ASSESSMENT KIOSK

Functional Specifications & Recommendations

Feb 27, 2017

Table of Contents

1. Introduction

This specification intends to define the high-level functionality required for the Secure Browser application. It is intended for the development team and stakeholders of the MIST system.

1.1 Purpose

This specification is designed to cover how the application should behave. It defines key requirements but does not go into the details of how the application should be constructed. This is intended to be a living document and is by no means meant to be exhaustive.

1.2 Scope

This document is focused on the critical features required to administer an assessment from the student interaction standpoint. This application's purpose is to facilitate a secure assessment and is primarily a shell that loads a web-based test. For clear separation of concerns, this application must only support assessment delivery and avoid any details that concern the implementation of an actual test.

1.3 Definitions

These are the key terms discussed in this document:

Lockdown Mode

The ability to control the device so that the student has access only to this application. This is also referred to as being in Kiosk mode.

Test Security

A broad term, but in this document, it refers to the prevention of cheating on a test. This includes preventing access to other applications and protecting the content of a test.

1.4 Design Goals

We want to be forward-thinking and plan for a long-term life for this application. With that in mind, the following are the high-level design goals:

Easy Updates

We want to minimize the deployment costs that are passed onto school administrators. We will strive for critical updates to be rolled out in a seamless and user- friendly manner.

Low Maintenance

We will be supporting a wide range of devices. Ideally, we want to avoid having to deal with a ton of device or OS specific code changes. In addition, adding new devices should be an easy task.

Helpful Diagnostics

We need to be able to diagnose and resolve problems that happen in the field; therefore, having a robust error reporting system in place is critical. If we report errors to the user, we must provide clear feedback and actionable steps to resolve the issue, when possible.

Lightweight

We need to create an application that does not overly burden local resources. This includes CPU, memory, and most importantly, network resources. One of the bigger issues that online assessment platforms struggled with is that they tax the available network resources, thus causing slow and unresponsive test experiences. We must plan for ways to alleviate this stress on the networks and plan to incorporate those solutions into this application.

1.5 Constraints

The following are the known design constraints for this application.

Device Support

We must be able to run on the following platforms:

Desktops

- Windows 7+
- OSX 10.7+
- Ubuntu (LTS) 12+
- Chrome OS 50+
- openSUSE 13.1+
- Fedora

Tablets

- iOS 9.3+
- Android 5+

HTTPS Only

Only HTTPS will be allowed for all network communication.

2. Core Features

The Secure Browser is a multi-platform application that is the primary interface for students taking a test. Its primary goal is to display test content in a secure way. This section lists the critical features that must be made available across all platforms.

ASSESSMENT SECURITY

This application must have the ability to lock the device down so the user is unable to switch between other applications, preventing cheating to a reasonable extent.

BROWSER COMPONENT

The application must contain a modern browser component with support for HTML5, DOM, CSS, and JavaScript (EC6).

NATIVE COMMUNICATION

A means for making native system calls via JavaScript should be made available, with a consistent API that works across platforms.

2.1 Assessment Security

The primary driver for this application is the need for a secure means of delivering an assessment. We want to make sure students are unable to look up answers in other applications, and at the same time, we want to make sure any sensitive test or student data is always kept secure.

Device Security

To minimize cheating during an assessment, this application must:

- Prevent access to other applications
- Display in full-screen mode
- Display on a single monitor only
- Prevent minimizing or maximizing the application window
- Hides all system menus and task bars
- Disable task switching
- Disable all system-level context menus
- Empty the clipboard on application start and exit
- Not start if any predefined blacklisted applications are currently running
- Only allow access to whitelisted URLs
- Disable browser keyboard navigation
- Restrict access to browser development tools

Any known methods for breaking out of lockdown mode should be programmatically prevented. If for some reason this is not feasible, we must document any insecure scenarios so that we can warn test administrators. It would be valuable if we could provide steps that could be taken to manually secure the devices in these cases.

