Web-Based E-Assessment Beyond Multiple-Choice: The Application of PHP- and HTML5 Technologies to Different Testing Formats

Documentation

Master's Thesis

in

Linguistics and Web Technology

presented to the Faculty of

Foreign Languages and Cultures

at the Philipps-Universität Marburg

by

Julia Neumann

from Naumburg (Germany)

Marburg, 2015

Contents

List of Abbreviations

AJAX	Asynchronous JavaScript and XML
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hypertext Markup Language
JPEG	Joint Photographic Experts Group
MVC	Model-View-Controller
MySQLi	MySQL Improved
PHP	PHP: Hypertext Preprocessor
PNG	Portable Network Graphics
SQL	Structured Query Language
SVG	Scalable Vector Graphics
XML	Extensible Markup Language

1 Introduction

This documentation provides an overview of an application developed for the creation and management of web-based assessment tasks in three different formats. The application consists of a user-friendly interface to a database structure for storing the created tests and allows its users not only to generate new tests, but also to edit, delete, view, and run existing tests. Thus, it constitutes a tool that can be used by administrators of e-learning platforms, who do not need to have programming skills on their own, to create interactive exercises for their learners. These exercises go beyond the traditional multiple-choice tests commonly used in web-based learning environments: Users of the e-test editor application can build crossword, dynamic multiple-choice, and drag-and-drop tests, thus offering learners a variety of online exercises that can increase motivation and be tailored to the requirements of different topics.

Modern HTML5 technologies in combination with PHP-based server-side components provide the technological foundation that allows for the implementation of such interactive exercises. While most of the general functionality of the e-test editor is implemented on the client-side using JavaScript, the server-side parts mainly take care of the communication of data to and from the database. The following sections will describe in detail the overall organization of the code, as well as the database structure and the specifics of the implementation of each of the three testing formats. Moreover, the implementation of general features of the interface, which aim at making the management of the tests as easy and user-friendly as possible, will be outlined briefly. Before the technological aspects mentioned above are discussed, a brief user guide will be provided to explain the main aspects of the user interface for end-users.

2 User Guide

The application is started by accessing the 'index.php' file in the browser. This page allows the user to choose a format for creating a new test by clicking on one of the big buttons on top, which contain the icons and names of the formats. Hovering over one of these buttons will display a short description of the format in question. Apart from this, in case tests have been stored to the database already, the user can work with these tests. They are displayed in a table, which can be sorted according to the criteria given in the table header by clicking on any such criterion. The gray triangle appearing next to the current order criterion indicates whether the tests are displayed in descending or ascending order, and it can be clicked to reverse their order. By checking or unchecking the checkboxes next to the table, the user can limit the display to one or several testing formats, which might prove useful if one is searching for a specific test in a table containing a great number of tests. By clicking on one of the tests in the table, the buttons below are activated, allowing users to choose whether they want to edit, view, delete, or run the selected test.

To keep the organization of the application, and thus also navigation, as simple as possible, all pages can be accessed via the interface on the index page, as described above. Therefore, the application does not contain a menu as it is part of many traditional web pages. Instead, the header on top of the page will let users know which part of the application they are currently working with, and clicking the 'Home' button on the right will always take them (sometimes after asking for confirmation) back to the index page. Clicking on the author's name in the page footer will open the contact page in a new tab, whereas the small colored squares in the bottom right corner allow users to choose their preferred theme color for the application.

The different pages for working with the tests are designed to be as self-explanatory as possible. Short instructions on how to proceed are given on each page. In general, when creating a new test or editing an existing test, questions

and items can be added via form fields and/or buttons. Elements that have been added to the test can be modified by simply clicking on the text that needs to be changed. This will convert said text to an input field where the changes can be made. Pressing 'Enter' or clicking somewhere else on the page will apply the changes. Elements that can be deleted will display an 'X' on the right-hand side when hovering over them. Clicking on it will ask the user for confirmation before the deletion is carried out. When viewing a test, the user is given the option to change between a view that shows an empty version of the test and a solved view showing the solutions for all the questions in a test. Furthermore, solved and unsolved views of tests can be printed, as well as exported in two different image formats (note that these options might not be available in all browsers). Running a test means being given a fully functional interactive version of the test, as it would appear in an e-learning environment. The 'Check' button, which is available here, becomes active as soon as the user has provided a complete solution (that is, answers to all the questions/items currently displayed on the page) and will display feedback on the user's performance.

While these explanations, along with the instructions given on the different pages, should be sufficient for end-users to work with the e-test editor, the following sections of this documentation will provide more information on the technological background and implementation of the application.

