Service Feature List

This document provides a complete list of features. Features are structured with numbering than be used for organizing tests. We intend to provide link back to these numbers in test plans.

1. Service definitions can be organized into modules. This is for convenience. 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 ‘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.
      2. Connection object is made available to custom java action. However, the custom code should not alter the transaction (no commit/rollback/begin etc..) This is to stick to the underlying principle that one service is one rdbms transaction. However, in case of a specific requirement, use code can get another connection and do the needful.
   2. Connection details are configured at the application level
      1. connectionString, dbDriver and dbDriverClassName attributes in application.xml can be used.
      2. It is a better practice to use dbVendor and dataSourceName attributes instead of connection string. It provides more flexibility to database administrator.
   3. Uunderlying RDBMS associates a default schema for a given credential. This is the schema that is used by all services by default. However, if the project need to use more than one schema:
      1. Additional schemas and the required credentials can be given in application.xml using schemaDetails elements.
      2. A service can choose one of these alternative schemas. Simplity creates connection using this schema.
      3. It is not advisable for a service to have more than one RDBMS transaction boundaries. However, in case it is required, and the different transactions happen to be with different schema, a user code has the flexibility to do so.
4. Input Data Specification : Expected input for a service can be specified. Simplity parses request data for these input and invokes the requested service only if the input data passes all validations. In case of any validation error, a response is sent back to client with appropriate messages.
   1. Input may contain fields. These are extracted into the fields collection in service context.
      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.
      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. Input may contain records. A record may be used to get fields or a list/table of rows.
      1. If sheetName is specified, then input is expected in array. Else fields are expected as attributes. Accordingly, data is parsed into fields, or into a data sheet.
      2. In case input is expected as a single row in a sheet, but you want it to be extracted into fields collection, then extractIntoFields may be set to true. Additional rows, if any, are ignored.
      3. If array is not received as per sheetName, and minRows is not set to more than one, then we try and extract fields instead of sheet. That is, we assume as if sheetName is not specified.
      4. A record specification is mandatory to specify the fields/columns in an array to be extracted into a data sheet.
      5. 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 deleted.
         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. Default is unlimited. It is generally a good idea to specify this to a practical limit. Say 1000 for an interactive client, and say 10,000 if this service is used by batch uploads.
    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” and parentSheetName=”customers”, then each row in customers may an attribute named “orders” with an array of child rows for this sheet.
       1. A single data sheet is created with all rows across all parent sheet rows.
       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. Use referredServiceForInput attribute to specify the qualified service name from which to clone the input specification.
  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. Use requestTextFieldName=”fieldName.”
  3. Fields that are actually tokens for attachments can be specified. Simplity uses their value to get the attachment in the temp area, push them to permanent storage, and replace the value of the field with the token/key to this permanent storage.
  4. Columns in sheet may also be designated as attachment tokens.

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. Fields as name-value pairs can be put into response by specifying a comma separated list of field names. If a field is not present in the service context, it is not added to response.
   2. Datasheets (lists/tables) can be put into response by specifying a comma separated list of sheet names. If a sheet is not found in the service context, it is not added to response.
   3. Output may be extracted and formatted based on record definitions. This element is called outputRecord.
   4. sheetName of an outputRecord defaults to simple name of the record.
   5. We look for a sheet and output it as per fields defined in this record. If sheet is missing in the context, then we check if any of the field names in this record would clash with the fieldNames defined at the outputData level. If there is no clash, we try and output the fields as key-value pairs, as in 5.1.
   6. Output record may be used to send hierarchical data to client. parentSheetName, linkColumnInThisSheet and linkColumnInParentSheet are the attributes to facilitate this feature.
      1. 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.
      2. 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”:[]}
      3. Hierarchical data could be of any level. That is a child sheet may be set as parentSheet for another sheet and so on.
      4. If top level parent sheet is not found in the service context, child-sheets are output as if they are the top sheets.
   7. 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.
   8. Fields that are actually tokens for attachments can be specified. Attachment file is retrieved from permanent storage and made available in temp storage. Value of this field is replaced with the key to temp storage.
   9. Columns in sheets may also be designated as tokens.
   10. If service context has errors, an error response is sent to client. No data is sent.
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 the response 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.
   1. Caching can be unconditional. Response once received from server is cached for all subsequent requests.
   2. Caching can be for specific input values. 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 security risk, 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 add 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. All 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 database.
    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 primarily 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.
24. Data in a table or updatable view can be updated/inserted/deleted with the help of a record. This feature is available with save action.
    1. During an insert operation, if the primary key is auto-generated by the RDBMS, it is made available back in service context.
    2. Some rdbms systems automatically insert a value for auto-generated key field, while others require the a function/sequence to be used as field value in the corresponding sql. This aspect is taken-care of with the help of sequence attribute in record.
    3. A common operation called “save” may be used. Record is updated if primary key values is found, or inserted as a new row if primary key is missing (provided the key is marked for auto-generation)
    4. Update,insert or delete operation may be explicitly specified at design time.
    5. Update, insert or delete operation may also be specified at run time with specific value in a field named tableName\_action
    6. save operation can be extended to child rows of one or more child tables.
    7. while child rows are inserted for an insert operation, they may be inserted/updated appropriately in case of an update of parent.
    8. save operation automates population of created-time, created-user, modified-time and modified-user fields as per specification in record.
    9. a time-stamp-check is carried out before a row is updated. That is, existing modified-time is checked with the supplied value before the row is updated.
25. Data can be updated using prepared statement to which data is bound at run time.
26. Stored procedures defined in the underlying rdbms can be executed by binding data at run time.
    1. stored procedure parameters can be primitives, structs, array of primitives or array of structs.
    2. stored procedure can return data in output parameters, input-output parameters or as returned result sets.
27. Database row may contain a clob field. content of this field may be treated as a text field by the service/client. Simplity takes care of storing/retrieving this text value as part of the same transaction.
28. Simplity does not provide any feature to automate handling of blob fields. We discourage use of such a field. We recommend using “attachment” features wherever it is feasible to do so.
29. A field in a db row may be marked as a token for an externally-saved attachment(document). Simplity automates storage/retrieval and removal of such documents from permanent storage with the help of a storage-manager-adapter mechanism.
30. Service offered on another server may be executed as an httpClient action.
    1. url for the service may be known at design time, or parts of it may be assembled based on runtime data.
    2. accompanying data for such a service request may be made available in a field at run time.
    3. accompanying data for such a service may also be formatted the same way as a response is put together using outputData specification. Refer to outputData above. This feature is available only if the content type is json.
    4. service request may require proxy credentials. While proxy server details with port number are to be known at design time, userId and password may be made available at run time.
    5. output from this http service may be set to a field in the service context.
    6. all output fields at the top level (name-value pairs) can be extracted and put to the service context as text values if the content type is either json or xml.
    7. output from this http service may also be parsed into service context using inputData specification described earlier if the content type is json.
    8. Application error is generated in case of any http related error during this communication.
    9. content type could be anything, but data extraction and formatting features are restricted to only to json and xml.