Lockdown must be controllable via the native API, but the application should also allow for an auto lock configuration. If, at any time, the application detects a breach in security, we must notify the testing application via the native API.

Data Security

The application should make the best possible effort to secure all sensitive test and student data by:

- Disabling print-screen
- Preventing printing of any kind
- Encrypting all data at rest
- Deleting all sensitive data on application exit

2.2 Browser Component

The assessment itself is delivered as a web application, so we want to ensure that we have a modern browser that is stable, efficient, and uses mature standards.

Approach

Because we will support a wide range of platforms, it may not be practical to try to dictate a list of required browser features on the application. For example, with iOS we are currently restricted to the built-in browser (Safari, WKWebView).

So, taking the opposite approach may be more practical; we pick the best available browser or browsers and then document the intersection of supported features.

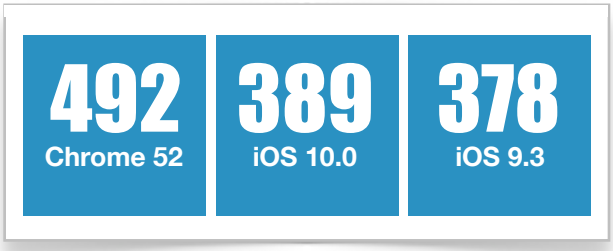
Standards Support

We want to have a modern, but not necessarily bleeding edge browser. We want the implemented standards to be as mature as possible. In terms of standards support, we should focus on HTML, CSS, and JavaScript. We also need DOM, but I think it's safe to say that any browser we would be considering will have adequate DOM support.

TARGET STANDARDS

Analysis

Using HTML5Test, we can compare a set of potential browser candidates. In the case below, we looked at the latest version of Chrome alongside iOS 9.3 and iOS 10. The following scores are out of 555, but what we really care about is the intersection of supported features.



HTML5 COMPATIBILITY SCORES

The full details can be found on the website, but for the most part, these three browsers are very compatible. Obviously, Chrome has implemented more features, but many of those will likely not be needed by the assessment content engine.

There are some minor differences in support for the more text-level semantic and interactive elements, but this should have little impact. A few of the newer form type elements are not yet supported in iOS, like a *color* element. Shadow DOM is not supported in iOS 9.3 and only partially supported in Chrome 52.

We do want to pay attention to the audio and video supports. For video, we are limited to H.264, and for audio, we need to stick with MP3 and AAC. We are also missing support for touch events on iOS, like drag and drop. ECMA6 and CSS3 look pretty well implemented across these browsers.

Generally, I think we need to keep the assessment content team aware of what is and what isn't available across all supported devices, and we need to remember to include all the browsers we intend to support outside of kiosk mode.

2.3 Start-up Checks

The first view that the user will see is a native view that shows a branding graphic a progress bar. This view is designed primarily to give the user feedback while the testing assets are being downloaded, but it also performs a few system checks before allowing the user to continue. If all checks pass, the test view will be displayed. If there is a critical error, the application will give a summary of the issue and is some cases give the user an opportunity to fix the problem and rerun the checks. This section defines the list of things to check for.

2.3.1 Hardware Checks

- Cumulative CPU speed is greater than **233MHz**.
- Total RAM available is greater than **256MB**.
- At least **50MB** of free hard-drive space. (Warning)
- iPads must be 2nd generation or later.

Display

- Can only have one monitor.
- Minimum resolution supported is **1024x768**.
- Minimum color depth is **24-bit**.
- For tablet's we prefer **9.5 inch** displays. (Warning)

Battery

- Device should be plugged in. (Warning)
- If not plugged in, battery level should be **50%** or higher. (Warning)

Operating System

- OS is not running in a virtualized environment.
- macOS: version **10.9** or later.
- Windows: **Windows 7** or later.
- Ubuntu: version **12.04** or later.
- iOS: version **9.3.2** or later.

Network

- Testing server can be reached.
- Minimum download speed is **15Kbps**.

