# TEMPLATETOR PART

*Templatetor is a Mustache wrapper with some addition functionality like replacing mustache placeholders in already rendered DOM elements and checking for them in the string. Also it has a container of static view values, that makes language templating easier.*

Templatetor.extendView(o); -

Adds values to templatetor's view. View is a assoc array for replacement using Mustache. @param o {Object} view to add. @returns {Templatetor} current object (flow);

Templatetor.setTemplate(o); -

Sets a template where Mustache's placeholders should be processed. @param o {jQuery|String} template text/object. @returns {Templatetor} current object (flow);

Templatetor.replace(b); -

Setter of replacement mode. If replacement mode is on. @param b {Boolean} replacement mode switch. @returns {Templatetor};

Templatetor.render(); -

This method performs replacement of Mustache's placeholders with data from view. If replacement mode enabled, method uses {@link $.fn.replaceText} to perform replacements. @returns {String} null if replacement mode is one, otherwise rendered string;

Templatetor.clean(); -

Cleans templatetor's view, template data and disabling replacement mode. @returns {Templatetor}.

# SCORER PART

Scorer.start(); -

Starts monitoring;

Scorer.end(); -

End monitoring;

Scorer.addScore(s); -

Adds points to user's global score. @param s {Number} amount of points;

Scorer.addUserStepScores(s); -

Adds points to user's global score. @param s {Number} amount of points;

Scorer.getFormattedStartTime(); -

Gets a time when monitoring was started. @returns {String} formatted time string and false, if monitoring is not finished yet.

Scorer.getFormattedEndTime(); -

Gets a time when monitoring was ended. @returns {String} formatted time string and false, if monitoring is not finished yet.

Scorer.getTimeDifference(); -

Gets time time difference between start and end of monitoring in seconds. @returns {Number} time diff in sec or false, if monitoring is not finished yet.

Scorer.getScore(); -

Gets a total users score. @returns {Number} user's total score or false, if monitoring is not finished yet.

Scorer.getScoreInPercent(); -

Gets a total users score in percent. @returns {string} user's total score or false, if monitoring is not finished yet.

Scorer.getUserStepScores(); -

Gets an array of users scores. @returns {Array} of user's step scores or false, if monitoring is not finished yet.

Scorer.getTotalScore(); -

Gets a trainer's score. @returns {Number} user's total score or false, if monitoring is not finished yet.

Scorer.setTotalScore(); -

Sets total score of the trainer. @param s {Number} amount of points.

# I18N PART

I18N.setPath(p); -

Sets a path to language files .json. @param p {String} path to language folder. @returns {I18N} current object (flow);

I18N.setAvailableLanguages(a); -

Sets an object of all available languages. @param a {Object} assoc array of allowed languages. @returns {I18N} current object (flow);

I18N.setLanguage(l); -

Sets current language of trainer. Language, specified by this setter, may be loaded with {@link I18N.loadLangugae}. @param l {String} language code. @returns {I18N} current object (flow);

I18N.loadLanguage(callback); -

Loads language data from json lang database. @param callback {function} a callback, that will be called after a successful download;

I18N.getConstants(); -

Gets the language strings. @returns {Object} assoc array of lang constants;

I18N.getLangNames(); -

Gets language names from available languages. @returns {Array} array of language code-local name pairs;

I18N.getCurrentLang(); -

Gets a language code of current language. @returns {String} language code;

getNumEnding(); -

Returns correct word ending using cases. @param iNumber Integer Number used to make ending. @param aEndings Array Array of words or endings for numbers (1, 4, 5), e.g. ['яблоко', 'яблока', 'яблок']. @returns String;

# SERVICE PART

Service.setConfigPath(p); -

Sets trainer's config file path. @param p {String} path to trainer's config file. @returns {Service} current object (flow);

Service.getTrainerConfig(); -

Gets trainer's settings, if they are loaded. @returns {\*};

