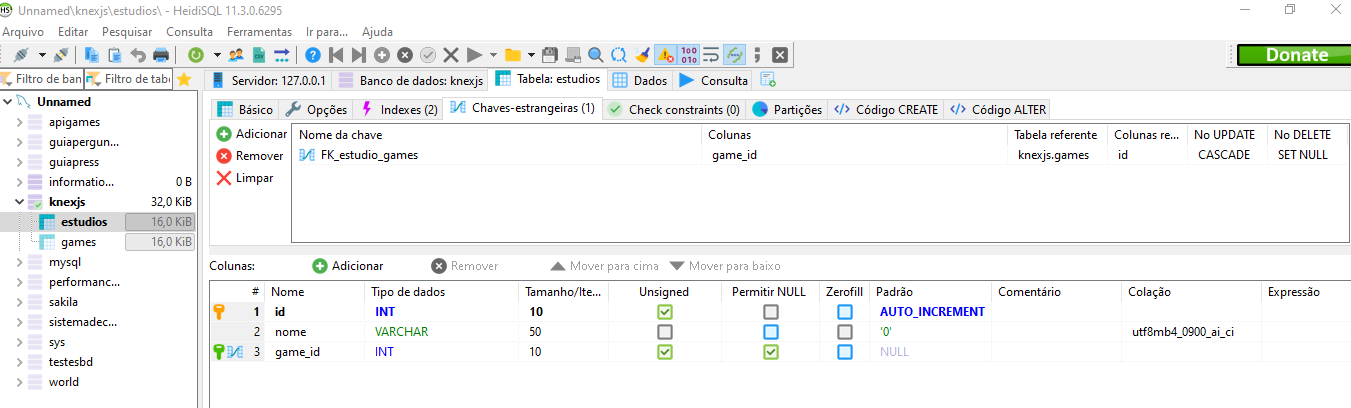
ROTEIRO PARA UTILIZAÇÃO DO KNEXJS NO NODE

1. Na página <https://knexjs.org/> há toda documentação do knexjs
2. Crie uma pasta chamada banco
3. Na pasta banco incie um novo projeto
4. ‘npm init’
5. Instale o knex: ‘npm install knex –save’
6. Instale a biblioteca para o mysql2
7. Npm install mysql2 –save
8. Crie um novo banco de dados chamado ‘knexJs’
9. Crie um arquivo de configuração chamado databaseKnex.js
10. No arquivo databaseKnex.js crie a configuração da conexão
11. var knex = require('knex')({
12. client: 'mysql2',
13. connection:{
14. host:'127.0.0.1',
15. user:'root',
16. password:'123456',
17. database: 'knexJs'
18. }
19. });
20. *module*.*exports* = Knex;
21. Crie um arquivo chamado index.js
22. No arquivo index.js realize a importação do database.js
23. const database = require("./database");
24. Crie uma tabela utilizando o heidisql ou outro aplicativo
25. Após criado a primeira tabela vamos fazer o primeiro insert;
26. No arquivo index.js
27. Crie um array com os dados que serão inseridos
28. //criando os dados que serão inseridos na tabela
29. var dados = [
30. {
31. nome: "Call of duty",
32. preco: 76.34
33. },
34. {
35. nome: "Call of duty 2",
36. preco: 98.34
37. },
38. {
39. nome: "GTA",
40. preco: 60
41. },
42. {
43. nome: "WOW",
44. preco: 120
45. }
46. ]
47. Configure o database para inserir os dados na tabela ‘games’
48. /forma correta para salvar no banco de dados
49. database.insert(dados).into("games").then(*game*=>{
50. console.log(*game*);
51. console.log("Registro salvo com sucesso");
52. }).catch(*err*=>{
53. console.log("Erro ao salvar no banco de dados: "+ *err*)
54. });
55. Vamos agora realizar uma consulta com ordenação de todos os registros da tabela games
56. //realizando um select de todos os registros
57. database.select().table('games').then(*games*=>{
58. console.log(*games*);
59. }).catch(*err*=>{
60. console.log(*err*);
61. });
62. VAMOS MUDAR NOSSA ABORDAGEM PARA UTILIZAÇÃO DE CLASSE E MÉTODO ESTÁTICO EM UMA FACTORY DE QUERIES.
63. Vamos criar uma classe chamada FactoryQueries.js que irá importar o arquivo do banco de dados “databaseKnex.js” e terá métodos estáticos que serão as SQL´s para um CRUD, além de buscas mais sofisticadas.
