser, the browser will share session and persistent data (localstorage) across all instances. For example, if a framework persistently stores an ID associated with each device, by default every instance of an application opened in the same browser will have the same ID, even if different windows or tabs are

[14] https://developer.chrome.com/devtools/docs/debugger-protocol

used. We have observed developers using multiple user profiles or private windows to mitigate this issue, however, we aimed for a more scalable solution. As the data is stored per domain, we request a new domain for each emulated device from the DNS server. The new domain will still resolve to the application server, but the browser will treat it as a different domain and will thus keep session and persistent data separate for each emulated device.

For coordinated record and replay, event handlers are assigned for all events that are tracked by XDTools. It is crucial that these handlers are assigned before any other handlers, thus requiring the developer to inject our script at the top of the page, and that event capturing[15] on the *document* node is used rather than the default bubbling. Otherwise other event handlers could receive the event first and modify the state of the application or even stop the propagation of the events. When replaying, devices schedule all events up to the first breakpoint using *setTimeout*. Once the breakpoint is reached, the main application is informed and the next events will be scheduled when the user decides to resume the execution.

The automatic connection management clearly depends on the connection mechanisms used in the cross-device application or the framework with which it was built. We implemented support for connecting via URLs encoding device IDs which is used in multiple cross-device frameworks such as Panelrama [27] and XD-MVC. To cater for different mechanisms, there is an option to disable the URL-based approach and configure a script-based approach. When using the latter, developers have to provide the implementation for two functions (List. 1). XDTools selects one device as the main device to which all other devices will connect. The main device can be changed in the UI. The function *getConnectionParam* will first be called on the main device, giving it a chance to execute any necessary code and to provide a parameter (for example a device identifier) that will be passed on to all other devices in *connectWithParam*. In that second function, these can then execute the necessary steps to connect to the main device. Note that this approach also allows for all devices to join a predefined session or room if such a concept is used.

```
1  function getConnectionParam() {
2    // Return a parameter that is required
3    // to connect to this device.
4    return XDmvc.deviceId;
5  }
6
7  function connectWithParam(param) {
8    // Establish a connection to a device.
9    // Device information is given in param.
10   XDmvc.connectTo(param);
11 }
```

Listing 1. Configuring automatic connection management.

PRELIMINARY EVALUATION

The lack of experienced cross-device developers limits the form of evaluation studies that can be carried out and precluded a long-term study in the wild. Instead, we conducted a preliminary evaluation with 12 developers that we recruited from our department. The main goal was to receive qualitative feedback on XDTools. The developers were given an implementation and a debugging task that encouraged the use of multiple devices. The participants were provided with a 30 inch screen (2560x1600 pixels) and a desktop computer as a main device running XDTools. A Nexus 7 tablet and an HTC M9 Android phone were available for testing

At least basic skills in web programming (JavaScript, HTML and CSS) were required to participate in the study, however, we relied on self-assessment. Nine participants had experience in developing responsive applications and eight had at least some experience developing cross-device applications. Among those eight participants, five reported using browser tools for emulating devices and six using real devices for testing. Other strategies mentioned were using multiple browsers, multiple browser profiles, and private windows (to avoid shared data).

In general, we received some enthusiastic feedback. All participants agreed or strongly agreed that they would use XDTools for implementing cross-device applications and all but one participant strongly agreed that they would use XDTools for debugging. P3 said *"I think it is very useful for cross-device applications and I wish I had access to it when I was working on a cross-device application last year."* In particular, the auto-connection option was very well received. Also the integration of the debugging tools was praised. Another participant (P8) stated that *"it was very convenient to use. I especially liked the CSS editor and connection features. It really simplifies the process of cross-device application development when one has everything visible on one screen and does not have to switch to other devices, which might be a bit distracting."*

Participants requested an even tighter integration with current debugging tools. As some of the debugging tools were reimplemented and not all functionality of the debugger was available (e.g. breakpoints in arbitrary locations), some developers missed some of the features that they were used to. For example auto-complete of function names in the CSS editor was requested. Only one user connected a real device in each task. Three participants commented that they like to see everything (including emulated devices) in one window or one screen. On the other hand, one participant requested separate windows for emulated devices which our system currently does not support.

DISCUSSION & LIMITATIONS

The feedback in our study showed that the *emulation of multiple devices* and *automatic connection management* are both important in supporting cross-device testing and debugging. In combination, they support fast iterations in the implementation and testing cycle by removing tedious reloading and pairing steps. This allows a developer to check even small changes in the code without a large overhead which in turn may encourage them to test the code more often and detect design flaws or bugs earlier.

Even though six participants mentioned generally using real devices for testing cross-device applications, only one person

[15] http://www.w3.org/TR/DOM-Level-3-Events/#event-flow

connected a real device when using XDTools. The reason behind this could be the somewhat artificial setting of the study. It was possible to solve the tasks using emulated devices only. However, we still consider *integration of real devices* important. During longer development phases, we assume that developers would test on real devices at least occasionally. For example, even when touch is emulated, it does not accurately convey what it feels like to use an application on a smartphone.