Service.loadConfig(callback); -

Loads trainer's config file. @param callback {function} callback to call after successful file loading;

Service.getUrlParam(name, url); -

Returns a value from URL query. @param name {String} query param's name. @param url {String} source for value extraction. Optional parameter, by default window.location.search. @returns {String} value of param;

Service.appendScript(url, callback); -

Loads external script file and appending it to DOM. @param url src prop of <script> tag. @param callback func;

Service.about(); -

About trainer. @returns string with info about jTrainer;

Service.notifyServer(callback); -

Sends data about trainer start to SSU server. @param callback func that will be called after notifying;

Service.pushResults(options, callback); -

Pushes user's results to SSU server. Old Behaviour, just for backward compatibility. @param options additional options to add to userResult variable. @param callback func i'll call when transferring is done;

Service.pushResultsPromise(options); -

Pushes user's results to SSU server. Behaviour for done and fall is defined by .done and .fail methodes. @param options additional options to add to userResult variable;

Service.pushResultsEarly(callback); -

Pushes user's results to SSU server. @param callback func i'll call when transferring is done;

getActionPath(); -

Return action URL (path without id);

Service.getStorageUrl(); -

Return storageUrls holder (put\_url, and get\_url). @returns promise with url;

Service.fetchStorageInfo(callback); -

Get stored trainer-related data from SSU server. Data is cross-session and cross-user visible. Value 'max\_score' is rewrited by server. @param callback func that will be called after fetching data, if data is fetched well;

Service.pushStorageInfo(push\_data, callback); -

Push data to shared storage. params 'max\_score' is ignored, and rewrited by server. @param push\_data - hash, data to bush. @param callback - callback function, run if data is pushed well;

Service.getHelpModal(callback); -

Get's help from SSU server. @param callback func i'll call when transferring is done.

# SCRIPT INVOKER

*This is a class to invoke functions by string including ability to execute function from .js source file.*

ScriptInvoker.autoClean(b); -

Auto clean invoker stack switch. @param b auto clean if true, otherwise saves stack values. @returns {ScriptInvoker} current object (flow);

ScriptInvoker.addSource(src); -

Adds a source file to execute function from. @param src path to js file. @returns {ScriptInvoker} current object (flow);

ScriptInvoker.addCommand(funct, args); -

Adds a function and it's arguments to invoke. @param funct function name. @param args arguments of function. @returns {ScriptInvoker} current object (flow);

ScriptInvoker.loadScripts(callback, i); -

Loads scripts from 'scripts' array. @param callback funct to call after loading. @param i pointer to current scr;

ScriptInvoker.Invoke(); -

Starts to invoking commands;

ScriptInvoker.clear(); -

Clears sources and commands stacks. @returns {ScriptInvoker} current object (flow).

# STEP INVOKER

*This is an instance of ScriptInvoker used by engine, especially by rotator. Rotator invokes all commands of StepInvoker after calling nextStep(). Used for executing necessary scripts for elements etc. @type {ScriptInvoker}.*

# COLLECTIONS PART

getIndexOfKey(k); -

Returns index of elements pair in Map. @param k {\*} key to find. @returns {Number} index of elements pair if exists, otherwise -1;

Map.add(k, v); -

Adds key-value pair in map. @param k {\*} a pair's key. @param v {\*} a pair's value. @returns {Map} current object (flow);

Map.get(k); -

Gets a value pair in map by key. @param k {\*} a pair's key. @returns {\*} value if exists, otherwise null;

Map.size(); -

Returns a size of map. @returns {Number} amount of elements;

Map.keys(); -

Returns all keys from this map. @returns {Array} keys of map. @override;

Map.values(); -

Returns all values from this map. @returns {Array} values of map;

Map.randomize(); -

Ramdomize map's elements. @returns {Map} randomized map;

Map.iterate(callback); -

Map's iterator. For each map's key-value pair calls callback with (key, value) params. @param callback {function} function to call. @returns {boolean} true, if there are elements in map, otherwise false;

Map.slice(start, end); -

Create a sliced copy of this map. @param start {number} first element position. @param end {number} last element position. @returns {Map} new sliced map;

Map.merge(m); -

Merge another map's key-value pairs with this one. @param m {Map} map to merge. @returns {Map} current object (flow).