64. const database = require('./databaseKnex');
65. class FactoryQueries{
67. //Função para realizar cadastro em qualquer tabela
68. static insert(*data*,*table*){
69. return **new** *Promise*((*resolve*,*reject*)=>{
70. setTimeout(() => {
71. var erro;
73. if(*data* ==undefined || *table* == undefined){
74. erro = "A função possui dois parâmetros e devem ser preenchidos";
75. }
76. resolve(
77. database.insert(*data*).into(*table*).then(*result*=>{
78. console.log("registro inserido :"+ *result*);
79. }).catch(*err*=>{
80. erro = "Erro na execução: "+ *err*;
81. })
82. )
83. if(erro != undefined){
84. reject(console.log(erro));
85. }
86. }, 1000);
87. });
88. }
89. //SELECT para todos os casos basta adicionar o campo que ser quer localizar
90. static select(*fields*,*table*, *orderField*=undefined, *orderType* = undefined){
91. return **new** *Promise*((*resolve*,*reject*)=>{
92. setTimeout(() => {
93. var erro;
95. if(*orderField* == undefined){
96. *orderField*="id";
97. }
98. if(*orderType* == undefined){
99. *orderType* = "asc";
100. }
101. if(*fields*==undefined){
102. *fields*= '\*';
103. }
104. resolve(
105. database.select(*fields*).table(*table*).orderBy(*orderField*,*orderType*).then(*result*=>{
106. console.log(*result*)}).catch(*err*=>{
107. erro = *err*;
108. })
109. )
110. if(erro != undefined){
111. reject(console.log(erro));
112. }
113. }, 1000);
114. });
115. }
116. //utilizando where
117. static where(*condition*,*table*){
118. return **new** *Promise*((*resolve*,*reject*)=>{
119. setTimeout(() => {
120. var erro;
121. resolve(
122. database.where(*condition*).table(*table*).then(*result*=>{
123. console.log(*result*);
124. }).catch(*err*=>{
125. erro = *err*;
126. })
127. );
128. if(erro != undefined){
129. reject(console.log(erro));
130. }
131. }, 1000);
132. });
133. }
134. //utilizando whereRaw
135. static whereRaw(*condition*,*table*){
136. return **new** *Promise*((*resolve*,*reject*)=>{
137. setTimeout(() => {
138. var erro;
139. resolve(
140. database.whereRaw(*condition*).table(*table*).then(*result*=>{
141. console.log(*result*);
142. }).catch(*err*=>{
143. erro = *err*;
144. })
145. );
146. if(erro != undefined){
147. reject(console.log(erro));
148. }
149. }, 1000);
150. });
151. }
152. //funcão para deletar um registro pelo ID
153. static deleteId(*id*,*table*){
154. return **new** *Promise*((*resolve*,*reject*)=>{
155. setTimeout(() => {
156. var erro;
157. var jsonId = {id: *id*};
158. resolve(
159. database.where(jsonId).delete().table(*table*).then((*result*)=>{
160. if(*result* == 1){
161. console.log("Registro apagado "+ *result*);
162. }else{
163. console.log("Registro não existe.")