3 Overall Organization of the Code

The main part of the code base of the application consists of collections of interface files for each of the three test formats, supplemented by their respective JavaScript and CSS files. The index page and the contact-form page are also linked with such individual client-side scripts and style sheets. Furthermore, a number of general scripts, as well as a general CSS style sheet, are used by all parts of the application alike. On the server-side, these include, for instance, PHP support files, such as a class declaration file (class_db_connection.php) that can be used for creating objects which allow for an easy interaction with the database. Its implementation is based on the MySQLi extension of PHP [INT 1]. Objects inheriting from this class have methods available that construct and carry out SQL queries for selecting, deleting, inserting, and updating database records. Furthermore, a configuration file (config.php) for setting constants and including scripts needed on all pages, as well as a collection of common functions (general_functions.php) are provided. Lastly, another support file (manage_tests.php) mainly receives the user input from the index page and re-directs to the required parts of the application. It also handles AJAX requests that are the same for all types of tests. Another collection of files used by all pages can be summarized as PHP page elements, containing templates for a page header, a page footer, and a form soliciting general test information (test name and level). These elements are basically the same for a number of different pages, on which the template scripts are included as needed. The general JavaScript file will be discussed in the context of the discussion of the JavaScript design under 3.1.

Aside from these general components, the parts of the application handling the different testing formats are organized as follows: For each format, a 'newtest', an 'edittest', a 'viewtest', and a 'runtest' page are provided, containing the basic skeletons of the interface needed for performing the corresponding interactions with the tests. These files also set some important global variables that are used both in the supporting PHP and JavaScript files to indicate which type of test

('crossword', 'dynmc', 'dragdrop'), as well as which mode ('new', 'edit', 'view', 'run') is currently being handled. The application relies on these variables, as well as on the files being named consistently and using the correct labels for the testing formats, so that the common page header, for example, can link the correct files for each part of the application.

The PHP files for each testing format furthermore include a 'managetests' script, which is mainly responsible for handling AJAX requests made by the JavaScript component. These include saving new and changed tests to the database, as well as retrieving data of existing tests for the edit, view, and run modes. Furthermore, in the run mode, AJAX requests are used to dynamically retrieve questions and evaluate user input. This is transferred from the client- to the server-side, so that learners who are competent in using developer tools will not be able to find solutions to questions in the JavaScript code. Finally, the deletion of a test, which does not require a separate interface page and is instead triggered directly from the index page, is also handled by this managing script.

Apart from the interface pages and the 'managetests' script, there is a 'templates' file for each test format. This file contains templates for HTML elements that are dynamically added to the page with JavaScript. It is included on every interface page, but, as the templates are embedded in 'script' tags with the type 'text/template', they are ignored by the browser, while at the same time, they can be read by JavaScript to generate the respective HTML elements as needed.

As the main functionality of the application can be found in its different JavaScript components, their organization will be discussed in more detail now. JavaScript rather than PHP was chosen as the main language, since some of the functionality required by the test formats (such as drag-and-drop) simply could not be implemented in PHP. Therefore, all pages of the application contain a check, which will show a warning instead of the page content if JavaScript is disabled in the user's browser.

3.1 General Design of the JavaScript Components

As the application at hand requires the JavaScript components to efficiently handle both complex data structures (that is, all the data needed for defining a test and its single questions) and a user interface with a variety of different functions (such as, for example, adding new questions, changing questions, dragging items into containers, displaying a crossword grid, etc.), it was designed based on the Model-View-Controller (MVC) design pattern. The MVC pattern suggests organizing code by a strict separation of concerns, where the so-called 'model' manages all the data for the application, while the 'view' takes care of the visual representation that users interact with, and the 'controller' acts as an intermediary between the two, updating the model according to changes that are made by the user through the view [INT 2].

Combining this concept of a separation of concerns with an object-oriented programming approach resulted in the following architecture of the JavaScript code: The general JavaScript file used by all parts of the application (js_general.js) provides class declarations for the classes 'E_Test', 'TestItem', 'View', and 'Control'. These contain general properties and methods that are needed by all (or at least several of) the different parts of the application (such as, for example, View-object methods for disabling or enabling the buttons on a page). In each of the JavaScript files for the three test types, three main objects are constructed: a test object, which inherits from 'E_Test' and represents the data model, a view object inheriting from 'View' for handling the user interface, and a control object that constitutes the controller and inherits from 'Control'. The different items of the test, as they are represented in the model, are defined by specific sub-classes of the 'TestItem' class that differ between testing formats (while dynamic multiple-choice and crossword tests have 'Question' objects with different properties, drag-and-drop tests have two different kinds, 'Item' and 'Container' objects). These sub-classes are used for storing all test items in a uniform format within the test model.

