

Create (CGS) Front End - New Team Member Guide

Asaf Shochet

Created: May 2015

Last modified: May 2015

Table of Contents

[1 Intro 4](#_Toc420848459)

[2 Architecture overview 5](#_Toc420848460)

[2.1 Client-Server communication 5](#_Toc420848461)

[2.2 Data handling in the browser 6](#_Toc420848462)

[2.3 Data flow example – opening a course 7](#_Toc420848463)

[2.4 Project File Structure 7](#_Toc420848464)

[2.5 Managing dependencies 10](#_Toc420848465)

[2.5.1 External libs 10](#_Toc420848466)

[2.5.2 Declaring dependencies 10](#_Toc420848467)

[2.5.3 CSS dependencies 10](#_Toc420848468)

[3 CGS main components 12](#_Toc420848469)

[3.1 Author\Editor vs. admin mode 12](#_Toc420848470)

[3.2 Basic screen components 12](#_Toc420848471)

[3.2.1 General – type configuration 12](#_Toc420848472)

[3.2.2 Course screen 13](#_Toc420848473)

[3.2.3 Lesson screen 14](#_Toc420848474)

[3.2.4 Sequence screen 15](#_Toc420848475)

[3.2.5 Task 15](#_Toc420848476)

[3.3 Dialog screen 16](#_Toc420848477)

[3.4 Dialog view (not dialog screen) 17](#_Toc420848478)

[3.5 Communication with server 19](#_Toc420848479)

[3.6 Handling files 19](#_Toc420848480)

[3.7 Browser URL manipulation 19](#_Toc420848481)

[3.8 Customization pack 20](#_Toc420848482)

[3.9 Data Conversions 21](#_Toc420848483)

[3.10 Conversion to DL’s XML format 22](#_Toc420848484)

[3.11 Validation Utils 22](#_Toc420848485)

[3.12 Publish flow 23](#_Toc420848486)

[3.13 Polling mechanism 23](#_Toc420848487)

[3.14 Logging 24](#_Toc420848488)

[3.15 Translations support 25](#_Toc420848489)

[3.16 Mathfield 26](#_Toc420848490)

[3.17 Integration with DI plugins 27](#_Toc420848491)

[3.18 lock/transaction mechanism 27](#_Toc420848492)

[4 Exercises and practice 27](#_Toc420848493)

[5 Javascript+HTML+CSS resources 27](#_Toc420848494)

[5.1 JS tutorials: 27](#_Toc420848495)

[5.2 JQuery tutorial: 28](#_Toc420848496)

[5.3 Css3 tutorial: 28](#_Toc420848497)

[5.4 Client performance: 28](#_Toc420848498)

# Intro

The following document is designed to give you a basic introduction to the client-side code of the CGS (AKA Create). Reading it alone will not make you an expert, but hopefully will give you a pretty solid place to start and a basis for asking good questions.

The best practice for this guide is to read it while playing around the parts you read about.

If you read about the course screen – open it and debug it, play with its CSS.

Reading about the publish process? Open developer-tools (Hit F12) and see how the polling mechanism works and how can you set its interval.

Diving into the open course process? Debug and see when its applets are downloaded.

**P.S.**

Please update this document with your own thoughts and add forgotten areas.

Welcome ☺

# Architecture overview

## Client-Server communication

Server communication layer

restDictionary.js

dao.js

rest.js

**REST API**

Some usages:

- Save\open course, lesson, etc.

- Locks mechanism

- Permissions

- Publish, export courses

- get\set account & user settings

Client

Logic

rest

Server

­­­

## Data handling in the browser

Browser file system

All the files used by a course\lesson that the user opened in his browser are saved in local file system (including – media, customization pack, manifests, converted data for DL, applets and more).

Supported only in chrome.

[Read more..](http://www.html5rocks.com/en/tutorials/file/filesystem/)

Browser DB

Used for the logging mechanism, keep client logs on a persistent layer before sent to server

­­­­­Client

Java script code

Repo.js

Lives in JS memory (deletes on refresh)

Gets its initial data from the server (i.e. when opening a lesson).

Saves the state of the current lesson\course\sequence\task that the user is working on during his session. When the user hits “save”, the data from repo is transformed into the server format and then sent to server to be saved.

## Data flow example – opening a course

This is an example trying to explain the flow for working with the dialogs (modals) + getting data from server + working with a data model + updating repo.

Opening a course:

A user enters to CGS and the open course menu is shown.

The code opens the dialog is in cgsUtils, which set the callback for the dialog’s chosen course response to be a method called onOpenCourseChosen.

var dialog = require('dialogs').create('openCourse', dialogConfig, 'onOpenCourseChosen');

Now, the *openCourseDialog* (held in modules/dialogs/openCourse) is shown. When a user chooses a course and hits “open” – the dialog is closed, and the onDialogClose method is called.

In this case – the caller for the dialog sets the onOpenCourseChosen callback to be a function that downloads the course and opens it.

The id of the course that the user wants is now passed as a parameter to the courseModel.openCoursemethod.

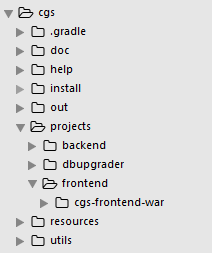
*courseModel.openCourse* uses the files.jsto download the course assets from server and then updates the course manifest in the repo, adding the course itself as an entity in the repo, where all its attributes are in in repo.

