# WhizniumSBE Code Generation/Iteration Service

## The Service Builder‘s Edition For Connected Embedded Systems

## Quick facts

- WhizniumSBE is an innovative software development tool providing automated code generation and iteration based on fine-grained model specifications.
- WhizniumSBE offers significant time savings and superior source code quality in the development process of real-time, multi-threaded embedded software. Reference projects are available online both for ARM-based (RaspberryPi3, gumstix, Zynq) and for Intel Atom-based (Minnowboard, Galileo) systems.
- Development with WhizniumSBE covers all major ingredients of IIoT / Industrie 4.0 software and delivers them in ready-to-deploy fashion: database with access library, main executable („engine“) including HTTPS server, node executables („operation engines“) to perform compute operations remotely, web-based human-machine interface (HMI) and application programming interface (API) library.
- Programming languages employed by WhizniumSBE include C++, SQL, XML and HTML5/SVG/JavaScript. Connectivity with third-party tools is facilitated by means of app-generation wizards for C++, C# and Objective-C which in turn use the project-specific C++ API library. Communication via industry-standard OPC UA is supported as well.
- To ensure maximum transparency, the generated code only relies on a minimum of external libraries (libxml2, libmicrohttpd and client libraries to the DBMS used), all of which are Open Source.
- WhizniumSBE places special emphasis on hassle-free intermixing of automatically generated and manually written code. On source code tree iteration, i.e. the process of letting WhizniumSBE generate an updated source code tree based on new model information, manually written code is preserved. Version control is strictly enforced, optionally using Git.
- Deployment options for WhizniumSBE include an on-premise container-based solution, or cloud-based / pay-per-use as an alternative.

## Runtime structure

Application development with WhizniumSBE results in the components depicted in Figure 1. From the beginning, they are available in compile-ready fashion through the automated code generation process.

Server-side a *combined engine*, written in C++, serves as the primary executable, taking the role of the controller in the model-view-controller (MVC) pattern. It has access to the application‘s MySQL/PgSQL/SQLite database and can handle client-side requests within its *job processor* threads. Opening client sessions and performing actions within results in a dynamic *job hierarchy* of C++ objects with dedicated *jobs* for sessions, user interface features and hardware control, respectively.

The *operation processor* threads are meant for the execution of atomic compute *operations*.

All communication with the *combined engine* is handled by means of context-specific XML data blocks. These can originate from the web-based UI which is provided in the form of HTML/XML/JavaScript files. Alternatively, XML block communication is made available through the API library that can be included in external C++ applications.

For debugging purposes, command-line access is possible. WhizniumSBE also implements an interface to a dedicated monitoring/logging tool, which itself is a WhizniumSBE-generated project. It optionally captures all changes to the *combined engine*‘s internal *job hierarchy* along with XML traffic, either in a log file or in a database.

Finally, an XML preferences file and a managed file archive provide for permanent storage.

In an alternative implementa-

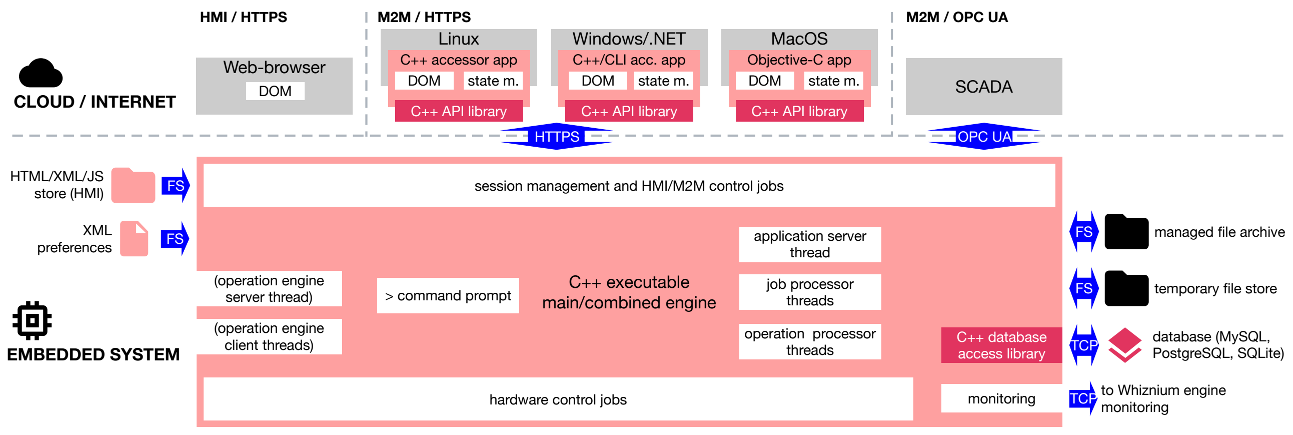


**Figure 1:** Representative situation of a WhizniumSBE-developed project at runtime

tion option of any WhizniumSBE project, based on the same source code, two different types of executable can be built: a single *engine* on one hand, and multiple *operation engines* on the other hand.