# ELEMENTS PART

Element.setName(nm); -

Sets element's name. @param nm {String} name of the element. @returns {Element} current object {flow);

Element.getName(); -

Gets element's name. @returns {String} name of the element;

Element.setLabel(l); -

Sets a label of element. @param l {String} label of element. @returns {Element} current object {flow);

Element.getLabel(); -

Gets a label of element. @returns {String} label of element;

Element.setValue(v); -

Sets a value of element. @param v {String|Number} element's value. @returns {Element} current object {flow);

Element.getValue(); -

Gets a value of element. @returns {String} element's value;

Element.setId(id); -

Sets #id to element. @param id id tag's value. @returns {Element} current object {flow);

Element.getStyle(); -

Gets element's style. @returns {string};

Element.setStyle(st); -

Sets element's style. @param st element's style. @returns {Element} current object {flow);

Element.getId(); -

Gets element's id. @returns {string};

Element.getParams(); -

Returns all params of elements in one string. @returns {string} string of params;

Element.render(); -

Renders the element. SHOULD BE OVERRIDDEN. @returns {String} rendered element;

Element.addAttribute(n, v); -

Adds attribute to element. @param n {String} attribute's name. @param v {String|Number} attribute's value. @returns {Element} current object {flow);

Element.getAttributes(); -

Gets attributes of element. @returns {String} of attribute=value pairs;

Element.addClass(c); -

Adds class to element. @param c {String} class name to add. @returns {Element} current object {flow);

Element.getClasses(); -

Gets classes of element. @returns {String} of classes;

Element.removeClass(c); -

Removes class of element. @param c {String} class name to remove. @returns {Element} current object {flow);

Element.clearClasses(); -

Clears all classes of element. @returns {Element} current object {flow);

Radio.checked(b); -

Makes radio checked. @param b {Boolean} isChecked switch. @returns {Radio} current object {flow);

Radio.render(); -

Renders the element. @returns {String} rendered element;

Radios.randomize(); -

Scramble options inside radio group;

Radios.addRadio(label, value, checked); -

Adds a radio to radio-group. @param label {String} text of option. @param value {String} value of option. @param checked {Boolean} if this radio checked. @returns {Radios} current object {flow);

Radios.render(); -

Renders the element. @returns {String} rendered element;

CheckBox.render(); -

Renders the element. @returns {String} rendered element;

Select.randomize(); -

Scramble options inside select group;

Select.addOption(label, value, checked); -

Adds an option tag to element. @param label {String} text of element. @param value {String} value of element. @param checked {Boolean} is this option checked. @returns {Select} current object {flow);

Select.disabled(b); -

Makes element disabled. @param b {Boolean} isActive switch. @returns {Select} current object {flow);

Select.render(); -

Renders the element. @returns {String} rendered element;

TextInput.placeholder(text); -

Sets placeholder text. @param text {String} placeholder text. @returns {TextInput} current object {flow);

TextInput.disabled(b); -

Makes element disabled. @param b {Boolean} isActive switch. @returns {TextInput} current object {flow);

TextInput.render(); -

Renders the element. @returns {String} rendered element;

TextArea.render(); -

Renders the element. @returns {String} rendered element;

DroppableArea.render(); -

Renders the element. @returns {String} rendered element;

DraggableGroup.randomize(); -

Scramble options inside Draggable group;

DraggableGroup.addOption(label, value); -

Adds a draggable to draggable-group. @param label {String} text of option. @param value {String} value of option;

DraggableGroup.render(); -

Renders the element. @returns {String} rendered element;