Afterwards, the course manifest is updated in the repo, using repo.js, adding the course itself as an entity in the repo, where all its attributes are in in repo.

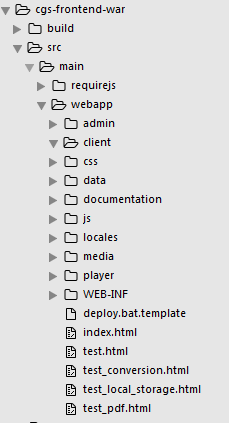
## Project File Structure

The CGS has 3 sub-projects: frontend, backend, dbupgrader.

The frontend source code is here:



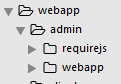
Inside it, there are these folders:



requirejs/ - holds the RequireJS files – this is [RequireJS](http://requirejs.org/). We are not sure why this folder is in not in js/libs

Webapp/ - holds the CGS application

Webapp/admin: holds the admin application (read more about it in section about CGS roles).



Css/: holds css files + favicon (little icon shown in chrome) + sprite images for button icons + other images.

Data/ a folder containing data for tests, the test that uses them is in: test\_local\_storage.html.

These tests are not running ever – need to be deleted!!

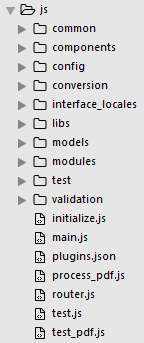
Documentation/: documentation regarding the integration with DI plugins. Available on: <http://localhost/cgs/client/documentation/out/index.html>

Locales/: zip files of default customization packs.

Media/: more media files, fonts and icons.

Player/: holds both SCP and DL players, used for preview. This folder is not in SVN, but copied here when running the gradle (build) process.

Js/: all javascript files used by CGS including external libs.



Js/common/: contains general functionality that is used across the system, such as file handling (files.js, assets.js), events management (events.js), clipboard management (clipboard.js).

Also, it contains some of the base classes used across the system such as: BaseStageView.js, BaseScreen.js.

Js/components/: contains elements that appear in multiple places in CGS.

Js/config/ - holds the types.js file and more configuration files.

Js/conversion/: holds the conversion templates between the data format in CGS (JSON) to the one used by the DL player (XML). More info in this document under “[Conversion to DL’s XML](#_Conversion_to_DL’s)”.

Js/interface\_locales/ : contains the language translations for the CGS buttons, popups, messages, etc. Each locale has a folder: en\_US/, fr\_FR/, pt\_BR/ and translations JS file inside.

Js/lib: contains external libraries such as cropperJS (for image manipulation), html2canvas (for thumbnails creation), zipjs (for extracting zip files from server) and more. Challenge! If you have some time - try to find a lib that is not used and remove it! (less code – less bugs).

Js/models: keeps the data models that are used in the repo.

Js/modules: all editors, screens, and many other components.

Js/test: tests folder. Not sure if It is used.

Js/validation: classes related to repo validation process. Read more on[: Validation Utils](#_Validation_Utils) part.

## Managing dependencies

### External libs

All external libs are held in folder:

js/libs

Some of the main dependencies are: JQuery, Lodash, Backbone & Mustache.

### Declaring dependencies

RequireJS is used for dependency management. The dependencies for a file are stated at the beginning of it using the requireJS syntax. Example:

define(['lodash', 'cryptojs', 'cryptojs\_sha1', 'canvas\_blob', 'browserDb'],

function(\_, CryptoJS, cryptojs\_sha1, canvas\_blob, browserDb) {

The linking between the names to the file that we depend upon is held on: main.js

### CSS dependencies

The css dependencies are managed in file initialize.js, in it – each CGS role (admin, author, editor) holds a list of CSS files that it uses.

So, if you want to add a css, add it to the list on initialize.js*.*

A note about css:

If you want to use an image, for example to use sprite icons, it is important to add the CGS version as a variable like this:

background-image: url('../media/CGS\_icons.png**?ver=30**');

In that way we will avoid caching problems if we would change this file in future.

# CGS main components

## Author\Editor vs. admin mode

There are several roles in the CGS:

Super admin: a user that can create accounts.

Account Admin – a user that can add users to a specific account and change that account’s settings.

Editor – can create courses and modify them.

Author – can see content, but cannot edit it.

Code wise, the admin(s) page and the author\editor’s pages our completely different. The code for the admin is under: webapp\admin\webapp

While the code for author\editor is under webapp\.

Note that 99% of the CGS code deals with editor mode.

And yes, there is a lot of code duplication between admin webapp and editor webapp that needs to be removed.

## Basic screen components

### General – type configuration

Every type that exists in the repo is mapped in types.js, which indicates what is its controller and screen.

