Documentation

Table des matières

Description de l'application

L'application permet de gérer une base de donnée de questions afin de générer facilement des questionnaires à choix multiples au format LaTeX.

Elle s'utilise principalement en ligne de commande. Un interface graphique a été développée par l'équipe précédente, mais elle est incomplète.

Voici les fonctionnalités de l'application :

Import : importe et analyse des fichiers LaTeX afin d'en extraire des questions vers la base de données.

Export : génère un fichier LaTeX à partir de questions choisies dans la base de données.

Tag : applique des tags à certaines questions afin de pouvoir les associer à une catégorie.

Recherche : cherche et affiche les questions dans la base selon un critère donné.

Modélisation

Côté utilisateur

Cette partie de la documentation regroupe les commandes de l'utilisateur.

Note : toutes les commandes décrites ici doivent être précédées de GQCM pour fonctionner (exemple : `GQCM import fichier.tex`).

Import : Permet d'importer les questions contenues dans un fichier .tex dans la base de donnée du logiciel. On a plusieurs cas d'utilisation :

« GQCM import Cheminversfichier/fichier.tex »
Les questions sont transformées en objet Question utilisables par le système et stockées dans une sélection temporaire, l'utilisateur peut ensuite choisir celles qu'il désire sauvegarder tel qu'indiqué par la commande.

« GQCM import »
« Cheminversfichier/fichier.tex »
Même principe mais entrée de la commande en plusieurs lignes.

Show :

« GQCM show »
Affiche toutes les questions sauvegardées dans la base de donnée, sous le format :

```
id : Nom {TYPE} [tags]
Début de l'énoncé...
```

Search :

« GQCM search »
« critèreDeRecherche »
« critèreA critèreB ... »

Affiche les questions de la base de données correspondant aux critères données (nom, tags ou id), puis permet d'effectuer certaines opérations sur les questions affichées. Une utilisation en une seule ligne se fait de la sorte :

« GQCM search name Question1 Question2 »
« tag tagA tagB »
On cherche ici les questions de nom `Question1` et `Question2`, et une fois trouvées on décide de leur appliquer les tags `tagA` et `tagB`.

Export :

« GQCM export »
« critère »
« nomCritère »
« quit »
« finish »
« fichierDestination »

Permet d'exporter des questions dans un fichier destination.

On choisit un critère d'export (`tag`, `nom` ou `id`) puis on entre le critère (exemple : on tape `nom` puis « `nomQuestion` »), on tape ensuite `quit` puis on peut choisir de sélectionner plus de question de la même manière ou passer directement à l'export en tapant `finish`. On tape ensuite un nom de fichier (exemple `fichierExport`) qui doit être différent de « », ensuite les questions sélectionnées seront sauvegardées dans le fichier fichierExport.tex.

Côté développeur

Cette partie de la documentation explique les différents modules utilisés par le système, et donne le rôle des différentes fonctions.
Les fonctions simples (getters, setters, etc.) sont volontairement omises par soucis de clareté.

Note : La plupart des fonctions expliquées ici sont également expliquées directement en commentaire dans le fichier DB.py

QCM :

Ce module défini les classes **`Question`** et **`Reponse`** qui servent à représenter les questions à choix multiple qui seront stockées dans la base de données.

`class TypeQCM(Enum):` Cette classe énumération représente le type de question (à réponses multiple ou non) sans utiliser une chaîne de caractère.

`type_from_str(string):` Permet d'obtenir l'objet **`TypeQCM`** à partir d'une chaîne de caractère.

`str_from_type(type_qcm):` Permet d'obtenir la chaîne de caractère (format LaTeX) associée à l'objet **`TypeQCM`**.

`moodle_from_type(type_qcm):` Permet d'obtenir la chaîne de caractère (format LaTeX Moodle) associée à l'objet **`TypeQCM`**.

`class Reponse:` Cette classe représente la réponse à une question.

`self.est_correcte :`	indique si la réponse est bonne ou mauvaise
`self.enonce :`	l'énoncé de la réponse.
`to_latex(self):`	retourne un string au format LaTeX représentant la réponse
`to_moodle_latex(self):`	retourne un string au format LaTeX Moodle représentant la réponse

`class Question:` Cette classe représente une question à choix multiples.

`self.type:`	le type (réponses multiples ou non) de la question
`self.nom:`	le nom (court, ce n'est pas l'énoncé)
`self.amc_options:`	options de Auto Multiple Choice
`self.enonce:`	l'énoncé complet
`self.reponses:`	la liste des réponses

`self.tags:` la liste des tags
`self.numberColumn:` le nombre de colonnes
`self.id:` l'id de la question. Il est unique dans la base de données, et n'est défini que lorsque la question est bien insérée dans la base.
`short_str(self):` renvoie une string décrivant la question. Elle est au format :

```
id : Nom {TYPE} [tags]
Début de l'énoncé…
```

`to_dict(self, index):` renvoie un dictionnaire représentant l'objet Question. Chaque attribut correspond correspond à une clé. C'est ce dictionnaire qui est inséré dans la base de données.
`to_latex(self):` renvoie une string contenant le code LaTeX représentant la question
`to_moodle_latex(self):` renvoie une string contenant le code LaTeX Moodle représentant la question

DB :

Ce module implémente une base de donnée stockant des objets provenant du module QCM. Cette base de donnée est sauvegardée dans un fichier au format json en tant que dictionnaire.