WolframAlpha.doQuery(callback); -

Performs a query with WolframAlpha API through SumDU server. @param callback {function} callback to call after loading;

WolframAlpha.setQuery(q); -

Sets a query to perform. @param q {String} query sting. @returns {WolframAlpha} current object (flow);

hasQuery(); -

Checks if query was specified. @returns {boolean} true if query is specified, otherwise – false;

WolframAlpha.plot(callback); -

This method src to plot image, drawn by WolframAlpha. @param callback {function} callback to call after receiving img src;

GoogleCharts.setType(type); GoogleCharts.setData(data); GoogleCharts.setOptions(options); GoogleCharts.setLibrary(library);

GoogleCharts.doQuery(id); -

Performs a query with WolframAlpha API through SumDU server. @param callback {function} callback to call after loading;

LateX.setFormula(f); -

Sets a LateX text to render. @param f {String} LateX formtted string. @returns {LateX} current object (flow);

LateX.render(); -

Renders LateX formula. @returns {String} rendered formula as an img tag;

# COGWHEEL PART

Cogwheel.setCogWheelElement(o); -

Sets a wrapped DOM element to show on loading. @param o {jQuery} wrapped element. @returns {Cogwheel} current object (flow);

Cogwheel.setCogWheelDescElement(o); -

Sets a wrapped DOM element where loading description situated. @param o {jQuery} wrapped element. @returns {Cogwheel} current object (flow);

Cogwheel.setText(s); -

Sets a loading description. @param s {String} description. @returns {Cogwheel} current object (flow);

Cogwheel.show(); -

Shows a loading splash. @returns {Cogwheel} current object (flow);

Cogwheel.hide(); -

Hides a loading splash. @returns {Cogwheel} current object (flow).

# BREADCRUMB PART

BreadCrumb.setBreadCrumbElement(o); -

Sets a breadcrumb to display step names. @param o {jQuery} wrapped element. @returns {Cogwheel} current object (flow);

BreadCrumb.setBreadCrumbStepNames(step); -

Sets a breadcrumb at top of page to display step names;

BreadCrumb.show(); -

Shows breadcrumb. @returns {BreadCrumb} current object (flow);

BreadCrumb.hide(); -

Hides breadcrumb. @returns {BreadCrumb} current object (flow).

# PROGRESSBAR PART

ProgressBar.setProgressBarElement(o); -

Sets a progressbar to display trainer completition progress. @param o {jQuery} wrapped element. @returns {Cogwheel} current object (flow);

ProgressBar.changeProgressBarState(total, next); -

Changes progressbar completition state;

ProgressBar.show(); -

Shows progressbar. @returns {ProgressBar} current object (flow);

ProgressBar.hide(); -

Hides progressbar. @returns {ProgressBar} current object (flow).

# ROTATOR PART

*Rotator is one of the main object of trainer that is responsible for the rotation of steps.*

Rotator.setPrevButton(o); -

Ties up an wrapped DOM element of Prev Button. @param o {jQuery} wrapped DOM element of Prev Button. @returns {Rotator} current object (flow);

Rotator.setNextButton(o); -

Ties up an wrapped DOM element of Next Button. @param o {jQuery} wrapped DOM element of Next Button. @returns {Rotator} current object (flow);

Rotator.enableNextButton(); -

Enables next button. @returns {Rotator} current object (flow);

Rotator.disableNextButton(); -

Disables next button. @returns {Rotator} current object (flow);

Rotator.enablePrevButton(); -

Enables prev button. @returns {Rotator} current object (flow);

Rotator.disablePrevButton(); -

Disables prev button. @returns {Rotator} current object (flow);

Rotator.setStepsPath(p); -

Sets patch to steps files. @param p {String} step path. @returns {Rotator} current object (flow);

Rotator.setScriptsPath(p); -

Sets patch to step's script files. @param p {String} step's scripts path. @returns {Rotator} current object (flow);