For example:

For an item with type “course”, these are the configurations:

course: {

screen: 'CourseScreen',

editor: 'CourseEditor',

group: 'course',

showpreview: false,

clipboard: {

canCopy: false,

insert: 'last',

childrenTypes: ["toc", "lesson"]

}

}

Let’s go over the properties:

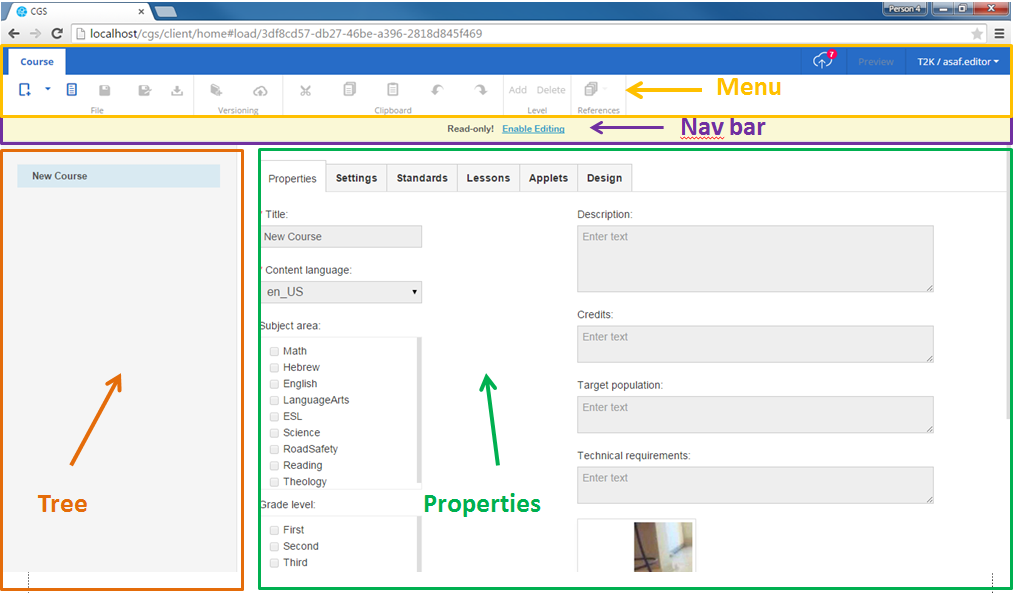
screen: this is the screen opened when viewing this type of item: CourseScreen.js

editor: this is the editor used to edit a course: CourseEditor.js

showpreview: a Boolean variable indicating whether the preview button  will be enabled or disabled. When enabled, the preview buttons opens a modal of the DL player and shows the data as it will be shown when played to the students.

Clipboard: defining the behavior of the copy-paste actions (also enable\disable it)

### Course screen



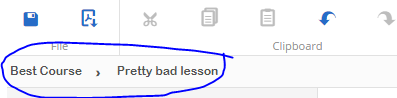
The course screen is composed of base components, each one has it’s own html template and behaves as a separate component. All of them are used also in other screens. For example – the “properties” part takes a large part of the screen when viewing a course, but takes much less space when viewing a lesson.

**Menu** – shows all options available for the the course in the current state. If the course is in read-only mode, some of the options will be disabled.

Menu’s files are all related to: js/modules/menuComponent/MenuComponentView.js. The menu ribbon can be rendered in any place using its initialize method.

**Nav Bar** – shows indication about the lock state (enable\disable editing), and also data about publishes.

The navbar’s files are in js/modules/NavBarComponent folder.

The same file NavBarComponent.js is responsible for the lock messages, enable\disable and the breadcrumbs ( ) shown on lesson, seqeunce and task screens

**Tree** – shows the course structure, it’s TOCs, lesson, assessments, etc.

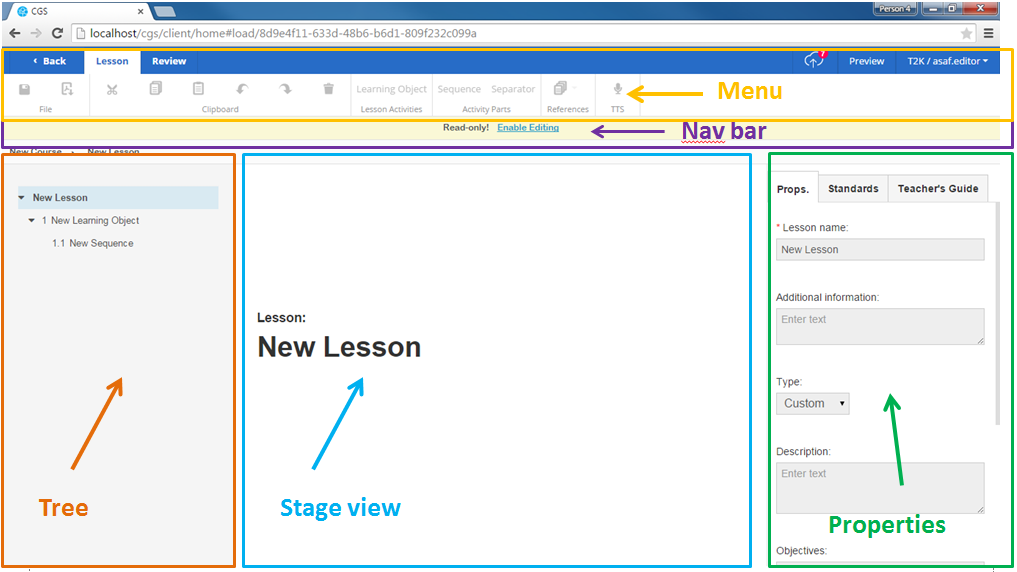
Its files are in: js/modules/TreeComponent folder.