Security

Simplity considers three aspects while dealing with security.

* Service level : Includes authorization and xxx-level for services for the logged-in user. For each service, the application has to decide whether the logged-in user is authorized to
  + avail this service.
  + deal with the specific document (record, data) being asked for.
  + Modify/create data as requested in this service.
  + Any business logic/rule that would apply differently based on the logged-in user

This is the responsibility of service designer. Simplity provides necessary tools and techniques, but it is the primary responsibility of the service designer to elicit required information from business users and implement them.

* Application level : Primarily deals with infection of run-time logic with help of malicious but legitimate data. For example SQL injection and script injections.   
    
  In general, this is the kind of security issue where an authenticated user makes a legitimate request but the data that accompanies the request exploits the programming vulnerabilities to change the behaviour of the application.   
    
  This aspect is handled primarily by Simplity. All the features provided by Simplity are thoroughly reviewed for such vulnerabilities, and we continue to re-check them based on latest best practices in the industry.  
    
  For example, Simplity transmits all data as pure data to the client. Client-side features of Simplity ensure that user-originated data is never parsed by the browser. Similarly, all SQLs that are executed at run time are either prepared statements, or are synthesized by Simplity based on data supplied by the service.
* External to the application : These are the risks associated the way the application is deployed on the field. For example if this is exposed to the internet. Simplity simply delegates this to the infrastructure. It takes no accountability, at the same time puts no restriction on the kind of security set-up that the infrastructure layer puts in place.  
    
  Micro-services architecture enforced by Simplity on the application design and development ensures that the application inherently has no awareness about the deployment. Application is ALWAYS working inside a secure JVM. Simplity supplied “agent” is its only connection to the external world. This arrangement provides complete flexibility for Simplity to keep evolving along with the infrastructure layer.

Logging

Simplity uses a thread-centric logging mechanism for the application. That is, when a service is executed, Simplity provides a single log entry or that service execution. This log entry has the text emitted by the underlying components as they do their part of the job. So the log provides a simple and clean account of how the service started with input-data and ended in providing output data. We call this “service-log”.

This mechanism should not be compared or confused with the general logging used by programming community. For example the way a programmer uses Log4J. That paradigm is class centric. Java code written for a Simplity application may continue to use such logging paradigms for their intended use.

Simplity allows you to decide where and how the service-log is to be written. By default, SImplity pushes the log to the console. If you want this to go to the logging infrastructure that is enabled for the project, you should use loggingFramework attribute in application.xml.

service-log is wrapped as an xml element with the log pushed as a CDATA section. If you need the log to be archived/organized in any other way, you may provide your wrapper using traceWrapper attribute of application.xml.

Debugging

How does a developer debug a service that is implemented in a .xml ?

Simplity provides a simple trace of what happened during the execution of a service. During development, this trace can also be rendered on the browser. This is quite helpful in figuring out what could be wrong with your service.

If you need to debug the java-way, you may include Simplity sources in your project, and debug your service exactly the way you would with your own class. Refer to Service.java class.

However, our experience has been that the actions you write in a service are so simple and straightforward that you do not require a source level debugging.

I18N

Simplity server deals with data and messages. Html pages or fragments are not designed at run time by Simplity. SImplity follows client-side-mvc pattern. Client-side components are all static, and may be statically generated and organized into appropriate folder structures. That is, the root folder may be selected based on the language for the app to be made available in different languages.

* Content of data is assumed to be coming out of databases, and hence there is no ready feature for any translation.
* Date fields are always exchanged in yyyy-MM-dd format.
* Date with time are always exchanged in UTC in yyyy-mm-ddTHH:MM:SS.fffZ format.
* messages are returned with a default text in default language. They accompany message code and any run-time data used for formatting this message. Client-side component of Simplity can be configured to translate this message before transmitting back to client.

Hosting the Application

Application designers and developers are completely unaware of the deployment scenario. All the services are designed for a JVM to be run independent of others. This approach provides complete flexibility for deployment, thereby making it future-proof in terms of advancements in infrastructure like cloud, clustered servers, in memory db etc..

Simplity provides configuration utilities and drivers at the application level to work with the available infrastructure.