Rotator.setSettingsPath(p); -

Sets patch to step's setting files. @param p {String} step path. @returns {Rotator} current object (flow);

Rotator.setStepSpace(ss); -

Sets an object where steps will be loaded. @param p {jQuery} wrapped DOM element. @returns {Rotator} current object (flow);

toSpaceStep(data); -

Pushes data into step's space. @param data {String} data;

Rotator.getStepsCount(); -

Gets a max score for next step. @param step {Number} step's index. @returns {Numeric} amount of points for this step;

loadStepsSettings(callback); -

Loads step's setting file. @param callback {function} callback to call after loading;

getStepScript(step, callback); -

Loads step's script file and executing it. Every step's script should contain function expressing named the same with step. Firstly, stepClass.preDispatch() method runs. It can be with/without callback. This method goes first and usually is used to perform async things before step will be displayed. Than goes stepClass.mustache() method. This method should return a view object for replacement. This method usually is used to bind data from stepClass and application view. The latest is stepClass.postDispatch(). This method goes last and executes after rendering step is performed. It's great for binding events. It executes in {@link loadStep} method. @param step {String} name of step. @param callback {function} callback to call after loading script file;

getStepData(step, callback); -

Loads html template of a step. @param step {String} step's name. @param callback {function} callback to call after loading html file;

loadStep(step, callback); -

This method is an union of several methods, that performs the whole cycle of loading step. @param step {Number} name of step. @param callback {function} callback to call after loading step;

fadeStepIn(id, callback); -

Methods allows you to navigate through already loaded steps. @param id {Number} index of a step. @param callback {function} callback to call after changing step;

Rotator.switchStep(step); -

Allows to switch step by id, it it's loaded. @param step {Number} step's id;

Rotator.getStepScore(step); -

Gets a max score for this step. @param step {Number} step's index. @returns {Numeric} amount of points for this step;

Rotator.getNextStepScore(step); -

Gets a max score for next step. @param step {Number} step's index. @returns {Numeric} amount of points for this step;

Rotator.getAllStepScores(); -

Gets an array of scores for each step. @returns {Array} of scores;

Rotator.getAllStepNames(); -

Gets an array of step names for each step. @returns {Array} of step names;

Rotator.nextStep(callback); -

Performs transition to the next level. @param callback {function} callback to call after changing step. @returns {boolean} true, if the transition was successful, otherwise – false;

Rotator.nextResults(callback); -

Performs transition to the results after clicking 'End trainer' button;

Rotator.prevStep(callback); -

Performs transition to the prev level. @param callback {function} callback to call after changing step;

Rotator.currentStepId(); -

Gets current step's index. @returns {number} id of a step;

Rotator.lastLoadedStepId(); -

Gets last loaded step's index. @returns {number} id of a last step;

Rotator.init(callback); -

Factory method for preloading first step. @param callback {function} callback to call after loading step.

# VALIDATOR PART

*Validator is a class for checking fields and forms in trainer. Current version of validator has 2 modes: strict and non-strict mode. Non-strict mode ignores amount of user's attempts and allows the user to go to the next step as soon as all fields are written correctly. In strict mode, Validator monitors the number of attempts and allows to go to next level if the number of attempts is 0 (in this case the number of points for this step decreases) or if everything was entered correctly.*

Validator.setStrictMode(b); -

Sets a strict mode for Validator. @param b {Boolean} strict mode switch. @returns {Validator} current object (flow);

Validator.setIgnoreCase(b); -

Switch for case sensitivity of validator. @param b {Boolean} sensitivity mode switch. @returns {Validator} current object (flow);

Validator.setAttempts(a); -

Sets an amount of attempts in strict mode, that user can use to write a correct answer. @param a {number} amount of attempts. @returns {Validator} current object (flow);

Validator.getAttempts(); -

Gets an amount of attempts in strict mode. @returns {Validator} current object (flow);