`tag_check(tags, required)` : retourne vrai si **`tags`** contient tous les tags contenus dans **`required`**, faux sinon. (**`tags`** et **`required`** sont des listes de variables de type **`string`**)

`keyword_check(text, keywords) :` retourne vrai si **`text`** contient tous les mots contenus dans **`keywords`**, faux sinon.

`reponse_from_dict(rdict) :` retourne une variable de type **`QCM.Reponse`** crée à partir du dictionnaire **`rdict`** passé en paramètre.

`question_from_dict(qdict):`retourne une variable de type **`QCM.Question`** crée à partir du dictionnaire **`qdict`** passé en paramètre.

`class Base:` Il s'agit de la classe utilisée pour représenter la base de donnée dans le système. Elle crée un dictionnaire python à partir de la base de données au format JSON.

`__init__(self, input_file):` stocke input_file dans `self.filename`, puis stocke les données récupérées dans le fichier ouvert via `input_file` dans le dictionnaire `self.data`.

`self.nextindex :` l'indice du prochain élément à insérer dans la base

`persist(self):` Sauvegarde data dans le fichier ouvert via `filename`. Tant que `persist` n'est pas appelée, les changement ont lieu dans le dictionnaire python mais pas dans le fichier JSON.

`add_question(self, question):` Ajout de l'objet `QCM.Question` à `data` après conversion au format dictionnaire.

`add_multiple(self, questions):` Appelle `add_question` pour chaque `QCM.Question` contenue dans la liste `questions`

`del_question(self, index):` Supprime la question d'`index` correspondant de `data`

`get_question(self, index):` Retourne la question d'`index` correspondant de `data`

`update_question(self, index, question):` Remplace la question d'`index` correspondant dans `data` par la nouvelle `question`, ou la crée si aucune question d'`index` correspondant n'existe

`question_by_*(self, *):` Retourne la liste des questions correspondant au critère `*` (nom, id ou tag) de la base de donnée, après conversion en objet `QCM.Question.`

`all_questions(self):` Retourne une liste de toutes les questions contenues dans data, après conversion en objet `QCM.Question.`

Parser:

Ce module permet de parser les fichiers .tex et d'en obtenir les questions, c'est à dire de construire des objets de classe `Question` (qui ont des attributs de classe `Reponse`) à partir d'un fichier .tex.

Les différentes fonctions décomposent le fichier en lignes et utilises les différentes marques de latex pour reconnaître les éléments lus.

`get_block(text, block_open_char, block_close_char):`
Cherche et renvoie dans `text` le premier bloc de texte compris entre les caractères `block_open_char` et `block_close_char`. Par exemple, `get_block("test {content {sub} content}", '{', '}')` renvoie `'content {sub} content'`. Les caractères ouvrant et fermant sont omis.

`pattern_at(text, index, pattern):` Renvoie un booléen indiquant si le motif `pattern` commence à l'index `index` dans la chaîne de caractères `text`.

`parse_latex(latex):` Lit le fichier `latex` ligne par ligne et fait appel à `parse_qcm` pour créer des objets `QCM.Question` à partir des données récupérées. La fonction détecte les balises `\\begin{response}` et `\\end{response}` et fait appel à `parse_qcm` pour créer un objet `QCM.question` à partir des lignes (`q_lines`) comprises entre ces balises.

`parse_qcm(q_lines, nb_questions):` Crée un objet `QCM.Question` à partir des lignes `q_lines` comprises entre `\\begin{response}` et `\\end{response}`. De manière similaire, la fonction fait appel à `parse_reponses` quand elle détecte des lignes (`r_lines`) correspondant à une réponse.
Cette fonction donne également un nom générique à la question si cette dernière n'en a aucun indiqué dans le fichier LaTeX. Ce nom est « QuestionN » avec N le rang de la question dans la liste des questions extraites. Le nom généré peut donc ne pas être unique dans la base de données.

`parse_reponses(r_lines):` Crée un objet de classe `QCM.Reponse` à partir des lignes `r_lines` correspondant à une réponse.

Gestion :

Ce module est le cœur de l'application. C'est le module qui fait le lien entre les quatre autres modules. Tous les autres modules devraient éviter autant que possible d'utiliser d'autres modules que celui-ci.