Keyboard

- For tablets, a physical keyboard should be connected. (Warning)

Applications

- Make sure no blacklisted applications are running.

2.4 Native Communication

For all platforms, we need a JavaScript based interface that allows the web-based assessment engine to make system calls on each supported device. The following is a list of required API methods:

The latest version of the native interface can be found at:
https://learninglogistics.github.io/kiosk-diagnostics/nativeapi.html

`kiosk.getDeviceInfo(cb)`

Returns the hardware and operating system data for the device.

`kiosk.lock(callback)`

This call locks down the device to provide a secure testing environment. The steps needed to lock down a device will be specific to the operating system being used, but generally, this method should prevent access to any other applications, disable any possible interruptions, and ensure that the student cannot escape the application.

The success of this call can be determined by passing a callback function that accepts a "response" parameter. The response will contain a JSON object with a boolean (isLocked) to signify the success of the call.

`kiosk.unlock(callback)`

Unlocks the device so the user can easily exit the application. Any content displayed to the user while in an unlocked state should be considered unsecured.

The success of this call can be determined by passing a callback function that accepts a "response" parameter. The response will contain a JSON object with a boolean (isLocked) to signify the success of the call.

`kiosk.isLocked(callback)`

Returns the current kiosk mode state of the device. The state will be passed as a boolean (isLocked) key/value pair, in which true means the device is currently secured.

`kiosk.getVolume(callback)`

Returns the current system volume for the device. Volume will be a decimal value between 0.0 (min) and 1.0 (max).

`kiosk.setVolume(volume,cb)`

Sets the volume for the device. Volume must be between 0.0 and 1.0.

kiosk.clearClipboard(cb)

Clears the system clipboard so no data can be pasted during a test.

kiosk.clearCache(callback)

Clears the browsers cache so all future pages are completely reloaded with content from the server.

kiosk.clearCookies(callback)

Clears any cookies that are currently stored by the browser.

kiosk.saveState(callback)

Attempts to save the current state of the test session by saving both the current URL and any cookies to disk. This is done so that if the application unexpectedly exits, it can be restarted and gracefully return the tester to the same point in the test.

kiosk.restoreState(callback)

Attempts to restore the testers saved state by loading any cookies that have been previously saved and sending the browser to the last saved URL.

kiosk.clearState(callback)

Deletes all state data that has been saved to disk.

kiosk.exit(save,restart,cb)

Closes the application with options to save the testers current state and to automatically restart.

3. Advanced Features

We want to be forward-planning and make sure we do not box ourselves out of features that we may want to include in the future. This section describes some of those features.

Connection Awareness

It would be helpful if the client could have some way of establishing whether or not the test server is responsive and possible actions to take if it is not.

Disaster Recovery

If the server goes down or we lose our network connection, the application should be able to save state and resume gracefully. If the app crashes, we must be able to restore to the last question.

Bundled Content

The browsers should use local caching to ensure that media is not fetched more than once. It would also be great if we could tell the client to fetch content while the application is waiting for the student to answer the current question.

Prefetching Content

The browsers should use local caching to ensure that media is not fetched more than once, but it would also be nice if we could tell the client to fetch content while we are waiting for the student to answer the current question.

Report Issues

Having crash reports and a way for users to report errors would be helpful.

