**RS.ts Overview**

**BufPack -** name/value (number,string,BufPack,ArrayBuffer) pairs can be added using the **add** function, and BufPack data can be converted to/from AB (ArrayBuffer) using the bufIn/bufOut calls. Names starting with < '0' denote control pairs (used to define handling of the data), while alphanumeric names (>= '0') define data pairs. BufPack can be used to pass data to/from client and server, but also to/from database. For example, to store a record to the database, we first add the data fields (name/data pairs) to the BufPack, then add a control name/data pair ('!Q' {query} ,'I' {insert}) and pass the BufPack to packRequest (which returns a BufPack as result). {Note that RSData has a direct function "toDB" that sends the record to the database.} Each of the name/value pairs is stored in a **PackField,** which is returned by BufPack.field (name), or the values can be returned if desired by BufPack.str (name), BufPack.num (name) or BufPack.pack (name). A BufPack may also be "packed" from an array of BufPacks using the BufPack.pack ([bufpackarray]) and BufPack.unpack returns the original [bufpackarray]. BufPack.multi is TRUE if the BufPack is a "packed" BufPack containing other BufPacks.

BufPack.desc returns a multiline string about the BufPack, and BufPack.expand returns a multiline string which expands any BufPacks which are packed into the BufPack.

**RSData -** the RS base data class (including database operations) from which most other data classes are extended. Standard Fields in RSData records are Name, Desc, Type, Sub{type}, ID {unique database ID, if known}, Details, Str and **Data** (which stores the specialized data which is unique to the specialized class extending RSData). A database editor or viewer can thus display standard fields (Name/Desc/Type) for each of the records, and have a special function for displaying/editing the specialized **Data.** RSData.LoadPack will load the data from a BufPack, and RSData.SavePack will store its data to a BufPack. The constructor for RSData will accept a BufPack and load its data appropriately. The load/save functions will also call the appropriate specialized function to handle the specialized **Data** in extended records. For example, **vList** is extended from RSData, and LoadPack will call its specialized "PostLoad" function to load handle its specialized **Data. RSData** constructor accepts a BufPack argument, loading its data from the BufPack.

A nice overview of database operations can be found in page.svelte in the APItesting client area.

**vList/vID/ListOfLists**

**v{ariable}List** is extended from **RSData**, its constructor accepts a string defining its name/desc pairs in the form:

{name:desc}|name1:desc1|…|nameN:descN|

Where the last character in the string is the delimiter of the name:desc pairs (in the usual case, delimiter is '|'). This allows a list of variables (parameters) to be passed as a single string, and variable values can be quickly accessed in the vList by name (GetStr (name), GetNum (name), GetDesc (name)). GetVID(name) will return the v{ariable}ID. The elements of the list can also be directly loaded to a select/ListBox for display to the user.

**vLists provide an efficient way to displaying (or accepting) various typed values e.g. Country Names, Language List, ListTypes, etc. In general, we cannot accept free format strings from user because his country name could be misspelled (Irland != Ireland).**

vLists can also be merged. This is important for inheriting parameters and behavior. Our Magic Tile, from which ALL other Tiles are descended, will have a long set of default parameters stored in a vList. Its descendant (e.g. Button) will have a short vList of those parameters that are different than Magic Tile, and this vList can be merged with the Magic Tile vList, replacing the values that have changed and adding new ones. The RoundButton descendant of Button will have a short vList, containing only those parameters which are different than Button. When the instance of a Tile is placed into another Tile (e.g. a RoundButton), the instance of that Tile would have its own vList, defining its changes from the standard RoundButton (e.g. color, text, activated options, junctions to other Tiles, etc.). This would be merged into the Magic Tile merged with Button merged with RoundButton to form the final operating vList for that instance of RoundButton.

vLists also allow for alias (dot notation) of parameters. There are many cases where you would like to have a named SET of parameters to implement a complex behavior. In such cases, this set can be designated in the vList as |.namedalias| where "namedalias" is the name of the set of parameters, defined within an alias vList as |brownround=color:brown=text:ABC=image:roundbutton|

**v{ariable}ID** also can contain an optional format at the start of its description, which may include a value. This allows list to be specified with formats for their input values, allowing a menu to be created for the user in which he is allowed to enter certain types of values (e.g. number within a range, member of a vList, string of certain length, etc.)