While the index and the contact page do not need a controller and a data model, as no data is manipulated here, they still both have a view object inheriting from the 'View' class, as some of their user interface functionality is shared with the pages for the testing formats. When one of the pages interacting with tests is called, the control object is initialized and takes care of initializing the model, as well as the view, with regard to the current mode ('new', 'edit' etc.). The initialization of the view includes the definition of event handlers for buttons and other parts of the user interface. Whenever such an event handler requires a communication with the model (e.g. when a new question is added via the user interface, it has to be added to the model as well), a control method is called that takes care of transmitting the changes, thus keeping a strict separation of concerns.

4 Implementation of the Testing Formats

While the aspects of the JavaScript code discussed so far are the same for all parts of the application, the following sections will explain the most important components of the code for each of the specific testing formats.

4.1 Crossword

For the creation of crossword tests, the user is asked to provide a list of questions and answers via text input fields. Answers are checked against a regular expression, making sure that they do only contain the letters of the English alphabet and/or numbers, so that no special characters or white spaces cause difficulties in the display of the crossword. Once at least two questions and answers are provided, a crossword can be generated by mouse click. The code for actually generating the crossword grid from the list of answers is stored in a separate file (crossword_generator.js) and constitutes a self-contained module with functions to receive a word list as input and give back the calculated crossword grid as a JavaScript object. Thus, this code could be used in any application, regardless of the form in which the crossword is to be displayed.

The algorithm of the crossword generator performs the following general steps: Given the list of words, it starts with a grid of 0 by 0 fields, to which it adds the first word in the list either in vertical or horizontal direction (decided randomly). Then the next word is retrieved from the list, and all possible positions for this word using the opposite orientation are computed (so if the first word was placed in horizontal direction, vertical direction will be tried first for the second word). A possible position is a position that shares a letter with at least one of the words placed already in the opposite direction, but does not border on or interfere with any other placed words. For all possible positions, the number of (legitimate) intersections is counted, and the position with the most intersections is chosen. If no possible position is found, the other orientation is tried as well (as this might

result in possible positions if more than one word has been added before already). Once the best position is determined, it is stored for this word, and the crossword grid is expanded by adding the fields that are necessary to change it into a rectangle that accommodates the newly added word. This process is repeated for all words. If words cannot be placed during one round, the algorithm runs through them again after finishing with the list (as words that came after them in the list might have created possible positions for them), repeating this process until no more words were added during one run through the list of unplaced words.

The entire grid creation process is repeated as often as possible during three seconds (testing showed that this amount of time suffices for obtaining good results), each time starting with the word list given in a different randomly created order. Thus, the generator ends up with a number of different crossword grids at the end of the three seconds processing time (how many can be created depends on the number of words given and the processing speed), from which it chooses the best by applying the following criteria: First, as many words as possible should be placed. This is the foremost criterion, as the main aim is of course to include all given words in the crossword. Second, the ratio between how close the rectangle of the grid comes to a square and the size of the rectangle is taken as a criterion, as a good crossword grid without too many empty spaces should be as close to a square as possible and at the same time as small as possible. Should this ratio be the same for different crossword grids in question, the size is taken as a final criterion, again preferring smaller to bigger grids. The crossword grid thus determined as the best solution is returned, including an object literal holding the values for all fields of the grid, as well as lists of the placed words (along with their positions and the numbers by which they are identified in the grid) and unplaced words (of which ideally there are none).

As this calculation process, which takes up some seconds, would freeze the page during its execution if it were started by a simple function call, a Web Worker [INT 3] is created for carrying out the generation process. This has the advantage of running in a different thread, so that it does not affect page performance and allows for an animation to be displayed while the code is

running, as an indication for the user to wait. Before trying to spawn the worker, a check for browser support is carried out, and, in case Web Workers are not supported, the generator script is loaded directly and started via a function call. In this case, it is not possible to display the animation, but at least not the script does not break down either.

Once the generated crossword grid is delivered, the regular crossword JavaScript file takes care of storing its data to the model and displaying a visual representation for the user, adding a list of the questions below the grid. A notification is displayed in case not all of the given words could be placed, prompting the user to delete or change the unplaced words. When the user is satisfied with the generated grid, it can be stored to the database (see section 5).