System Readiness

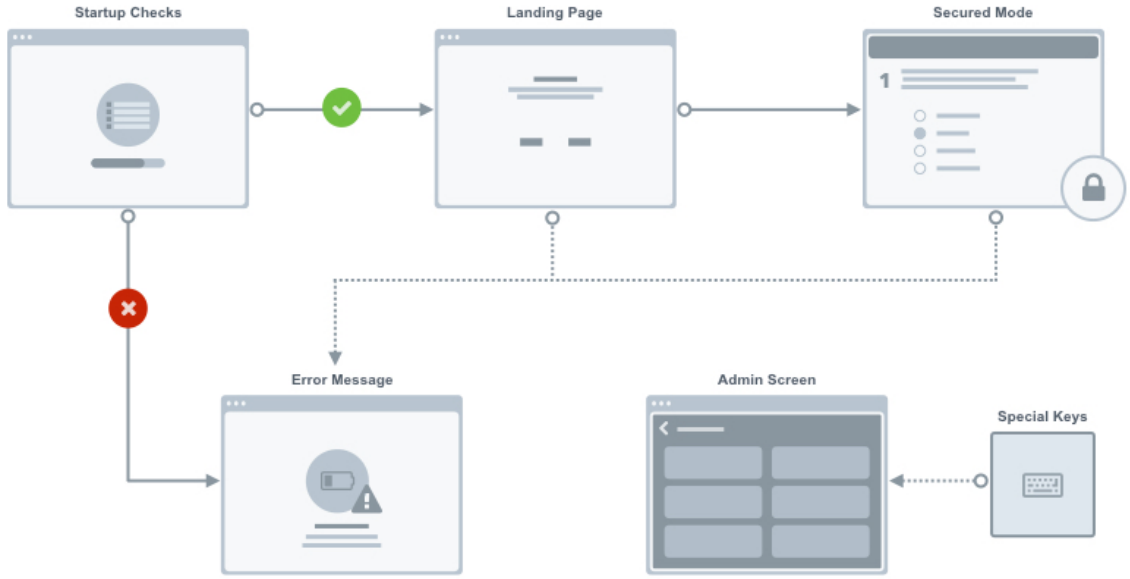
This would be a built-in tool that could be run to test a machine's capabilities and give some indication of expected performance during a live test event.

Custom Splash Screens

I'm not sure if these would be static or dynamic images (HTML local or via URL), but it would be great to be able to define splash screen content at build time.

4. Screen-by-Screen Details

Most of what will be presented to the user will be determined by the assessment content web application itself. This means that the number of screens this application is responsible for is fairly minimal. In this section, we define the expected functionality of each screen and describe the basic design and expected interactions. The exact look and feel will likely change more frequently than this document will change, so it will not be covered.



4.1 Application Flow

The Secure Browser itself is fairly straightforward, with the complexity of delivering a test relegated to other applications. The primary goals of our application are to display a web browser component in full-screen, load the test URL, and respond to any API requests. That being said, we should try our best to complete these actions in a graceful way to give the user a seamless experience. The application should always launch in full-screen mode but should allow the locking down to be controlled via the native API.

4.2 Initialization Screen

The initialization screen is here to give the user immediate feedback while the application starts up. Some basic system checks should occur, and we should display

the error screen if any of the checks fail. An example would be a low battery check for laptops and tablets. The initialization screen should show the testing application logo, the application title, a progress bar, and a status message.

It is highly recommended that the testing application be loaded in the background while the system checks are occurring to give the user the feeling of a fast application launch. On the same front, most of the system checks will take milliseconds to preform and will not be noticeable to the user unless we add some delay. As long as the user is getting fairly constant feedback – meaning the progress continues to move forward and they receive updated status messages – having a several second (about five seconds) system check process would be acceptable.

We also want to gracefully handle any page loading issues that might be happening in the background. If we haven't fetched the full test application within 10-15 seconds, we should notify the user of the issue.

4.3 Error Screen

There will be conditions that will prevent the user from starting a test or will possibly interrupt a test, and we will need to provide clear feedback to the user in these cases. This screen should have a clear message with details for how to remedy the error, if applicable.

4.4 Browser Screen

This is the primary screen that administers the test. It is simply a full-screen browser control that loads the testing web application's starting URL. We must not show any of the browser's chrome, like toolbars or tabs, and we should disable any navigation control via keyboard or mouse clicks.

4.5 Admin Screen

A special screen should be accessible to administrators and proctors that will expose any special functionality that would not be appropriate to display to a student. This screen should appear when a specific, predefined set of keystrokes or screen touches has been detected.