Service

1. Service definitions can be organized into modules. This is for convenience.
   1. moduleName attribute is of the form a.b.c etc.. Naming and storage follow the Java package and class convention. That is, the .xml file must be stored in the folder structure that mimics the module name. Service names within a module are to be unique but different services across modules may have the same name. A fully qualified service name is to be used to refer to a service. E.g. moduleName=”inv.stock” name=”issueMaterial”. File named issueMaterial.xml is expected in sub-folder /inv/stock/. This service is to be referred as “inv.stock.issueMaterial”
2. User defined service: A service can be completely managed by a custom code. className=”fullyQualifiedName” is to be used for this. This class must implement org.simplity.service.ServiceInterface. Once this is specified, other attributes are all ignored. (except of course the name and moduleName attribute)
3. RDBMS access and transaction management: A data base connection of the right type is made available to the service and commit/rollback is handled by Simplity.
   1. dbAccessType attribute is used to declare the type of db-access. “none”, “readWrite”, “readOnly” and “autoCommit” are the valid values.
      1. If dbAccessType is not “none” a schemaName attribute can be specified to be used for this service. This feature is to be used if this service uses a schema that is different from the default schema.
      2. If dbAccessType is ‘readWrite” then a transaction is initiated before the actions of the service are called. This is committed if and only if the service exits normally, and the service context has no errors in them. Else the transaction is rolled-back. A service is modeled as a single RDBMS transaction by default.
      3. Connection object is made available to custom java action. However, the custom code should not alter the transaction (no commit/rollback/begin etc..)
4. Input Data Specification : Expected input for a service can be specified. This is used to validate the input data. Service is invoked only if the input data conforms to the specification. Else request is returned with due error message.
   1. Input may contain fields. These are extracted into the fields collection in service context.
      1. Fields may be specified specifically for this service.
         1. Field is identified by a name. name is to be unique for a service. However, it may be the same as column name of any other list/table data.
         2. Field must be associated with a pre-defined dataType. Value of the field as received from client is validated using the dataType specification.
         3. A default value may be specified for a field. This value is deemed to have been supplied by client if it is missing or is empty in the input.
         4. Field may be marked as mandatory.(isRequired=”true”) Validation error is raised if user does not supply a value for a field marked as mandatory. Empty text field is deemed to be not-specified.
      2. Fields may be specified using a record structure (inputRecord with sheetName not set)
         1. Field specifications in the record are used for extracting and validating input data.
         2. A subset of fields for a record can be used with the attribute fieldnames=””. For the purpose of this input, the record is assumed to have only these subset of fields.
         3. A record may be used with a purpose.
   2. Input may contain records.
      1. A record specification is mandatory to specify the fields/columns expected.
      2. If sheetName is specified, data is expected to be a list/table. Otherwise they are fields.
      3. Record is used with specific purpose. “read”, “filter”, “save” and “subset” are possible values.
         1. “read” means it is meant to read this record with primary key. Only the primary key field of the record is expected as input. Other fields are all ignored.
         2. “ filter” means the fields in this record are used to specify filtering criterion to get a subset of rows .
            1. Each field is assumed to be optional.
            2. Field value is validated for the value-type (like numeric/date etc.) but dataType specification is not used for validation. For example a dataType may have min-length of 5, but as filter field we accept even if only one character is specified.
            3. Each field may optionally have an associated condition named as fieldNameOperator. For example if fieldName=”cusomerName” then customerNameOperator. If this is missing, a value of ‘=’ is assumed.

‘=’ means exact match.

‘!=’ means should not match.

‘<’ means less than. Valid only for numeric and date fields.

‘>’ means greater than. Valid for numeric and date fields.

‘<=’ means less than or equal. Valid for numeric and date fields.

‘>=’ means greater than or equal. Valid for numeric and date fields.

‘><’ means value is to be a range(inclusive). fieldValue is from-value. fieldNameTo is the name field. For example amount=”10” amountOperator=”><” amountTo=”20” will match rows with amount fild between 10 and 20 (inclusive)

‘~’ means containing this text anywhere

‘^’ means field that starts with this text

‘@’ means matching any one value in the list. fieldValue should have a comma separated list of values. For example color=”red,blue,green”, colorOperator=”@” means rows with color set to red, blue or green.