**Properties** – shows editable properties regarding the course, its language, subject area, and also provides a way to edit its customization pack.

The base view JS file is: BasePropertiesView.js

Each type of item has a different properties view, for example – the sequence uses SequencePropsView that extends the base view: var SequencePropsView = BasePropertiesView.extend({

### Lesson screen



Same as the course screen, but with another area called stage view.

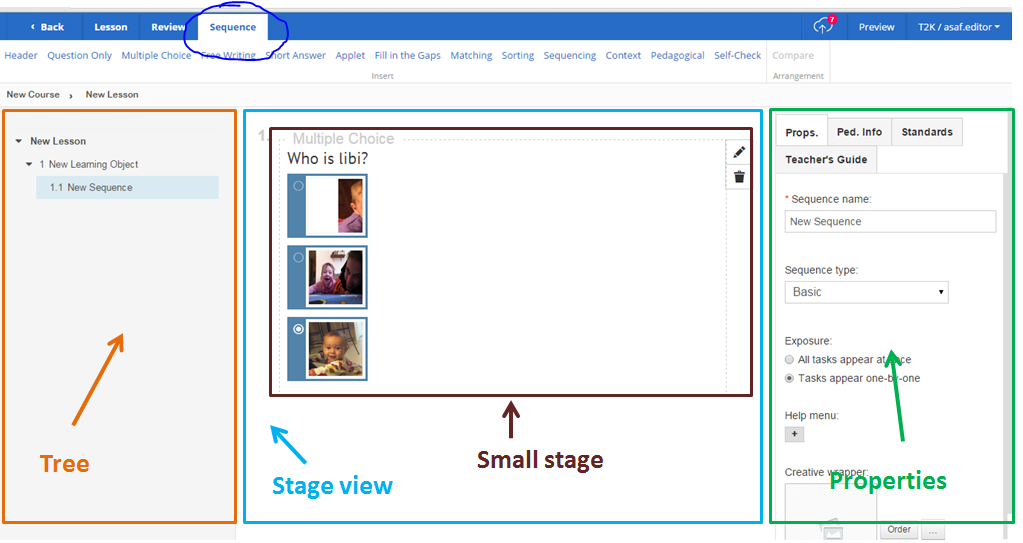
The properties part is exactly the same but smaller.

**Stage view** – shows previews of the lessons content.

A word about that: stage view of the lesson extends the base stage view BaseStageView.js:

var LessonStageView = BaseStageView.extend({

### Sequence screen



This is the sequence window. As before – it uses the same base components that **extend** the base classes.

According to what is written on types.js, the sequence configuration is:

screen: 'LessonScreen',

editor: 'SequenceEditor',

group: 'sequence',

fullName: "Sequence",

validationBubbleUp: false,

showpreview: true,

Meaning, the base screen layout is like the one of the LessonScreen, but the editor is the SequenceEditor

Properties view: SequencePropsView (extends BasePropertiesView)

The stage view contains small previews of tasks inside the sequence: this is called a “small stage”. Each task has a small stage preview. For example: McSmallStageView.js

### Task

Each task contains a templates folder, config json and a JS file.

To edit the task, we use an entity called TaskEditor, the base entity is BaseTaskEditor.js, and every task has a different editor that extends it. For example: AppletTaskEditor.js handles editing of applet task.

Different types have different models and views, most of them are in separate folders.

For example, all files regarding the Free Writing task are under: Js/modules/FreeWritingEditor

The file structure is:

templates/: a folder containing the html template(s).

Config.js – configuration file (optional).

Editor.js – the logic for editing this task.

Stageview.js – describes the way the editor looks on the stage view (big window).

Smallstageview.js – describes the way the task looks and behaves when viewing it in small stage (a little preview-like look).