Apart from the actual generation of the crossword grid, another challenge of implementing an online crossword task consisted in finding an appropriate way of allowing the user to fill in the crossword in the run mode. Simply making each field where a letter should be inserted a text input field which the user would have to click on or navigate to via the keyboard would result in a rather cumbersome interaction with the crossword. An alternative solution was found by not displaying any input fields in the grid by default, but instead asking users to click on the question for which they want to provide the answer. Upon this click, the fields containing this answer in the grid are turned into input fields, and the focus switches to the next input field of the word as soon as a letter is entered, so that the user can simply type in the word as a whole at once, making the interaction with the grid easier and more efficient.

4.2 Dynamic Multiple-Choice

While the implementation of dynamic multiple-choice tests did not require the creation of a complex algorithm, such as the one needed for the generation of crossword grids, a few challenging aspects of this testing format shall be discussed as well. As dynamic multiple-choice questions provide users with one answer option at a time, for which they have to correctly decide between true and

false before the next option is shown, it seemed suitable to not fix the number of answer options for each question. Thus, the new and the edit mode, where new questions can be added via form fields, also provide buttons by which as many incorrect answer options as desired can be added to each individual question. This way, one test can contain questions with different amounts of answer options, making it more challenging for learners to judge whether any current option they are presented with might be wrong or correct. As, especially with a lot of answer options per question, the display of existing questions on the interface page could take up a lot of room causing the user to scroll a lot, a more compact solution for the display was found: By default, only the question text of existing questions is displayed, along with a clickable symbol by which the answer options of the question can be shown (and hidden again), so that they are available when necessary for further editing.

It follows from allowing for a flexible number of answer options that, in the run mode, not only the solutions for questions should be hidden from the users (as explained in section 3), but also the number of options for each question. For this reason, only the questions themselves are retrieved from the database when a test is started, and answer options are then dynamically loaded via AJAX requests one at a time (or several at once, in case the solution to a question is to be displayed). In order to avoid making complicated database requests each time an option is called for, the server-side component retrieves all options of all questions when the test is first loaded from the database and stores them to the session, from which they are retrieved when an AJAX request is received. As a user could open several parallel tabs for running tests at the same time using the same session, the data stored for each test in the session is associated with a unique key created by using the timestamp of the time when the test was loaded.

For displaying dynamic multiple-choice questions to the user in a clear and pleasant design, a possibility of displaying the answer options along with the input element for users to choose between correct and incorrect (without overloading the display with too much text) had to be found. Here, one of the graphical tools provided by the HTML5 standard proved useful: Simple symbols for marking an

answer option as wrong (an x mark) or right (a tick) are drawn onto the page using the SVG element [INT 4]. Apart from saving loading time by taking the place of images, this has the advantage of making it easy to change the colors of the symbols. As shapes drawn within an SVG element are represented in the DOM as HTML elements, their properties can be accessed and manipulated just like other parts of the page with JavaScript. This is put to use in the run mode, where the symbols' colors are changed to indicate whether a user's choice for an answer option was correct or incorrect.

Lastly, dynamic multiple-choice tests are the only test type where it was considered useful to allow for re-using questions from existing tests. Thus, the interface for creating or changing tests contains a selection box, which can be displayed upon request, and by which questions that already exist in the database can be imported to the current test. When such questions are added, they are displayed on the interface page along with the questions added manually via the form fields and can be changed or deleted just as the latter. This option to include questions from other tests, as well as the flexible amount of answer options per question, called for a database structure which is slightly more complicated than those of the other test types, and the details of which are explained in section 5.

4.3 Drag-and-Drop

The implementation of the third type of tests makes use of the drag-and-drop functionality which is part of the HTML5 standard [INT 5]. While this feature is supported in all modern browsers, testing showed that the interpretation of the different kinds of events associated with the drag-and-drop functionality differs to some extent between browsers. The final version, which works across all major modern browsers, uses the following events: The elements which can be dragged (items) have an attribute 'draggable', which must be set to true, and listen to the events 'dragstart' and 'dragend'. Thus, when the dragging begins, the ID of the element which is dragged is saved as transfer data, the opacity of the element is reduced (as a visual cue for the user, indicating which item is currently being

dragged), and an image of the element is attached to cursor, following it along the screen. Once the drag ends, the element's opacity is reset to normal.

Elements into which a dragged item can be dropped (containers), on the other hand, listen to four different events: 'dragenter' and 'dragover' indicate what happens when a dragged element is moved over the container in question (both have to be present, to allow for support by all browsers). When those two events are registered, the default behavior of the container element is prevented (as all elements by default do not allow for a drop), and its border color is changed, in order to visually indicate that a drop is possible here. The 'dragleave' event is registered when a dragged element is moved out of a container and triggers resetting the border color to normal. Finally, the 'drop' event handler specifies what happens when an item is dropped into a container: The HTML-element representing the item is attached to the container (the ID stored in the transfer data is used to identify the element for this purpose), the border color of the container is reset, and the model is updated to reflect the relationship of this item belonging into this container.