* + - 1. “save” means this is used for add/modify operation. In addition to fields in the record, a special field named as \_saveAction may be sent by client with one of the following values.
         1. “add” means the row is to be added.
         2. “modify” means the row is to be modified with the new values.
         3. “delete” means the row is to be delated.
         4. “save” means the row is added if primary key is missing, else is used to modify the row with the specified primary key.
         5. Default is “save”
      2. “subset” means this is a special update operation where only the fields that are received are updated into the db, and the rest are not touched. Primary key field is mandatory, and all other fields are optional.
    1. minRows can be specified.
    2. maxRows can be specified. It is generally a good idea to specify this to a practical limit, rather than leaving it as zero, implying unlimited.
    3. We accept hierarchical data from client, and convert them to individual data sheets. This feature is triggered by setting parentSheetName attribute. For example if sheetName of this record is “orders” then each row in the parent sheet will have an attribute named “orders” with an array of child rows for this sheet.
       1. A single data sheet is created with al rows cross all rows of the parent.
       2. linkColumnInThisSheet specifies the field/column in this sheet that has the value of the parent key. Value for this is picked up from the corresponding parent row. Name of that column in the parent row can be different and hence an attribute named linkColumnInParentSheet is to be used to specify that column name.
    4. Inter-field relationship as specified in fields of record are used for validating input. For example from-to field values.
  1. If a service has the same input requirement as another one, then we can refer to that service rather than copy-pasting. referredServiceForInput=”qualifiedServiceName” does this job.
  2. Input string(payload) need not be processed as input data. It may be just assigned to a field in the service context. This can be processed by any custom action. requestTextFieldName=”fieldName.”
  3. Fields that are actually tokens for attachments can be specified so, and they are replaced automatically with the permanent stored keys.
  4. Columns in sheet may also be designated as attachment tkens.