## Dialog screen

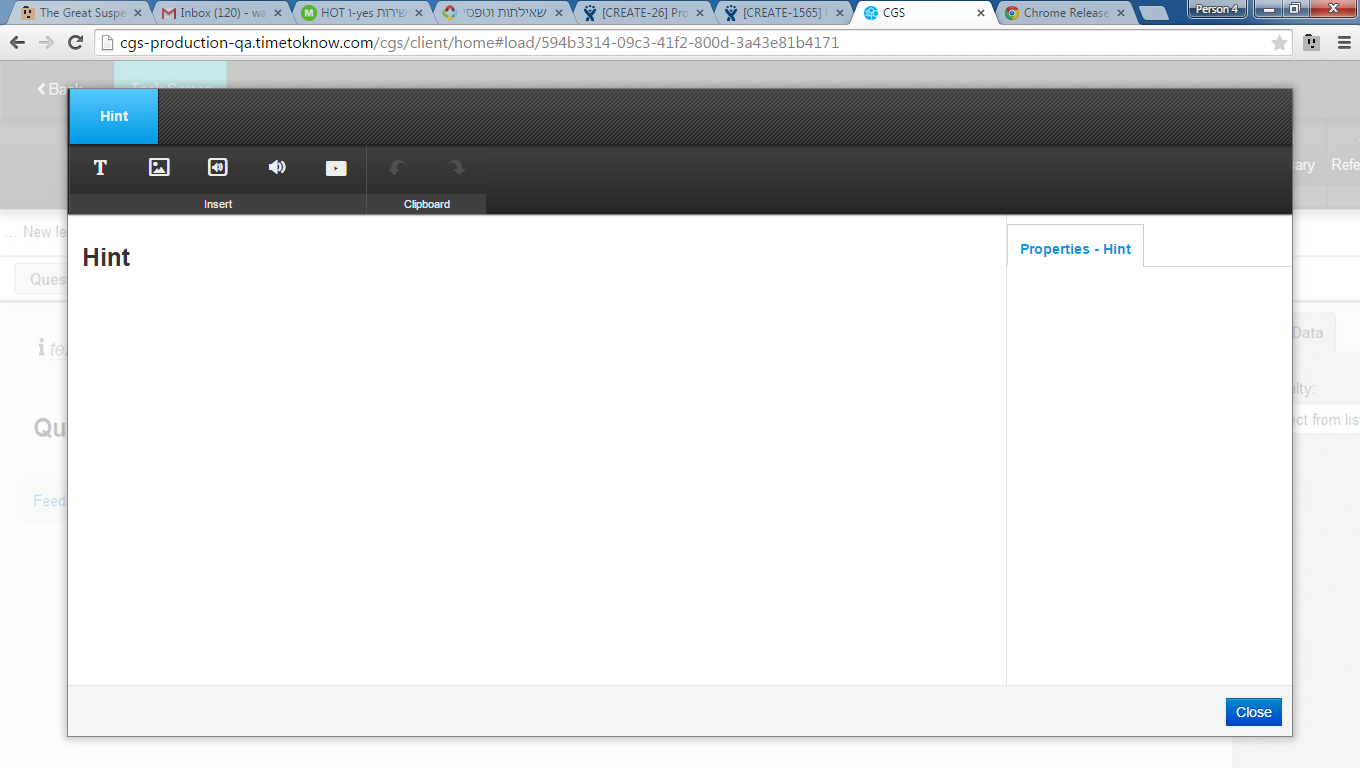
Dialog screen is a view that looks like the lesson screen, but looks a bit different.

Its base component is DialogScreen.js and there are many variations that extend it. The main thing to know about it is that it is exactly like the lesson editor screen (menu, stage view, properties, etc.) but it takes 80% of the screen.

For example:

The hint editor uses the dialog screen.

It looks like this:



How do we know? It is written in the type.js file:

hint: {

screen: **'DialogScreen'**,

editor: 'HintEditor',

group: 'hint',

fullName: "Hint",

validationBubbleUp:false,

showpreview: false,

clipboard: {

canCopy: false,

insert: 'last',

childrenTypes: []

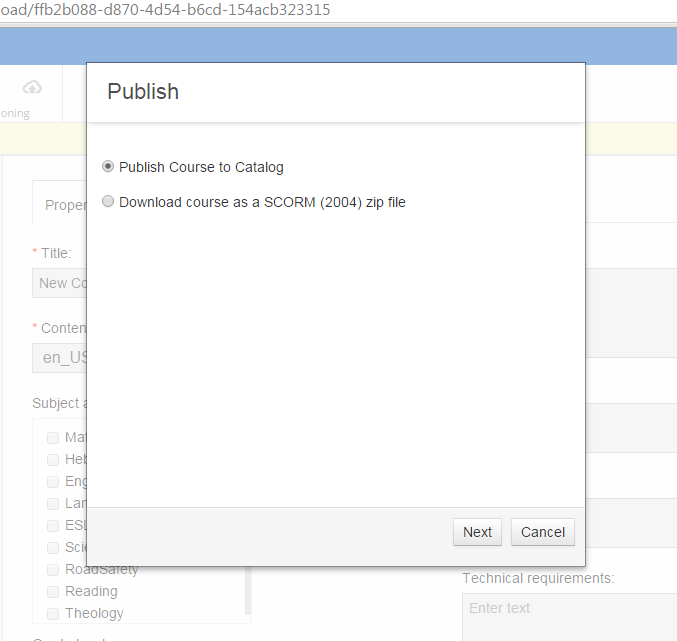
}

},

## Dialog view (not dialog screen)

(Like image cropper, applet dialog, file upload)

A dialog is a modal window shown as a layer over the current screen the user sees. Like this (publish dialog):



Each dialog uses the BaseDialog interface (extends BaseDialog.js) that provides API for rendering & passing callbacks to handle hitting OK\Cancel.

A dialog holds all its files under a separate folder (like: js/modules/BaseDialogScreen, js/modules/Dialogs/types/importCourse), which in most cases holds a template (html) and a js file to handle functionality.

The module that initializes the dialog can set the call back for dialogClose and handle its response (if exists).

## Communication with server

Used for many things such as getting data from DB, getting\sending images, updating publisher's account settings, publishing and more.

There are APIs to contact the server – they are in:

asset.js, dao.js – contain methods to send requests (mainly [XHR](http://bytes.com/topic/javascript/answers/699710-what-difference-between-xmlhttprequest-standard-http-request)s to server)

restDictionary.js – contains the mapping to all server paths. For example,

the path *SAVE\_COURSE* is mapped to url: /publishers/{{publisherId}}/courses/{{courseId}} using PUT method.