We did not evaluate the save and reload functionality for *switching of device-configurations*. It would have been necessary to add additional tasks or make the existing tasks more complex for participants to really appreciate this feature. For example, we could have asked participants to implement two different designs for smartphone and tablet as opposed to tablet and TV. However, at roughly two hours duration, our study was already quite long and we did not want to increase the time further. For the same reason, *coordinated record and replay* was not evaluated.

The participants' feedback suggests that *integration of debugging tools* is essential and that XDTools would benefit from a tighter integration. Ideally, in the future, browsers would be delivered with the support that XDTools now provides directly built into the developer tools as our participants clearly appreciated an integrated solution. One participant (P4) even suggested an integrated source code editor which would transform the browser from a runtime environment to an integrated development environment (IDE). Chrome[16] has already made some steps into this direction.

The main limitations of XDTools include the reduced functionality of the re-implementations. The CSS editor and JavaScript console have a reduced feature set compared to the browser implementations (e.g. inspection of existing CSS rules and auto-complete). The device emulation only takes into account screen size and does not emulate input modalities such as touch. Network and geographic locations are not emulated. XDTools has so far been used with applications based on XD-MVC and also one cross-device application that was built without using a framework. Although no other frameworks have been tested, no issues should arise with purely web-based frameworks. Setting up XDTools on a developer machine is rather complex and requires installation of a local DNS server and configuration of the operating system in order to emulate devices with separate session data. However, integrating the tool directly into the browser implementation would make the DNS superfluous. XDTools has been optimized for a single large screen on a developer machine. It is not uncommon for developers to work with multiple screens. In the future, XDTools could be extended to better cater for this usage scenario.

In general, the main limitation of our evaluation is the short amount of time that the developers used the tool and the fact that only rather simple tasks were feasible during this period. It would be interesting to explore long-term usage of XDTools and gather feedback from developers who use it outside of the artificial setup of a study. XDTools is currently used internally in our group, but the overwhelmingly positive feedback was encouraging and we have made XDTools available to the public as an open-source project[17].

[16] https://developers.google.com/web/tools/setup/workspace/setup-workflow

CONCLUSION

Let us come back to the question in the title: is a framework enough? Unfortunately many of the cross-device frameworks that have been proposed are not even publicly available to developers. There are a few exceptions that have been published as open-source projects, for example Connichiwa [25] and XD-MVC. We have been involved in the development of several cross-device applications based on XD-MVC, including a bike tour planning application, a maps application and a photo gallery application. Our experiences in developing these applications have opened our eyes to the challenges that remain despite building on a framework. While the framework proved to be essential and abstracted away low-level implementation details, for testing and debugging of the applications we had to resort to tools provided for general web development. Since these tools were built for single device applications, they turned out to be inadequate.

These experiences led us to create XDTools. Inspired by tool support for developing responsive web applications, we aimed at facilitating cross-device application development, focusing on testing and debugging. XDTools provides an integrated set of tools and our preliminary evaluation showed that this is appreciated by developers. We believe that XDTools adds value to the existing web-based cross-device frameworks and hope that it will help advance the field by facilitating the development of cross-device applications.

The enthusiastic response of the developers who took part in our study allows us to confidently answer that no, frameworks are not enough. Is better tool support sufficient to spur the development of cross-device applications? Will people actually use cross-device applications? The future will tell. However, privacy and security are also still largely unaddressed in cross-device research and they could play an important role in user adoption.

Once an application has attracted a solid user base, analysing the actual usage of the application could provide interesting insights and feedback which could then be used to improve the application in a next iteration. We suspect that regular web analysis tools may not be sufficient and have started to investigate how the usage of cross-device applications can be analysed.

ACKNOWLEDGEMENTS

This project was supported by grant No. 150189 of the Swiss National Science Foundation (SNF).