This configuration is intended for projects running in data centers: the *engine* maintains the role of managing the *job hierarchy*, while the multiple *operation engines* receive atomic compute task requesrs via HTTP/XML by the *engine*.

The *engine*'s *operation engine server* thread handles the registration of connected nodes running the *operation engine* executable. Within the *engine*, for each node, a dedicated *operation engine client* thread is established, allowing to construct highly scalable systems, in cloud environments. Database and file archive are shared between the executables via network.

## Model entry

WhizniumSBE currently uses a set of tab-separated text files as input for the application model. Header lines with specific keywords along with indentation allow for the specification of hierarchical structures. For example, table column definitions are entered line-by-line, indented by one tab, below the table definition line, starting with a corresponding header line. A total of eight input files, listed in Table 1, constitute the model. While these files mainly serve to enter information, retrieve/update/remove operarots also allow for the adaptation of elements generated automatically by WhizniumSBE.

Deployment information includes application *components*, *releases* (specific for each build target), third-party libraries and make file parameters.

Global features comprise *vectors* (multi-language enumerations) and XML data *blocks* (e.g. hardware settings to be stored in the preferences file) with application-wide validity.

The database structure is composed of *vectors* and user-extensible key lists of different types, main/auxiliary/relation/jumper tables with table columns, associated *vectors*, *load functions* (as wrappers for frequently used `SELECT` SQL statements) and boolean *checks* on table fields. Additional definitions include *relations* with sub-relations – from simple 1:N and M:N to sophisticated list functionality with insert/swap/remove operators ; in total, 39 types with sub-types are implemented. *Stubs* provide multi-language human-readable representations of records for manual implementation in code.

The basic UI structure consists of *cards* (each *card* is displayed

**Table 1:** Model files and their respective content

| Model component | Content |
|---|---|
| Deployment information | application components and build targets |
| Global features | vectors and data blocks for app-wide use |
| Database structure | tables, vectors, hierarchical relations and stubs |
| Basic UI structure | cards grouped by modules, adaptations of presettings |
| Import/export structure | external data import/export to/from groups of tables |
| Custom UI features | custom panels, dialogs, queries and presettings ; adaptations of the auto UI |
| Operation pack structure | ops grouped by op packs, invocation arguments /return values and squawks |
| Custom job tree features | custom jobs with stages, squawks, vectors and data blocks ; custom calls ; adaptations of the auto job tree |

in an individual web browser tab) based either on a main table or with custom functionality. *Modules* group *cards* by specific aspects of the application. Session-wide *presettings* allow for the meaningful pre-filtering of data in database table-backed *cards*.

The import/export structure specifies *import/export complexes* as patterns for text- or XML-based data exchange. Within each *import/export complex*, a hierarchical structure of *import/exports* defines the association of external data with database tables. The hierarchical structure is needed to express relations, as relations within the database are handled by numerical references which are not known externally. *Import/export column* definitions set the action to be performed at table field level (e.g. formatting).

Custom UI featues range from headbar/headline/list/form *panels*, wizard-style dialogs and controls with optional *feeds* (flexible data sources) to database query structures. For these, multiple table sources, ordering, filtering and display options are available. Additionally, non-standard *presettings* can be defined. The UI generated by WhizniumSBE based on database and basic UI structures can be adapted by adding/removing *panels*/controls or by cosmetical changes.

The operation pack structure outlines *operations* which atomically perform compute tasks based on a set (XML block) of *invocation arguments*, returning a set of *return values*. *Squawks* (multi-language strings with placeholders) describe in human-readable form the *operation*'s activity during execution. *Operations* are grouped by *operation packs* of similar functionality, e.g. requiring the same third-party libraries.