**TileStrings -** a specialized type of vList, can be defined by an array of strings or by a single string, a highly efficient way to represent the children contained by a Tile.

const TileStrings: string[] = [

        'T\ta|name:Full|\ts|display:flex|flex-direction:column|align:center|justify:center|background:black|min-width:750px|max-width:750px|min-height:500px|\t',

        ' T\ta|name:Top|\ts|background:magenta|min-height:150px|\t',

        '  T\ta|name:Left|\ts|background:green|min-width:100px|\t',

        '   T\ta|name:Top|\ts|background:magenta|min-height:50px|\t',

        '   T\ta|name:Bottom|\ts|background:magenta|min-height:100px|\t',

        '  T\ta|name:Right|\ts|background:cyan|width:100%|display:flex|\t',

        ' T\ta|name:Bottom|\ts|display:flex|flex-direction:row|background:white|min-height:350px|\t',

        '  T\ta|name:Left|\ts|background:green|min-width:100px|\t',

        '  T\ta|name:Middle|\ts|background:cyan|width:100%|display:flex|\t',

        '  T\ta|name:Right|\ts|background:yellow|min-width:200px|\t'

    ];

Each Tile name is preceded by a number of spaces, denoting the "level" of the Tile. The first Tile (type T) has 0 preceding spaces, indicating it is level 0. The next (also of type T) has 1 space, indicating level 1, and it is a child of the Tile above it with a lower level (in this case, level 0). The next two Tiles (also type T) are preceded by two spaces, making them level 2, and their parent is the Tile above them of lower level (1).

Each Tile name is followed by multiple TAB delimited strings which are the name/desc pairs of a standard vList. In this case, the TileList is a vList composed of multiple vLists within it. The whole thing can be read/stored as a single string with different delimiters for the embedded vLists (standard vList delim is '|', next level up is TAB, next level up is NL ('\n' or terminated string).

The details of how this is handled within the vList are beyond scope of this discussion - just know that vLists are highly versatile and efficient ways to handle these data.

**TileID -** referring to a Tile

{VillaName:TileName}=InstanceName{.InstanceName.InstanceName}

Each Villa (user home) has a unique alphanumeric name (preferably relatively short, while the description of the villa can be as long as desired). The name of our system villa (in which we are now working) is "S". With a Villa, each Tile has a unique name. In this case, for simplicity, we are also calling our primary Tile "S", where we are storing all of our main data. So a user wishing to refer to the main Tile (S) in the system Villa (also named S) would use "S:S". If he wanted to place a local instance of a Tile named "Q" in the system Tile, he would specify that by "S:Q=ABC" where ABC is the name of the local instance of that Tile within the Tile he is creating. Each Tile instance name must be unique, within the Tile you are creating. To refer to another Tile instance in a Tile that you are creating, you can specify only its unique name. If you wish to specify a Tile instance WITHIN that unique Tile, you can add .InstanceName.InstanceName…For example, suppose you place an instance of the "S:Q" Tile we named above into a Tile we are designing, calling it ABC. Of course, the ABC Tile can have Tiles already designed within it (named by its designer). Imagine there is a DEF Tile which is a child of ABC Tile, and within DEF, a child named XYZ. That Tile could be referred by ABC.DEF.XYZ. (Note that the VillaName:TileName (S:S) would not be required, since ABC uniquely identifies that Tile instance.)

**DB Requests -** Two special functions ReqPack and ReqAB are initialized by calling InitReq on the client and server. After that,  **ReqPack/ReqStr/ReqTiles/ReqNames** function the same way on client and server - the result or data is returned in a BufPack. Any RSData (descended class) can be stored to the database by RSData.toDB (). ReqStr (SQLStr) submits the string to the SQL database and returns result in a BufPack.

**SQL -** class contains multiple functions for constructing SQL queries and preparing a BufPack for database operations. Simply saving an RSData record to the database can be performed with RSData.toDB ().

**BONUS PROBLEM**

Our database operations between server and client seem to be "completing" and moving on in the code BEFORE the operation has actually been completed, receiving proper data. This is DESPITE the fact that we are using "await" for these async operations. If anyone can figure out a way to be sure that ALL pending async operations and their promises are complete before moving on, I would be grateful. {We are currently placing console.log operations in the code to add delay and give time for database or transmission to complete.}

Note: This file is currently in the repo/src/lib next to RS.ts.