In order to make creating and editing drag-and-drop tests as user-friendly as possible, the functionality described so far is not only used when a test is run, but also in the new and edit mode. This means that when working with a test, users can directly indicate which of the items they have created go into which containers by dragging and dropping them, so that they get an exact impression of what the final test will look and feel like. Another special feature of this test format is that in the run mode, there are two possibilities of how the test can be displayed: Learners can either see all items at once, drag them into the containers, and have the entire test checked, or they can choose to be presented with one item at a time. In the last case, an individual AJAX request is sent after each item to check whether the user's assignment to a container was correct or incorrect, in order to provide immediate feedback without making the solutions visible in the JavaScript code.

5 Database Structure

Complementing the discussion of the implementation of the different testing formats, this chapter shall now explain how the data representing each test is stored in the MySQL database of the application. For a graphical representation of the database structure, see Appendix I.

The database is organized into a general and a test-specific level. On the general level, each test is represented as one row in the table 'tests', where general information, such as the name of the test and its creation date, is stored. The table 'test_types' represents all testing formats currently supported by the application, associating each format with its abbreviation, description, and so on. This information is primarily used to generate the content of the index page, where the user is presented with the different formats that can be managed with the e-test editor. The IDs of the test types are used in the table 'tests' to identify which specific tests belong to which format. Similarly, the table 'test_levels' contains the labels and numbers associated with the IDs of the test levels specified in the 'tests' table. This assignment to a level is not actually put to use in the application itself, but was introduced here to indicate how information that might be specific to an actual learning platform, such as the organization of tests into levels, can easily be integrated into the e-test editor once it is put to use in the context of such a platform.

On the second level, different data for each test, depending on its format, is stored in the remaining tables of the database. The simplest structure here is the one for drag-and-drop tests: Each test is associated with a variable number of items (represented each by one row in the table 'dragdrop_items') and containers (represented each by one row in the table 'dragdrop_containers'). Item and container entries consist of an ID, the text to be displayed on the element, the test ID as a foreign key, and, in the case of items, the ID of the container the item belongs to.

The table structure for dynamic multiple-choice tests is somewhat more complicated. As each question can be associated with an arbitrary number of tests (see section 4.2), a look-up table ('dynmc_lookup') for representing the many-to-many relationship between tests and questions is necessary. The table containing the questions ('dynmc_questions') holds the question text, as well as the correct answer (as each question has one correct answer only), but cannot be used for storing incorrect answers, as the application allows for a variable number of the latter. Thus, incorrect answers are modeled as rows of a separate table ('dynmc_incorrect') and linked to the question they belong to by this question's ID as a foreign key.

Lastly, the database contains two more tables for data representing crossword tests. The table 'crossword_questions' contains questions along with their answers, the ID of the test they belong to, and information on where they are positioned in the crossword grid, that is, the x- and y-coordinates of their start field, their orientation (0 for vertical, 1 for horizontal), and the number by which they are identified in the grid display. This information, along with the general grid information stored in the table 'crossword_grid', which holds the number of fields in the x- and the y-direction for each test, is enough to compute the entire crossword grid. Strictly speaking, the way the crossword generator is currently implemented, the information given in the table 'crossword_grid' would not even be necessary for calculating the grid, as the positions of all the words taken together already define the extensions of the rectangle containing the grid. Storing these numbers, however, makes reconstructing the grid from the data stored in the database considerably easier. It furthermore allows for changes to the way crossword grids are constructed, for instance, if one should decide to adapt the algorithm to allow for empty rows or columns at the borders of the crossword (in order to make it a proper square, for example). Therefore, this kind of representation was chosen as the most suitable for representing crossword tests.

6 General Features

Apart from the parts of the application that manage the manipulation, display, and storage of the tests in different formats, some more general components of the e-test editor still need to be discussed. The following sections will deal with them.

6.1 Index Page

The first part of the application that users encounter is the index page. This page retrieves information about the test types that are currently available in the database, including their descriptions and icons. The latter are used for displaying buttons, which can be clicked in order to create a new test using the respective format. More testing formats could thus easily be integrated. Assuming that not all users might know what the different formats consist in, but avoiding cluttering the page with too much text, the descriptions of the formats are initially hidden and only appear when the user's mouse hovers for a moment over one of the buttons.