Validator.setAttemptsOnCheckButton(b); -

Sets an amount of attempts in strict mode, that user can use to write a correct answer. @param a {number} amount of attempts. @returns {Validator} current object (flow);

Validator.setPenalty(p); -

Sets penalty for each failed try in strict mode. @param p {number} penalty size. @returns {Validator} current object (flow);

Validator.getPenalty(); -

Gets penalty. @returns {Validator} current object (flow);

Validator.fixRadio(o); -

Fixes radio buttons adding value to them so validator can read them properly;

Validator.fixCheckbox(o,n); -

Fixes checkboxes adding value to them so validator can read them properly. Second param - whether you want the non-checked state of checkbox to be a correct answer;

Validator.disableAnswersBacklight(b); -

Disables backlighting of correct/uncorrect answers in the current step;

Validator.enableStepFinishAlert(b); -

Enables alert popup after success/fail to do a step;

Validator.addValidator(o, v, multicorrect, multiple); -

Adds an object to observe by the Validator. @param o {jQuery} wrapped DOM element where to get value to check. @param v {Array|\*} correct values of element's value. It can be an array of values or only one value. @param multiple {Boolean} check it, if you want validator to explode your element's value and check separately. @returns {Validator} current object (flow);

Validator.validate(); -

Method validates all Validator's observables;

setScoreOnPushResults(); -

Calculating score on step, where "Finish trainer" button clicked.