164. }
165. }).catch(*err*=>{
166. erro = *err*;
167. })
168. )
169. if(erro != undefined){
170. reject(
171. console.log("Não foi possível apagar o registro")
172. )
173. }
175. }, 1000);
176. });
177. }
178. //função para editar um registro pelo ID
179. static editId(*id*,*data*,*table*){
180. return **new** *Promise*((*resolve*,*reject*)=>{
181. var erro;
182. var jsonId = {id: *id*};
183. resolve(
184. database.where(jsonId).update(*data*).table(*table*).then(*result*=>{
185. console.log("Registro atualizado "+ *result*)
186. }).catch(*err*=>{
187. erro = *err*;
188. })
189. )
190. if(erro != undefined){
191. reject(
192. console.log("Erro ao atualizar registro: "+ erro)
193. )
194. }
195. })
196. }
197. }
198. module.exports = FactoryQueries;
199. Todos os métodos utilizão promises e por isso criamos também outra classe chamada testeClasse.js que irá importar a classe do banco de dados e a fabrica de queries,
200. //importa as classes de fabrica de queries e
201. const DatabaseKnex = require('./databaseKnex');
202. const FactoryQueries = require('./FactoryQueries');
203. instanciamos a classe do banco e passamos os dados que serão usados para se conectar ao banco de dados
204. //instancia o banco de dados com as variáveis necessárias para a conexão
205. const database = **new** DatabaseKnex('mysql2','127.0.0.1','root','123456','Knexjs');
206. Criamos um método principal que deve ser do tipo async pois ele executara métodos do tipo await,
207. Neste método passamos os dados que serão inseridos ou alterados de maneira simples, através de um array/json, além das linhas que possuem a execução dos métodos estáticos da classe FactoryQueries
208. async function principal(){
209. var data=[
210. {
211. nome: "Far Cry 6",
212. preco: 130
213. }
214. ]
215. //  var inserido = await FactoryQueries.insert(data,"games");
216. // var search1 = await FactoryQueries.select({nome: "nome", preco: "preco"},"games","preco","desc");
217. var search1 = await FactoryQueries.select("","games","preco","desc");
218. var search2 = await FactoryQueries.where({preco : 120 },"games");
219. var search3 = await FactoryQueries.whereRaw("preco > 140  or nome = 'GTA'","games")
220. var result = await FactoryQueries.deleteId(18,"games");
221. var editado = await FactoryQueries.editId(3, {nome: 'Call of duty Black ops', preco: 143}, "games");
222. // console.log(inserido);
223. console.log(search1);
224. console.log(search2);
225. console.log(search3);
226. console.log(result);
227. console.log(editado);
228. }
229. Por último executamos o método principal
230. principal();
231. VAMOS AGORA TRABALHAR COM CRIAÇÃO DE RELACIONAMENTO ENTRE TABELAS
232. Vá em algum aplicativo de banco de dados e crie no banco de dados Knexjs a tabela estudio com os campos id, nome e game\_id, onde o game\_id são os games que possuem fazem parte daquele estúdio e é uma chave estrangeira entre a tabela games x estúdio.
233. A tabela já com a chave estrangeira ficará conforme figura abaixo.
234. 
235. Vamos fazer um join nas duas tabelas
236. Innerjoin = é utilizado quando há registros nulos nas tabelas.
237. Esse relacionamento é de 1 para 1
238. Vamos inserir outro método estático na classe FactoryQueries.js para realizar o innerjoin
239. //metodo para retornar um innerJoin entre duas tabelas do banco de dados
240. static innerJoinFields(*fieldsTableBase*,*tableBase*,*tableInner*,*tableInnerFK*,*tableBaseId*){
241. return **new** *Promise*((*resolve*, *reject*)=>{
242. var erro;
243. if (*fieldsTableBase* == " "){
244. *fieldsTableBase* = "id";
245. }
246. resolve(
247. database.select(*fieldsTableBase*).table(*tableBase*).innerJoin(*tableInner*,*tableInnerFK*,*tableBaseId*).then(*result*=>{
248. console.log(*result*);
249. }).catch(*err*=>{
250. erro = *err*;
251. })
252. )
254. if(erro != undefined){
255. reject(
256. console.log("Erro ao atualizar registro: "+ erro)
257. )
258. }
259. })
260. }
261. VAMOS MONTAR UMA TABELA PARA RELACIONAMENTO MUITOS PARA MUITOS
262. Sempre que formos trabalhar com relacionamento muitos para muitos, devemos criar uma tabela que contenha os id´s dessas tabelas .
263. Para criar relacionamento muitos para muitos vamos criar uma terceira tabela que terá um id primarykey para os registros e mais dois campos ForeinKey um para cada tabela relacionada.
264. Agora vamos criar um busca nas duas tabelas e para isso faremos dois joins um para cada tabela