The lower part of the page displays the interface for working with existing tests. The script retrieves information about all tests currently stored in the database from the 'tests' table and displays them in an HTML table, initially ordered alphabetically by test names. As, depending on the size of the learning platform, the number of tests to be managed by the application could eventually amount to quite a huge number, it was considered useful to provide the user with some help for finding specific tests in the table. Therefore, different ways of manipulating the display of the table were introduced: First, the checkboxes associated with test types to the right of the table register when they are clicked, and, depending on whether they are checked or unchecked after the click, show or hide the rows of the table that contain tests of the respective type. Second, clicks on the table header allow for ordering the table according to different criteria. A click on one of the table column headlines triggers a function, which will, according to which headline was clicked, either take the test name, the test type,

or the test creation date as the criterion by which the rows are sorted, using the JavaScript sort-function. This function is set to grab the text content of the respective table cells and computes the order of the rows accordingly. The direction of the sorting (ascending or descending) can also be manipulated by clicking on the arrow symbol appearing in the table header, causing the function to reverse the sorting if required.

6.2 Contact Page

There is another page that is not associated with the specific test types: The contact page can be reached via a button on the index page or by clicking on the author's name in the footer of each page. Apart from some information about the application and the author, this page contains a simple web form for the purpose of contacting the author. A message submitted by this form is handed to the mail function of PHP, which would (if the application ran on a server connected to the Internet and configured appropriately) take care of sending the message to the specified e-mail address. As the application is currently running on the local server provided along with it, a fake e-mail address is given here, and the message is saved the 'mailoutput' folder of the server environment.

6.3 Color Changer

In order to make working with the e-test editor enjoyable for a variety of users, it was attempted to find a way to somewhat adjust its interface design to the users' different tastes. This was achieved by choosing neutral shades of gray as colors for most of the parts of the application's interface pages and combining these with one theme color used for accentuated components, such as the headers of the different sections of a page. This theme color, though initially set to blue, can be changed by the user, who is given a choice between four different colors at the bottom right corner of the footer. Changing the color is made possible by giving all elements that should have the theme color as their background, font, or border color a respective class, and then, upon a click on one of the color buttons,

changing the color of these classes using JavaScript. In order to ensure that the setting is effective also for future elements that are dynamically added to the page, rather than just selecting the elements with the classes in question and resetting their CSS, a style declaration inside the head of the page is updated each time a color button is clicked.

As administrators of e-learning platforms might work with the application often, they might want it to remember their color choice, so that they do not have to reset the theme color each time they access the editor. This is made possible by using Local Storage, another part of the HTML5 standard [INT 6]. Each time a color button is clicked, a variable in the browser's local storage is set to point to the respective color. Whenever a page is loaded, the storage is checked for this variable (given the browser supports local storage). If a color value is found, the theme color is updated to this color. This mechanism not only helps the application 'remember' the color when the application is closed and re-using it the next time it is opened, but also provides an easy way of making a color change effective throughout the application (and not only on the page where the color button was clicked initially).

6.4 Inline Editing and Deletion

Another measure that was taken to improve the user-friendliness of the application has to do with the way test data is edited in the new and edit modes: While new questions are mostly added via simple form fields, a way had to be found to allow users to easily make changes in the questions that have already been added to the page. Displaying them all in form fields or copying them to the form fields for adding questions both seemed rather confusing options. Therefore, a kind of inline editing was introduced to simplify making changes. For this purpose, all elements that the user should be allowed to change while working on a test, are given the class 'editable', once they are added to the test. The general 'View' class provides all view objects with a method that turns the content of an element with the 'editable' class attribute into a text input field upon clicking on it, setting the input

field's value to the former text content of the element. Thus, the text the element contained before the click is ready for editing within the input field. When the element loses focus or when the 'enter' key is pressed (which supposedly is a familiar way of confirming one's input), the value of the input field is checked for appropriateness (e.g. for not being empty), and, if the check is successful, the input field is replaced by the new value. Also, an update of the model is initiated to reflect the changes made.

Following the same line of thought of making the manipulation of added elements possible right at the place where the elements are displayed, the deletion of elements is done by clicks on 'X' signs that appear next to the elements on hover. These are hidden when the mouse is not currently over the element, in order not to clutter the page and to make it clearer which element will be deleted by a click. As deletion should not happen by accident, it is only carried out after asking the user for confirmation, which, if given, again triggers an update of both the view and the model, reflecting the change.