## Handling files

All the files related to CGS’ content are saved on the server as a persistency layer. More than that – when a user opens a course\lesson, all the data related to it is downloaded to the client’s file system. Currently, only Google Chrome supports the file system API.

Why do we do it?

Historically, the CGS was meant to be a system that can work in offline mode, so it makes sense to keep a copy of all files. But that created many problems, so currently we don’t support offline, but we do have a copy of the files.

Main components that deal with files (both upload\download and browser’s file system managing) are: asset.js, files.js

## Browser URL manipulation

The CGS is a single page application, so it is possible that the URL will be the same no matter what the user does. But, happily, the URL is changes when the user enters a lesson, course, sequence, task, and navigates to settings, narration or other tabs.

The management of all URLs (beside the login URL, which is returned from server in JSP) is done by Backbone’s Router. You can read about it [here](http://backbonetutorials.com/what-is-a-router/). The file deals with that in CGS is: *router.js*

All URLs that are generated using Backbone’s router starts with #load/

For example:

http://localhost/CGS/client/home#load/f5e22a85-4394-482d-ab5b-a661bb0c7466

## Customization pack

Customization pack is a collection of rules regarding the appearance of the course. Every course can have a different customization:

* Fonts
* Custom strings used for feedbacks and instructions
* Styles and Effects: font color, size, highlight, etc.
* DL’s appearance – colors, strings, etc.

Styles and effects are also defined in the customization pack, for example: I can set a different font style – that is David + pink background + italic, and save it with a special name PinkLady. It will be saved in the customization pack, and later I can apply this style inside a text.

Technically – the customization pack is saved in a zip file in webapp/locales folder. Each locale has a zip: fr\_FR.zip, en\_US.zip, iw\_IL.zip. When a new course is created, the relevant zip is unzipped, downloaded to client and then saved as one of the course resources.

Customization’s pack Structure:

* Cgs
  + Media (folder)- all img/sound relevant for the customization pack
  + cgsStyles.css – css file with css rules regarding the cgs. ( because of an historical mistake there are un needed css rules there, they should be moved to the general cgs css files)
  + config.json – file that contains all the customization related to the CREATE, such as strings to display and styles to add the text viewer menu.
* Defaults- contain the default values for the pack before the user has made any change.
  + StringsDefaults.js – strings in the DL player
  + stylesDefaults.json – default fonts/ styles/ effects
* overrides(do not appear in the zip, but added after a change is made in pack of course) – contain the changes that the user saved over the default values
  + config.json – overrides stuff that was saved in cgs/config.json
  + stylesOverride.json – overrides stuff saved in stylesDefaults.json
* DL
  + Fonts – the font files related to the pack + uploaded fonts.
  + Mathfield- relevant font for mathfield
  + SystemFonts- fonts related to DL appearance
  + Config.json- contains the api pack version- generally do not need to change unless a major structure change is made in the pack.
  + Style.css – css rules regarding the DL (usually direction and font family)
  + System-font.css – DL fonts declarations
* Manifest.json- contains a list of all pack resources. The pack version and more.

**NOTE:**

1. The styles, effects and fonts are saved in a json format. When rendering a text viewer we generate from the json some css files that applies the rules described in the json. The DL is doing the same thing and converting the json to css rules.
2. When making a change in a customization zip you will have to change the pack version to be higher and also change the value of the version on the file : C:\t2kdev\cgs\projects\frontend\cgs-frontend-war\src\main\webapp\locales\versions

## Data Conversions

The raw data for each item is different. For example: courses have a title, maybe a cover image and also content language config, but a sequence have a title and doesn’t have a content language config.

Moreover, The data structure on client is different than the structure in server.

Also, remember that the client needs a way to access a task’s properties, text, attributes, etc. but the server doesn’t need that, so it is sent to server as a stringified JSON (hopefully will change in the future).

So, every time lesson\course\sequence\task data is sent\received to\from server it needs to be converted to the correct format.

These conversions are managed by a component called conversionUtil.js that uses schema.js file that maps the conversions from\to server for each data type.

conversionUtil provides an interface for these conversions using 3 methods:

conversionUtil.dataRemoteToRepo converts data coming from server into the format used by client’s repo (used after getting data from server).

conversionUtil.repoToDataRemote converts the data from client’s format into the server’s format (used before sending data to server).

conversionUtil.repoToSCPPlayer converts the data from repo format to the SCP format (used to perform preview).

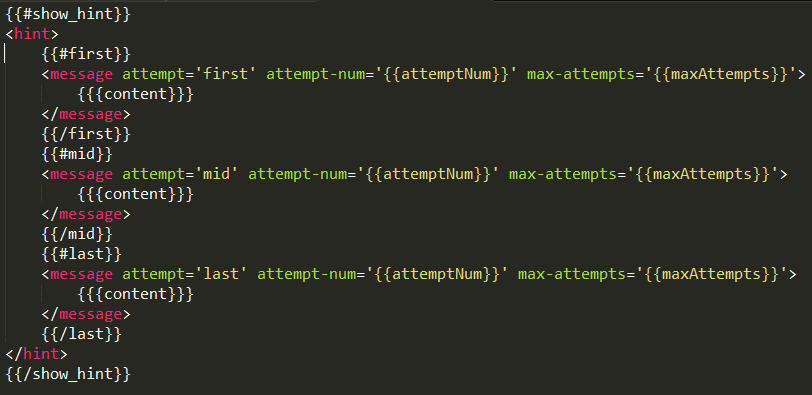
The conversion itself is configured in the schema.js file.