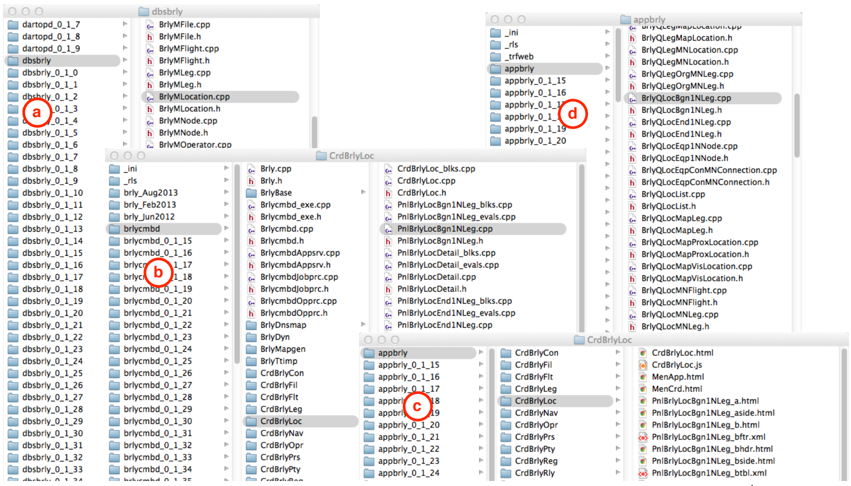

```
dbsbrly
dartopd_0_1_7      BrlyMFile.cpp
dartopd_0_1_8      BrlyMFile.h
dartopd_0_1_9      BrlyMFlight.cpp
dbsbrly            BrlyMFlight.h
dbsbrly_0_1_0      BrlyMLeg.cpp
dbsbrly_0_1_1      BrlyMLeg.h
dbsbrly_0_1_2      BrlyMLocation.cpp
_0_1_3             BrlyMLocation.h
_0_1_4             BrlyMNode.cpp
dbsbrly_0_1_5      BrlyMNode.h
dbsbrly_0_1_6      BrlyMOperator.cpp
dbsbrly_0_1_7
dbsbrly_0_1_8
dbsbrly_0_1_9
dbsbrly_0_1_10
dbsbrly_0_1_11
dbsbrly_0_1_12
dbsbrly_0_1_13
dbsbrly_0_1_14
dbsbrly_0_1_15
dbsbrly_0_1_16
dbsbrly_0_1_17
dbsbrly_0_1_18
dbsbrly_0_1_19
dbsbrly_0_1_20
dbsbrly_0_1_21
dbsbrly_0_1_22
dbsbrly_0_1_23
dbsbrly_0_1_24
dbsbrly_0_1_25
dbsbrly_0_1_26
dbsbrly_0_1_27
dbsbrly_0_1_28
dbsbrly_0_1_29
dbsbrly_0_1_30
dbsbrly_0_1_31
dbsbrly_0_1_32
dbsbrly_0_1_33
```
```
CrdBrlyLoc
_ini               Brly.cpp               CrdBrlyLoc_blks.cpp
_rls               Brly.h                 CrdBrlyLoc.cpp
brly_Aug2013       BrlyBase               CrdBrlyLoc.h
brly_Feb2013       Brlycmbd_exe.cpp       PnlBrlyLocBgn1NLeg_blks.cpp
brly_Jun2012       Brlycmbd_exe.h         PnlBrlyLocBgn1NLeg_evals.cpp
brlycmbd           Brlycmbd.cpp           PnlBrlyLocBgn1NLeg.cpp
brlycmbd_0_1_15    Brlycmbd.h             PnlBrlyLocBgn1NLeg.h
brlycmbd_0_1_16    BrlycmbdAppsrv.cpp     PnlBrlyLocDetail_blks.cpp
brlyc     1_17     BrlycmbdAppsrv.h       PnlBrlyLocDetail_evals.cpp
brlyc     1_18     BrlycmbdJobprc.cpp     PnlBrlyLocDetail.cpp
brlycmbd_0_1_19    BrlycmbdJobprc.h       PnlBrlyLocDetail.h
brlycmbd_0_1_20    BrlycmbdOpprc.cpp      PnlBrlyLocEnd1NLeg_blks.cpp
brlycmbd_0_1_21    BrlycmbdOpprc.h        PnlBrlyLocEnd1NLeg_evals.cpp
brlycmbd_0_1_22    BrlyDnsmap             PnlBrlyLocEnd1NLeg.cpp
brlycmbd_0_1_23    BrlyDyn
brlycmbd_0_1_24    BrlyMapgen
brlycmbd_0_1_25    BrlyTtimp
brlycmbd_0_1_26    CrdBrlyCon
brlycmbd_0_1_27    CrdBrlyFil
brlycmbd_0_1_28    CrdBrlyFlt
brlycmbd_0_1_29    CrdBrlyLeg
brlycmbd_0_1_30    CrdBrlyLoc
brlycmbd_0_1_31    CrdBrlyNav
brlycmbd_0_1_32    CrdBrlyOpr
brlycmbd_0_1_33    CrdBrlyPrs
brlycmbd_0_1_34    CrdBrlyPty
```
```
appbrly
_ini               BrlyQLegMapLocation.h
_rls               BrlyQLegMNLocation.cpp
_trfweb            BrlyQLegMNLocation.h
appbrly            BrlyQLegOrgMNLeg.cpp
appbrly_0_1_15     BrlyQLegOrgMNLeg.h
appbrly_0_1_16     BrlyQLocBgn1NLeg.cpp
appbrly_0_1_1      BrlyQLocBgn1NLeg.h
appbrly_0_1_       BrlyQLocEnd1NLeg.cpp
appbrly_0_1_19     BrlyQLocEnd1NLeg.h
appbrly_0_1_20     BrlyQLocEqp1NNode.cpp
                   BrlyQLocEqp1NNode.h