6.5 Exporting Tests

Administrators of e-learning platforms might consider it useful to be able to generate and save static versions of the tests they create, for example to send the solutions of a test to instructors or teaching assistants, who might not have access to the administration interface, or to include a picture of a test in some printed material accompanying an online course. As simply printing pages displaying tests in the view mode might result in rather unsatisfactory displays of the tests (as the dynamically created styles are not reflected properly, and buttons etc. are included in the display), a possibility of creating suitable images of the tests and allowing the user to save or print these, had to be found instead. The canvas [INT 7] and SVG elements introduced by the HTML5 standard made creating such export options possible. Basically, the conversion of the HTML defining the display of a test into an image works as follows:

When the user clicks one of the export buttons in the view mode ('Download as PNG', 'Download as JPEG', 'Print Test'), an SVG element is created, into which HTML elements can be added inside a 'foreignObject' tag. This tag is filled with the test name as a headline, a copy of all HTML elements displaying the test itself, and a footer line. Next, a Blob object is generated from the SVG element. This Blob object can be used to generate a URL containing all the data of the original SVG, which then allows for the creation of an image with the URL as its source. As only styles defined inside the SVG itself (and not those in the style sheets linked on the page) will be applied to this image, all styles of all HTML elements inside the SVG's 'foreignObject' tag are copied to a style tag at the beginning of the SVG element before the Blob object is created. The image could then be displayed in an image tag, but, in order to allow for exporting it to non-SVG formats, such as PNG or JPEG, it is drawn onto a canvas instead. If the test is to be printed, the content of the page is temporarily replaced by this canvas, and printing is initialized. Once the print window is closed, the page's original contents are restored. If the user has chosen to download the test as an image, however, this is achieved by using the 'toDataURL' method provided by the DOM interface of the canvas element. This method allows for the generation of image files from the canvas content in different formats and is in this case set to either create a PNG or a JPEG file. In the latter case, the canvas' background color is set to white before the image is drawn onto the canvas (as a transparent background would otherwise appear black in the JPEG image). A hidden download link for the image file is created next and triggered right away, so that the user is presented with the download prompt of the browser and can decide where to save the image file. The elements added to the page during these steps (the canvas and the download link) are removed once the process is finished.

It is important to note that the use of the technologies described here comes with some issues. First, at the time of writing this documentation, Internet Explorer did not yet support the 'foreignObject' element [INT 8], so that the process employed for converting HTML to an image will not work in this browser. The application tests for support of this element (as well as for the

canvas element) before attempting to generate the image and will issue a warning in case no such support is found, suggesting the use of a different browser. Second, due to cross-origin policy issues, the 'toDataURL' method used to generate image files from the canvas to which the image was drawn using the Blob URL does not work in Chrome and Opera in this specific case, as long as the application is running on a local server (see [INT 9], for example). Thus, exporting images as PNG or JPEG files should, if the application is not installed on an actual web server, be tested in a modern version of the Firefox browser. Printing tests, however, will function as expected in both Chrome and Opera, as well as Firefox. Again, the application is configured to account for this problem by issuing a warning, if the download of image files is not available.

As the export options are not an essential feature of the e-test editor, and one of the goals of this project was the exploration of how modern HTML5 technologies can be applied to the implementation of web-based e-assessment, they were included in spite of the shortcomings explained above, in order to further demonstrate how the graphical tools provided by the HTML5 standard could improve the implementation and management of electronic tests. This example, along with the use of drag-and-drop events, Web storage, Web workers, and the dynamic manipulation of graphical elements in a test display, illustrates that HTML5 technologies provide a lot of powerful possibilities for Web developers to integrate interactive and complex tasks into e-learning environments.

References

Internet Sources (cited in the documentation)

[INT 1] http://php.net/manual/en/book.mysqli.php, 06 May 2015.

[INT 2] http://www.addyosmani.com/resources/essentialjsdesignpatterns/book, 06 May 2015.

[INT 3] http://www.w3.org/TR/workers/, 06 May 2015.

[INT 4] http://dev.w3.org/SVG/proposals/svg-html/svg-html-proposal.html, 11 May 2015.

[INT 5] http://www.w3.org/TR/2010/WD-html5-20101019/dnd.html, 07 May 2015.

[INT 6] http://www.w3.org/TR/webstorage/, 07 May 2015.

[INT 7] http://www.w3.org/TR/2011/WD-html5-20110525/the-canvas-element.html, 11 May 2015.

[INT 8] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/foreignObject, 12 May 2015.

[INT 9] http://blog.chromium.org/2011/07/using-cross-domain-images-in-webgl-and.html, 11 May 2015.