When do we touch this file?

When a new property is added to a lesson, sequence, etc. we need to know how to convert it from the representations in client to server and vice-versa. This will be the place to update.

## Conversion to DL’s XML format

Tasks look different in DL and in CGS. The data structure used by CGS is a JSON, but in DL it is XML. Therefore – we need a way to transform data between the formats. There is a different conversion template for every task type and component. For example, the template for the hint type in js\conversion\templates\hint.xml looks like this:



If the {{}} syntax looks familiar it’s because we use the MustacheJS library for templates, and the way the conversion utils work is by getting a template and a data object to render it to.

The mapping between the entity and its template is in templates.js:

define( […, “text!conversion/templates/hint.xml", ..],

function(…,“hint”,…){

…

'hint': {

template:hint

},

## Validation Utils

repoValidation.js is a file containing the logic for validating the correctness of course & lesson’s structure and its children (sequence, etc.). Every time a lesson is opened then we run the validation function to tell if it is valid or not. Technically, when you open a lesson it is downloaded from server, then it is getting converted and saved to repo, and then the validation starts.

If the validation fails then an error popup is shown.

A call example from lessonModel.js

//validate lesson

var lessonvalidation = repoValidation.validate(id);

Another use for the validation process is to allow model update to the current data format by client.

For example: When a new type of task was added to CGS, it had no interaction\_type property. We used the repoValidation util to add it every time this kind of task was opened by a user.

matchingAnswer: function (thisRepo, validateObj) {

thisRepo.data.interaction\_type = 'drag\_and\_drop';

},

## Publish flow

The publish is a process where the user decides that he wants his course\lesson to be sent to LMS, so students and teachers could play it.

The user needs to choose whether he wants to: publish an entire course or a lesson, add release notes, download or send the course to another system (such as CGS, Blossom).

The zip creation is done by server, and the client-side’s responsibility is to send the configurations for the publish (what the user chose) and perform polling in order to keep track of the publish progress in real-time.

The process is handled by publishModel.js that of course has some html templates for the screens show to user where he chooses the publish target etc. All of them are under: js/Modules/Dialogs/types/publish/

Different publish targets reflect different actions that the user can perform, if a user publishes to a catalog then only a status is shown, but if he publishes to a zip file, then a download link will be shown when the file is ready. If you want to add a different kind of publish, it will be good to start in publishModel.js

## Polling mechanism

When performing long actions against the server, such as: publish, import course, export course, we want to show the user the progress of the action.

To do that we use a polling mechanism that asks the server for the status of a “job” that have a jobId:

In restDictionary.js (the place that holds the mapping to REST APIs) it looks like this:

'CHECK\_JOB\_PROGRESS' : {

path: '/utils/jobs/{{jobId}}',

method: 'GET',

sendAs: null

},

And a calling to this URL with a {{jobId}}, responds with a JSON describing the progress of each part of the job, and also a success\failed\cancelled flag.

The export course process uses this as follows:

var daoConfig = {

path:restDictionary.paths.CHECK\_JOB\_PROGRESS,

pathParams: {

jobId: jobId

}

};

dao.remote(daoConfig, function(jobState) { // this sends the REST, and sets the callback to handle response

if (jobState.status == 'COMPLETED') {

if (typeof callback == 'function') callback(jobState.refEntityId);

}

else {

if (jobState.status == 'FAILED' || (jobState.status ==

'STARTED' && ++this.runningJobs[jobId].counter > 100)) {

show\_course\_failed\_dialog();

… more code….

## Logging

The logger’s purpose is to collect data on errors, warnings, user behavior, and background processing, afterwards send the logs to the server.

Note that the frontend log’s time is sync with server’s time.

**Usage**

1. Examples:  
   logger.debug(logger.category.COURSE, 'Course assets uploaded successfully');  
   logger.error(logger.category.FILES, { message: "File upload failed: " + config.url, status: errorData.target.status, response: errorData.target.responseText });
2. “logger” is a global singleton and can be accessed from anywhere.
3. Logger methods are defined in logger config:  
   error/warning/debug/info/audit.
4. Logging methods all have identical signature, and receive these params:
   1. Category – options are defined in logger config
   2. String or a JSON object to be logged.

**Functionality**

1. Show logs in browser console.
2. Store logs in browser database, where they are kept even if browser shuts down.
3. When a log batch is ready, send it to the server.
4. A log batch is ready to be sent to server:
   1. Once per interval (configured in js/common/logger/config.js -> var flushInterval)
   2. When application first loads.
   3. When application unloads.
5. When logger loads it syncs time with the server.