Note 1 : la variable **`view`** *ainsi que les fonctions associées sont utilisées par l'interface graphique* **`GUI.py`**. *Elles sont laissées dans le code si une prochaine équipe souhaite continuer le développement de l'interface graphique, mais cette documentation se concentre sur l'utilisation en ligne de commande. Nous n'en parlerons donc pas.*

*Note 2 : Certaines fonctions du module ne sont jamais utilisées, mais pourraient l'être si de nouvelles fonctionnalités sont ajoutées à l'avenir. Nous avons donc décidé de les laisser dans le code et de les inclure à la documentation. Elles sont précédées d'une *.*

`sel: List[Tuple[str, QCM.Question]] = [] :` La liste des questions actuellement sélectionnées : ce sont celles auxquelles on va appliquer une opération (export, tag, affichage…). La sélection s'effectue depuis la base de donnés JSON. La chaîne de caractères de la paire contient l'id de la Question.
`buffer: List[QCM.Question] = [] :` La liste des questions résultant du parsing d'un fichier LaTeX. Elles ne sont pas encore dans la base de données.
`selbuff: List[QCM.Question] = [] :` Une sélection de questions du buffer. Elle est différente de la sélection **`sel`** car les questions de **`selbuff`** ne sont pas encore dans la base de données (on les sélectionne afin de pouvoir éventuellement les y ajouter).
`db: DB.Base :` le dictionnaire python représentant la base de données.

`init():` Cette fonction doit toujours être appelée afin de pouvoir utiliser les autres fonctions du module.
Elle initialise la base de données (c'est à dire initialise la dictionnaire python à partir du fichier JSON, cf. classe **`DB.Base`**) ou bien la crée si elle n'existe pas.

`export_sel_latex(filename):` Exporte les questions sélectionnées depuis la base de données dans le fichier filename au format LaTeX.

`export_sel_moodle(filename):` Exporte les questions sélectionnées depuis la base de données dans le fichier filename au format LaTeX Moodle.

`*export_buffer_latex(filename):` Exporte les questions du buffer dans le fichier filename au format LaTeX.
`*export_buffer_moodle(filename):` Exporte les questions du buffer dans le fichier filename au format LaTeX Moodle.

`parse_file(filename):` Parse le fichier LaTeX `filename` et stocke les questions extraites dans le `buffer`. Renvoie le nombre de questions trouvées.

`save_buffer():` Insère le contenu du `buffer` dans la base `db`. Attention, cela modifie le dictionnaire python, pas le fichier JSON : un appel à `persist()` reste nécessaire pour que les modifications soient conservées après la fin du programme.

`save_sel_buffer():` Insère le contenu du `selbuff` dans la base `db`.
`*remove_buffer/sel(index):` Supprime la question d'index `index` du `buffer/sel`.

`*save_sel():` Sauvegarde les modifications effectuées sur les questions de la sélection. Ne les enregistres pas à nouveau dans la base, mais modifie leur entrée.

`*update_index(index, update):` Met à jour la question à d'id `index` de la base, en la remplaçant par `update`.
`persist_db():` Appelle `db.persist` pour répercuter les modifications apportées au dictionnaire python `db` sur la base de données JSON. Sans appel à cette fonction, les modifications ne sont pas enregistrées après la fin du programme.

`apply_tag(index, tag):` Ajoute le tag `tag` à la liste des tags de la question de la base d'id `index`. Des variantes permettent d'appliquer le tag à : toute la sélection (`apply_tag_all`), tout le buffer (`apply_all_buffer`), ou la question d'id `index` du buffer (`apply_tag_all_buffer`).

`remove_duplicates():` Supprime les entrée en double de la sélection.

`select_[attribute](attribute) :` Ajoute à la sélection les questions dont l'attribut (`tags`, `name`, `keywords`) correspond.
Pour `tags` et `keywords`, l'entrée est une liste de tags/keywords et les questions renvoyées doivent posséder *tous* les tags/keywords de la liste. Les keywords sont cherchés dans l'énoncé de la question, pas dans son nom ni ses réponses.

`select_id(db_id):` Ajoute à la sélection la question d'id `db_id`, et renvoie `true` si elle a bien été trouvée.

`select_buffer_name(name):` Ajoute à la sélections `selbuff` les question du `buffer` nommées `name`.

`select_all():` Ajoute toutes les questions de la base de données à la sélection.

`*refresh_sel():` Rafraîchit la sélection, c'est-à-dire remplace chaque question de la sélection par la question de même id dans la base de données. Cela permet de voir les changements qui viennent d'être appliqués aux questions de la sélection.

GQCM :

C'est l'application en ligne de commande utilisée par l'utilisateur. Elle n'est constituée que d'un seul script qui interprète les attributs entrés par l'utilisateur et effectue divers opérations au besoin. Le module Gestion est écrit de sorte à ce que GQCM.py n'utilise que des fonctions du module Gestion.

Le fonction mesure le nombre d'arguments et préviens l'utilisateur s'il manque des arguments : ce dernier est alors amené à entrer les attributs manquants à chaque étape.

Pour plus d'informations concernant les différentes utilisations de GQCM, se référer à la documentation utilisateur.