Further Internet Sources (used as reference during the coding process)

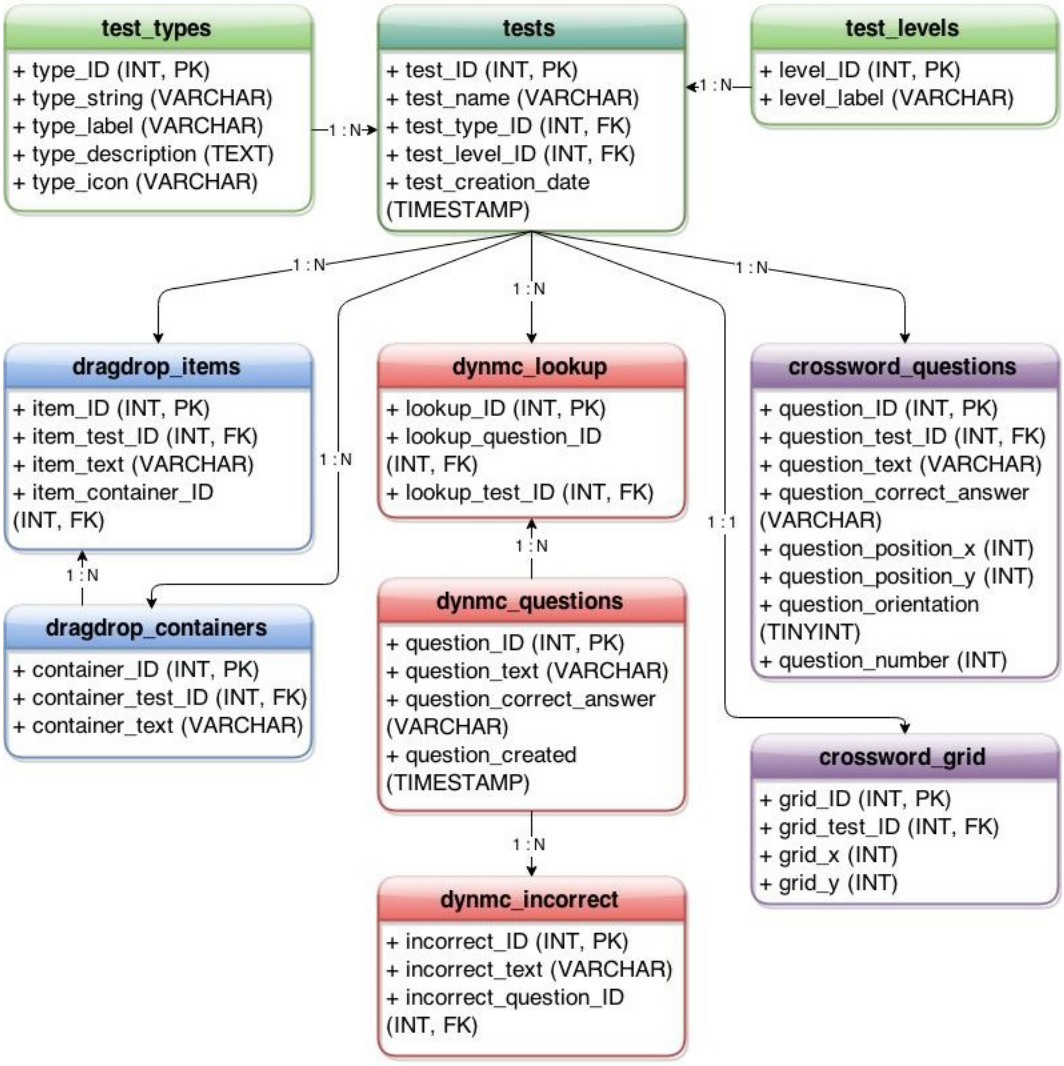
http://api.jquery.com/, 07 May 2015.

https://php.net/manual/en/, 07 May 2015.

http://www.w3schools.com/, 07 May 2015.

https://developer.mozilla.org/en-US/, 07 May 2015.

Appendix I: Database Structure



Graphical Representation of the Database Tables and their Relations (created by the author with https://www.draw.io/, 07 May 2015)

Declaration of Authorship

I hereby declare that my Master's Thesis in partial fulfillment of the requirements for the degree of Master of Arts (M. A.) with the title:

Web-Based E-Assessment Beyond Multiple-Choice:

The Application of PHP- and HTML5 Technologies to Different Testing Formats

is entirely my own work and does not involve any additional human assistance. I also confirm that it has not been submitted for credit before, neither as a whole nor in part and neither by myself nor by any other person. No resources or aids other than those cited have been used. All quotations and paraphrases but also information and ideas that have been taken from sources used are cited appropriately with the corresponding bibliographical references provided. The same is true of all drawings, sketches, pictures and the like that appear in the text, as well as of all Internet resources used. I am aware that in case of being found guilty of plagiarism, I shall bear the costs of any ensuing legal disputes as well as any additional sanctions.

Marburg 19 May 2015