# Examples

|  |  |  |
| --- | --- | --- |
| ***Text input*** | | |
| **HTML** | | |
| {{{STEP\_INPUT}}} | | |
| **Script** | | |
| //validation part  .addValidator($(‘input[name=”step-input”]’), ‘[value]’),  where [value] – correct number or string  //mustache part  STEP\_INPUT: new TextInput(‘step-input’).render() | | |
| **Browser** | | |
|  | | |
| ***Select*** | | |
| **HTML** | | |
| {{{STEP\_SELECT}}} | | |
| **Script** | | |
| //validation part  .addValidator($(‘select[name=”step-select”]’), ‘[name3]’),  where [name3] – name of correct option  //mustache part  STEP\_SELECT: new Select(‘step-select’)  .addOption(‘Element 1’, ‘[name1]’)  .addOption(‘Element 2’, ‘[name2]’)  .addOption(‘Element 3 (right)’, ‘[name3]’)  .render() | | |
| **Browser** | | |
|  | | |
| ***Radio*** | | |
| **HTML** | | |
| {{{STEP\_RADIO}}} | | |
| **Script** | | |
| //validation part  .addValidator($(‘input[name=”step-radio”]’), ‘[name4]’),  where [name4] – name of correct radio option  //mustache part  STEP\_RADIO: new Radios(‘step-radio’)  .addRadio(‘Radio №1’, ‘[name1]’)  .addRadio(‘Radio №2’, ‘[name2]’)  .addRadio(‘Radio №3’, ‘[name3]’)  .addRadio(‘Radio №4 (correct)’, ‘[name4]’)  .randomize().render() | | |
| **Browser** | | |
|  | | |
| ***Checkbox*** | | |
| **HTML** | | |
| {{{STEP\_CHECKBOX1}}}  {{{STEP\_CHECKBOX2}}} | | |
| **Script** | | |
| //validation part  .addValidator($('input[name="step-checkbox1"]'), true)  .addValidator($('input[name="step-checkbox2"]'), false)  //mustache part  STEP\_CHECKBOX1: new CheckBox('step-checkbox1')  .setValue("ch1")  .setLabel('Mark me!')  .render(),  STEP\_CHECKBOX2: new CheckBox('step-checkbox2')  .setValue("ch2")  .setLabel(‘Don't mark me!')  .render() | | |
| **Browser** | | |
|  | | |
| ***Draggable Group*** | | |
| **HTML** | | |
| {{{STEP\_DRAGGABLES}}} | | |
| **Script** | | |
| //mustache part  STEP\_DRAGGABLES: new DraggableGroup('step-draggables').addClass('value')  .addOption('Answer 1', 'one')  .addOption('Answer 2', 'two')  .addOption('Answer 3', 'three')  .addOption('Answer 4', 'four')  .randomize().render(),  TEST\_INPUT1: new TextInput('test-input1').render() | | |
| **Browser** | | |
|  | | |
| ***Droppable Area*** | | |
| **HTML** | | |
| {{{STEP\_DROPPABLE}}} | | |
| **Script** | | |
| //validation part  .addValidator($('div.droppable[name="step-droppable"]'), ['one', 'four'], false, true)  //mustache part  STEP\_DROPPABLE: new DroppableArea('step-droppable')  .addClass('input')  .render() | | |
| **Browser** | | |
|  | | |
| ***Wolfram Alpha*** | | |
| **HTML** | | |
| {{{PLOT}}} | | |
| **Script** | | |
| //preDispatch part  var plot = null;  this.preDispatch = function (callback) {  var w = new WolframAlpha(); // Making a call to wolfram api to build a plot  w.setQuery('3x^2+2x+5').plot(function (data) {  plot = '<img src="' + data + '">';  callback();  });  };  //mustache part  PLOT: plot | | |
| **Browser** | | |
|  | | |
| ***LateX*** | | |
| **HTML** | | |
| {{{STEP\_LATEX}}} | | |
| **Script** | | |
| //preDispatch part  step\_latex = null;  step\_latex = new LateX();  step\_latex.setFormula("G\_{\\mu \\nu }=8\\pi G(T\_{\\mu \\nu}+\\rho \_{\\Lambda }g\_{\\mu \\nu})");  //mustache part  STEP\_LATEX: step\_latex != null ? step\_latex.render() : "" | | |
| **Browser** | | |
|  | | |
| ***Text*** | | |
| **HTML** | | |
| {{STEP\_TEXT1}} | | |
| **Mustache** (*langs/[lang].json*) | | |
| “STEP\_TEXT1”: “Lorem ipsum dolor sit amet..”, | | |
| **Browser** | | |
|  | | |
| **HTML** | | |
| {{{STEP\_TEXT2}}} | | |
| **Mustache** (*langs/[lang].json*) | | |
| "STEP\_TEXT2": "<b>Lorem</b> <i>ipsum</i> <u>dolor</u> sit <sup>amet</sup>..", | | |
| **Browser** | | |
|  | | |
| **Script** | | |
| //frequently  ‘{{STEP\_TEXT}}’ / ‘{{{STEP\_TEXT}}}’  //in some cases  I18N.getConstants()[‘STEP\_TEXT’]; | | |
|  |  |  |

# Основні рекомендації

Вміст папки з тренажером (при завантаженні до системи папку cdn включати до архіву непотрібно. Архів має бути zip):

Изображение выглядит как стол

Автоматически созданное описание

cdn – локальна версія ядра, яке лежить на сервері. ЗМІНИ СЮДИ ВНОСИТИ НЕ МОЖНА!

css – містить файл стилів для поточного тренажера

fonts – набір шрифтів

img – папка для зображень, які можуть бути використані у тренажері

langs – містить файли для тексту тренажера різними мовами

trainer – містить html-сторінки для кожного кроку, скрипти та файли налаштувань

index.html – тут відкривається тренажер у iframe

report.html – для зв’язку із сервером

trainer.html – тут підключаються усі скрипти, файли стилів, розмітка статичних кнопок

Папка **trainer**:

Introduction.html – сторінка вступу. Містить інформацію про назву тренажера, кількість кроків, можливість вибору мови

results.html – сторінка результатів. Відображає кількість набраних відсотків зі 100 та час виконання

step1.html … stepN.html – розмітка кроків

папка scripts – відповідні скрипти до вступу, кроків та результатів

папка settings – налаштування

trainer.config.json – містить інформацію про мови, які використовуються, мову запуску, максимальну кількість відсотків за тренажер (завжди 100), розробник та автор курсу, статус (тестування або готовий до завантаження у систему);

trainer.steps.json – містить інформацію про розподіл відсотків за виконання за кожен крок (вступ та результати зазвичай не оцінюються).