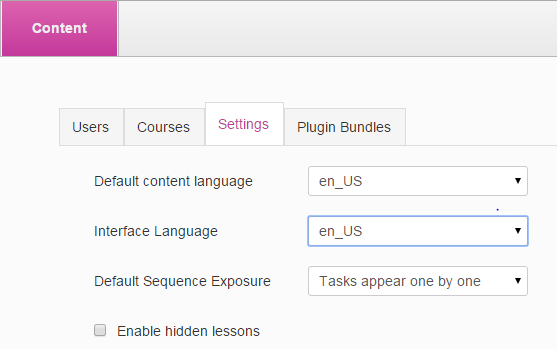
**Architecture**

1. Project path:  
   /js/common/logger
2. Folder contains 3 files:
   1. **Logger**  
      Loads log appenders and receives all log calls.
   2. **BrowserDbToAjaxAppender**  
      Stores logs in browser DB and Ajax’s them to server.
   3. **Config**  
      Logging configuration
3. We are using **log4javascript** library.
4. Logger is a singleton service that can load different appenders.  
   Currently is loads 2 log appenders, and more can be added in the future:
   1. BrowserDbToAjaxAppender.
   2. browserConsoleAppender.
5. Once appenders have been loaded by the logger, logging automatically activates each appender.

## Translations support

When a user opens a page that renders this template, translation will be taken from the interface locale JS file related to the user’s interface local in CGS.

The account admin can change the interface locale for his account by using the admin application:



Each locale has a folder: en\_US/, fr\_FR/, pt\_BR/ under Js/interface\_locales/ .

These folders contain JS files of a translation, in a key-value mode (JSON like).

Let’s focus on an example from pt\_BR locale. Have a look at these key-value pairs in pt\_BR.js:

"By Longest Answer": "Por Resposta Maior",

"Account Administrators:" :"Administradores de Conta:",

"Acronym": "Acrônimo",

We use these strings when showing an html which is a result of rendering a template by Mustache. The syntax is to wrap the key between double brackets: ((key)). Example of usage the above string in a link:

<li><a href="#T2KPublisherAccounts" data-toggle="tab">

((Account Administrators:))

</a></li>

After rendering it (assuming user’s account interface\_locale is pt\_BR) will show:

<li><a href="#T2KPublisherAccounts" data-toggle="tab">

Administradores de Conta:

</a></li>

Great, now you know how to write a template regardless of the interface language.

One more thing, the syntax of ((key)) is not a Mustach syntax, but of translate.js that looks for regular expressions and performs replacements.

## Mathfield

Create is getting its mathfield from the player project.

In order to integrate a newer version of mathfield inside the create you will run a power shell file located at: C:\t2kdev\player\projects\mathfield\Mathfield.ps1 (using right click -> run with powerShell).

This will generate 4 files in C:\t2kdev\player\projects\mathfield\build

These files should be copied to C:\t2kdev\cgs\projects\frontend\cgs-frontend-war\src\main\webapp\js\components\mathfield\internal

The problem begins when the mathfield has changes also in css, in this case in addition to the described above, you will also need to add the mathfield css in a style tag at the head of the text viewer iframe and update the paths of the fonts in the minified css.

Currently in the following files:

Search for ‘ initHTML += "<style>’

C:\t2kdev\cgs\projects\frontend\cgs-frontend-war\src\main\webapp\js\components\growing\_list\mathfieldGrowingListView.js

C:\t2kdev\cgs\projects\frontend\cgs-frontend-war\src\main\webapp\js\components\growingDouble\_list\growingDoubleListComponentView.js

C:\t2kdev\cgs\projects\frontend\cgs-frontend-war\src\main\webapp\js\modules\TextViewerEditor\TextViewerSmallStageView.js

C:\t2kdev\cgs\projects\frontend\cgs-frontend-war\src\main\webapp\js\modules\TextViewerEditor\TextViewerStageView.js

## Integration with DI plugins

The CGS provides an option to add plugins into it.

The plugin upload is by uploading a zip file in the admin screen.

These plugins can take control over the CGS’ buttons, busy indicator, and can create new task types, and expend the CGS to be more specific for the DI’s needs. It’s worth talking to one of the DI team members to know more.

## lock/transaction mechanism

# Exercises and practice

1. Fix a bug regarding a popup message’s text.
2. Add a log message every time the delete button is pressed for standards.
3. Draw a chart of everything going on when opening a lesson.
4. Ask about Text To Speech (TTS ) services, and find out where the client puts the parameters when it send a TTS request to 3rd party. Add a paragraph about it to this document.
5. Brain teaser #1 - Think of an alternative to using the browser’s file system.
6. Add a paragraph to this document regarding the thumbnails creation process
7. Look for the CGS about -> help. Make it open a new tab instead of a modal.
8. Brain teaser #2 – there are some JS files that are not used in the CGS, find 3 of them and delete them (please consult before commit ☺)
9. External libraries – some of our worst bugs were caused because of external libraries. What is the library that we depend upon the most? Check if we have the most updated version of it.

# Javascript+HTML+CSS resources

## JS tutorials:

<http://ejohn.org/apps/learn>

<http://stackoverflow.com/a/336868>

<http://stackoverflow.com/questions/864516/what-is-javascript-garbage-collection>

<http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/>

## JQuery tutorial:

<http://try.jquery.com/>

## Css3 tutorial:

<http://code.tutsplus.com/tutorials/30-css-best-practices-for-beginners--net-6741>

<http://www.htmldog.com/guides/css/beginner/>

<https://css-tricks.com/snippets/css/a-guide-to-flexbox/>

## Client performance:

<http://apmblog.dynatrace.com/2010/08/25/top-10-client-side-performance-problems-in-web-2-0/>