1. Output Data Specification : Data to be included in the response can be specified. Response is prepared based on this specification and the data available in the service context. Output data are not subject to validation, as it is pretty much part of the service implementation to ensure that the service context contains right values for output specification.
   1. Output may contain field names. Values for these field names are extracted from service context and added to the output payload. If field is not present in context, it is not added to the output.
   2. Output may contain sheet (list/table) names to be sent to client. A data sheet with the given name, if present in service context, is extracted and sent to client as it is.
   3. Output may be extracted and formatted based on record definitions. This element is called outputRecord.
      1. Output record may be used to specify all the fields in that to b output as fields. This is a short cut to listing the fields in outputFields. If sheetName is not specified, and the service context does not have a data sheet with the same name as the record, then fields are output.
      2. Output record may be used to send a data-sheet (list/table) to the client. sheetName may be specified it is different from the recordName.
      3. Output record may be used to send hierarchical data to client. parentSheetName, linkColumnInThisSheet and linkColumnInParentSheet are the attributes to facilitate this feature. For each parent row, relevant child rows, if any, are added as an array for an attribute named as child sheet name. For example a row in customer will have “orders”:[{….},{….}…] as its child rows.
      4. If there are no child rows in a hierarchical data, value for child-attribute is set as “null” instead of an empty array. For example {…., “orders”:null}. This feature is subject to change to a more elegant output like {…., “orders”:[]}
      5. Hierarchical data could be of any level. That is a child sheet may be set as parentSheet for another sheet and so on.
   4. Fields may be added to user-session. This is independent of what is sent as response to client. sessionFelds=”commas separated list”.
      1. Field may be available in fields collection of service context. In this case, this field will be available in service context of all subsequent service calls.
      2. Field may be available as any java object in service context. This will be made available as object in service context for subsequent services.
   5. Fields that are actually tokens for attachments can be specified so, and they are automatically retrieved and made available to the clients.
   6. Columns in sheets may also be designated as tokens.
   7. Output specification is not used, and data is not sent to client if service context has errors.
   8. If service context has errors, an error response is sent to client.
2. If output specification for a service is same as the one for another, then it can be referred rather than copy-pasting it.
3. If the response text to be sent to client is already prepared by the service in a specific field, then that field may be specified instead of extracting and formatting using output specification.
4. Service may be tagged for caching. That is, the client-layer can cache the output and re-use it for subsequent request instead of calling the server again. This caching can be unconditional, or for specific values of one or more fields. For example, getStateNames service may be marked to allow caching for field countryName. Client layer can cache response by country and re-use them.
5. Service may be specified to be executed in the background. That is, the service call will return immediately to the client layer, but will execute in the background in another thread. This feature is not yet implemented. Will be implemented on a need basis, with a specific mechanism to respond back to the client as and when the background completes.
6. Service may be directed to just accept the input as it is. This feature is quite dangerous, and should not be used in any application that is available to end-users. Hence this feature is not exposed to service.xml, but is available to be set programmatically from another java code. In this case all data elements are extracted into service context with no validation.
7. Service may be directed to output everything from service context to client. This being a dangerous from security point, is not exposed as a feature to xml. This can be used internally by another java class.
8. A service can be used as an action in another service. (as subservice action) In this case, input and output specifications are ignored and only actions are executed.
9. Using features available in its actions, a service may messages to the service context. Such messages are sent to the client. Messages text is in its primary(English?) language. However it also contains messageName and any run-time values, using which serviceAgent (caller of service) can translate them to client-specific language.
10. All dates are received and sent in yyyy-mm-dd format. Al times are in Z-format. It is up to the client side application to render/transform them to desired format.
11. Service has an ordered list of actions (zero or more)
    1. Action may have a logical expression (an expression that evaluates to a logical value) as a pre-condition to execute. Action is executed only if this expression evaluates to true.
    2. Action may be associated with a sheet as a condition to execute. We may specify that the action is executed only if there are rows in this sheet. Alternately, we specify that the action be executed if there are no rows in the sheet.
    3. An action may explicitly direct the execution to stop. Other actions are abandoned and the service execution is deemed to have completed normally.
    4. An action may signal an error and halt the execution by throwing ApplicationError. Service execution s halted and assumed to have failed.
    5. Returned value from an action is available in the service context with the name actionnameResult. For example if action name is “readRows” then readRowsresult will have the value returned by that action.
12. Service provides features to manipulate data in service context
    1. A field can be added or modified in service context using setValue action.
       1. Value can be just a constant. Use fieldValue=””.
       2. Value can be set to another field in service context. Use fieldValue=”$otherField”.
       3. Value can be an expression that may contain constants, field names and functions.
    2. Logged in user id can be copied to any other field using copyUserId action.
    3. A data sheet can be created with createSheet action. Sheet name and data are known at design time.
    4. A data sheet can be renamed using renameSheet action.
    5. Rows from one sheet can be copied to another using copyRows action.
    6. A column can be added to a data sheet at run time using addColumn action. Value of this column may be either specified as constant (if it is known at design time to be same for all rows) or as an expression to be evaluated at run time for each row.
13. A message can be added to service context possibly with run time parameters using addMessage action.
14. Service provides features to deal with the data base. (Common for all actions that deal with db)
    1. A message, possibly with run time parameters can be added to service context if the db operation succeeds.
    2. A message, possibly with run time parameters can be added to service context if the db operation fails.
    3. Service provides flexibility to either continue or stop if the added message is an error.
15. Whenever one or more rows are read from a table using record, relevant rows from child-records may also be read. This feature is available with record-based actions that read rows from data base.
    1. One parent may have more than one child records.
    2. Child records may be specified as part of action attributes.
    3. Child records to be read may also be specified as part of record definition. In this case, action level control is available to enable/disable cascading of child record read operation.
16. Whenever a row is inserted or updated in a parent table, rows may be added/inserted in its child-records. This feature is enabled for record-based actions that save data.
    1. One parent may have more than one child records.
    2. Child records may be specified as part of action attributes.
    3. Child records to be saved may also be specified as part of record definition.
    4. Only one level of operation is allowed. It is not possible to insert/update into grand-children records. (This is primariliy a limitation of the underlying driver that makes such an operation quite in-efficient if we do it automatically. It is better for the service designer to explicitly take care of them using other features)
17. A row can be read based on its primary key using a read action. This requires a record to be defined to represent the table/view being read.
    1. Read can be based on the primary key in fields collection, in which case at most one row is read.
    2. Read can be for each row in a sheet. Specify inputSheetName to use this feature.
    3. Output can be into fields collection or to a sheet (even if there is a single row)
    4. Rows from one or more child-tables can be read using related record feature.
18. Rows from a table/views can be read based on filtering criterion on their fields. filterAction provides this feature.
    1. Filtering criterion, (where clause for a sql) is formed based on filterfield concept. (field name, operator and optionally fieldTo name)
    2. Filter criterion may be specified based on a record which is different from the record that is used for read operation. This is useful when you want to extract a different set of columns from the underlying table/view than the columns for which you set where-clause conditions.
    3. Input data can be optionally in a data-sheet. However, we use only one from this data sheet.
    4. Child records may be explicitly specified as part of the action.
    5. Child records may also be picked-up from record specification with optional cascading of child-records.
19. Rows from a table/view may be read for a typical drop-down list as key-value pairs using specifications in a record. This is provided using keyValueListAction.
20. Rows from a table/view may be read to suggest matching rows for a client field (google-suggest-like) using suggest action.
21. Rows from a table/view can be read for a given value of its parent key using readChildren action.
22. It can be checked whether a row exists in a table/view for a given value of its primary key. This is useful if the actual data is not required, but we just want to check (typically validate) for existence of row. Rowexists action is used for this feature.
23. Rows can be extracted using a prepared statement. readWithSql action is used for this.