[17] https://github.com/mhusm/XDTools

REFERENCES

1. Sriram Karthik Badam and Niklas Elmqvist. 2014. PolyChrome: A Cross-Device Framework for Collaborative Web Visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces (ITS '14)*. ACM, New York, NY, USA, 109–118. http://doi.acm.org/10.1145/2669485.2669518
2. Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 308–318. DOI: http://dx.doi.org/10.1145/1453101.1453146
3. Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. ACM, New York, NY, USA, 473–484. http://doi.acm.org/10.1145/2501988.2502050
4. Pei-Yu (Peggy) Chi and Yang Li. 2015. Weave: Scripting Cross-Device Wearable Interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 3923–3932. http://doi.acm.org/10.1145/2702123.2702451
5. Luca Frosini and Fabio Paternò. 2014. User Interface Distribution in Multi-Device and Multi-User Environments with Dynamically Migrating Engines. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '14)*. ACM, New York, NY, USA, 55–64. http://doi.acm.org/10.1145/2607023.2607032
6. Google. 2015a. Material Design Guidelines - Adaptive UI. (2015). Retrieved September 8, 2015 from http://www.google.com/design/spec/layout/adaptive-ui.html.
7. Google. 2015b. Remote Debugging on Android with Chrome. (2015). Retrieved September 11, 2015 from https://developers.google.com/web/tools/setup/remote-debugging/remote-debugging.
8. Peter Hamilton and Daniel J. Wigdor. 2014. Conductor: Enabling and Understanding Cross-Device Interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2773–2782. http://doi.acm.org/10.1145/2556288.2557170
9. Steven Houben and Nicolai Marquardt. 2015. WatchConnect: A Toolkit for Prototyping Smartwatch-Centric Cross-Device Applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY, USA, 1247–1256. http://doi.acm.org/10.1145/2702123.2702215
10. Maria Husmann, Michael Nebeling, Stefano Pongelli, and Moira C. Norrie. 2014. MultiMasher: Providing Architectural Support and Visual Tools for Multi-Device Mashups. In *Web Information Systems Engineering (WISE'14)*. Springer International Publishing, 199–214. http://dx.doi.org/10.1007/978-3-319-11746-1_15
11. Andrew J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. ACM, New York, NY, USA, 151–158. DOI: http://dx.doi.org/10.1145/985692.985712
12. Dejan Kovachev, Dominik Renzel, Petru Nicolaescu, and Ralf Klamma. 2013. DireWolf - Distributing and Migrating User Interfaces for Widget-Based Web Applications. In *Web Engineering (ICWE'13)*. Springer Berlin Heidelberg, 99–113. http://dx.doi.org/10.1007/978-3-642-39200-9_10
13. Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 492–501. http://doi.acm.org/10.1145/1134285.1134355
14. Nicolai Marquardt, Robert Diaz-Marino, Sebastian Boring, and Saul Greenberg. 2011. The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 315–326. http://doi.acm.org/10.1145/2047196.2047238
15. Jérémie Melchior, Jean Vanderdonckt, and Peter Van Roy. 2011. A Model-Based Approach for Distributed User Interfaces. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '11)*. ACM, New York, NY, USA, 11–20. http://doi.acm.org/10.1145/1996461.1996488
16. James Mickens. 2012. Rivet: Browser-agnostic Remote Debugging for Web Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 30–30. http://dl.acm.org/citation.cfm?id=2342821.2342851
17. James W. Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855711.1855722
18. Patrick Mueller. 2011. Weinre. (2011). Retrieved September 11, 2015 from http://people.apache.org/~pmuellr/weinre/docs/latest/Home.html.

19. Michael Nebeling, Maria Husmann, Christoph Zimmerli, Giulio Valente, and Moira C. Norrie. 2015. XDSession: Integrated Development and Testing of Cross-Device Applications. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 22–27. http://doi.acm.org/10.1145/2774225.2775075

20. Michael Nebeling, Theano Mintsi, Maria Husmann, and Moira C. Norrie. 2014. Interactive Development of Cross-Device User Interfaces. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2793–2802. http://doi.acm.org/10.1145/2556288.2556980

21. Michael Nebeling, David Ott, and Moira C. Norrie. 2015. Kinect Analysis: A System for Recording, Analysing and Sharing Multimodal Interaction Elicitation Studies. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 142–151. http://doi.acm.org/10.1145/2774225.2774846

22. Michael Nebeling, Elena Teunissen, Maria Husmann, and Moira C. Norrie. 2014. XDKinect: Development Framework for Cross-Device Interaction using Kinect. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '14)*. ACM, New York, NY, USA, 65–74. http://doi.acm.org/10.1145/2607023.2607024

23. Stephen Oney and Brad A. Myers. 2009. FireCrystal: Understanding interactive behaviors in dynamic web pages. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*. IEEE, 105–108. http://dx.doi.org/10.1109/VLHCC.2009.5295287

24. Stephanie Santosa and Daniel Wigdor. 2013. A Field Study of Multi-Device Workflows in Distributed Workspaces. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '13)*. ACM, New York, NY, USA, 63–72. http://doi.acm.org/10.1145/2493432.2493476

25. Mario Schreiner, Roman Rädle, Hans-Christian Jetter, and Harald Reiterer. 2015. Connichiwa: A Framework for Cross-Device Web Applications. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '15)*. ACM, New York, NY, USA, 2163–2168. http://doi.acm.org/10.1145/2702613.2732909

26. Ben Shneiderman and Richard E. Mayer. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming* 8, 3 (1979), 219–238. DOI: http://dx.doi.org/10.1007/BF00977789

27. Jishuo Yang and Daniel Wigdor. 2014. Panelrama: Enabling Easy Specification of Cross-Device Web Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2783–2792. http://doi.acm.org/10.1145/2556288.2557199