У скриптах мають бути присутні функції this.preDispatch, this.postDispatch та this.mustache. Перша й остання можуть бути порожніми. Основна взаємодія прописується у this.postDispatch.

Якщо перевірка виконаного відбувається НЕ лише стандартними функціями тренажера (введення тексту, вибір з переліку, радіо або чекбокси), то треба прописати, що має відбутися, якщо користувач на цьому кроці захоче завершити тренажер. У trainer.html для кнопки модального вікна, що з’являється після натиснення «Завершити сеанс» треба прибрати функцію onclick, щоб вона не спрацювала, коли не треба. Цій кнопці для зручності можна призначити id. У introduction.js можна прописати наступне:

$('div#endModal button#end\_end').click(function(){

var stepId = Rotator.currentStepId();

//визначаємо, на якому кроці натиснуто кнопку «Завершити сеанс»

switch (stepId) {

case 1:

//опис функції перевірки на кроці 1

break;

case 2:

//опис функції перевірки на кроці 2

break;

}

Scorer.addScore(total); //якщо треба зарахувати бали за крок, замість total підставити потрібне

$('button.check').addClass('disabled').unbind('click'); //зробити кнопку перевірки неактивною

$('div#endModal button#closeButton').click(); //імітація натискання кнопки, що закриває модальне вікно

Rotator.nextResults(); //автоматичний перехід до кроку із результатами

});

Стандартна функція перевірки .validate() сама визначає, скільки балів дається за кожне поле на кроці (береться кількість балів за крок і ділиться на кількість елементів, для яких призначений валідатор). Але якщо потрібно за якісь поля давати більше балів, тоді краще за крок у файлі встановити 0, а у функції перевірки нараховувати те, що треба ( Scorer.addScore(score) ).

Щоб мінімізувати можливість нарахування додаткових помилкових балів за крок, можна після успішного виконання робити кнопку перевірки неактивною ($(‘.page1 button.check’).addClass(‘disabled’).attr(‘disabled’, ‘disabled’);). Так само можна хитрувати і з кнопкою «Завершити сеанс»: після успішного виконання кроку (де складна перевірка елементів) блокувати її, а після переходу до наступного кроку (де звичайна перевірка) – розблоковувати.

Тренажер такого типу передбачає лінійне виконання: всі кроки проходяться послідовно. Тому, буває, коли крок виконаний і стала доступною кнопка «Наступний крок», то потрібно перейти до наступного, щоб завантажився крок. Наприклад, якщо користувач виконав крок 2, активувалася кнопка «Наступний крок» і він перейшов до кроку 3, то звідси повертатися на попередні кроки не проблема. Але якщо користувач виконав крок 2, активувалася кнопка, а він повернувся до кроку 1, то при поверненні на крок 2 кнопка «Наступний крок» буде вже неактивною. Тут також можна перестрахуватися і заблокувати кнопку «Попередній крок», якщо крок виконаний. Щоб користувач міг йти тільки вперед (але після переходу не забути розблокувати кнопку). (Функції Rotator.disableNextButton() / Rotator.enableNextButton() див. вище у таблицях.

У локальній версії такого немає, але коли тренажер вже завантажений на сайт, то на кроці результатів кнопка «Завершити сеанс» неактивна, щоб користувачі на неї не натискали. Оскільки при переході на крок результатів вже відбувся обмін даними із сервером і робота з тренажером вважається завершеною. Достатньо просто закрити сторінку.

Для запуску локально я використовую вебсервер Open Server, але ви можете користуватися тим, який знаєте.