                   BrlyQLocEqpConMNConnection.cpp
                   BrlyQLocEqpConMNConnection.h
                   BrlyQLocList.cpp
                   BrlyQLocList.h
                   BrlyQLocMapLeg.cpp
                   BrlyQLocMapLeg.h
                   BrlyQLocMapProxLocation.cpp
                   BrlyQLocMapProxLocation.h
                   BrlyQLocMapVisLocation.cpp
                   BrlyQLocMapVisLocation.h
                   BrlyQLocMNFlight.cpp
                   BrlyQLocMNFlight.h
                   BrlyQLocMNLeg.cpp
                   BrlyQLocMNLeg.h
```
```
CrdBrlyLoc
appbrly            CrdBrlyCon     CrdBrlyLoc.html
appbrly_0_1_15     CrdBrlyFil     CrdBrlyLoc.js
appbrly_0_1_16     CrdBrlyFlt     MenApp.html
appbrly_0_1_17     CrdBrlyLeg     MenCrd.html
appbrl      18     CrdBrlyLoc     PnlBrlyLocBgn1NLeg_a.html
appbrl      19     CrdBrlyNav     PnlBrlyLocBgn1NLeg_aside.html
appbrly_0_1_20     CrdBrlyOpr     PnlBrlyLocBgn1NLeg_b.html
appbrly_0_1_21     CrdBrlyPrs     PnlBrlyLocBgn1NLeg_bftr.xml
appbrly_0_1_22     CrdBrlyPty     PnlBrlyLocBgn1NLeg_bhdr.html
appbrly_0_1_23     CrdBrlyReg     PnlBrlyLocBgn1NLeg_bside.html
appbrly_0_1_24     CrdBrlyRly     PnlBrlyLocBgn1NLeg_btbl.xml
```

**Figure 2:** Exemplary source code trees with naming conventions for a) database access library, b) engine/operation engine, c) web-based UI, d) API

Custom job tree features encompass adaptations to the job hierarchy generated by WhizniumSBE based on the UI. *Jobs* can be added and complemented with *stages* (for state machines), *squawks*, command-line commands, specific *vectors* and data blocks. Finally, *jobs* can be configured to trigger and/or listen to *calls* which serve for message passing within the job hierarchy.

## Source code

With WhizniumSBE, automated code generation is used consistently throughout the entire project lifetime cycle. This results in a very clean source code base, as illustrated in Figure 2. Meaningful naming conventions, designed carefully as the best trade-off between clarity, uniqueness and brevity, ensure fast familiarization for development teams. Examples include `TblBrlyMLocation` for a main ("M") table ("Tbl") of locations within the BeamRelay ("Brly") project and `PnlBrlyLocBgn1NLeg` for a *panel* ("Pnl") displaying the 1:N ("1N") relation of a location (airport) and legs (routes) beginning there as part of the location ("Loc") *card*. The *operation* `BrlyMapgenConmap` produces a dynamic map of a relay connection as part of the *operation pack* `BrlyMapgen` for map generation.

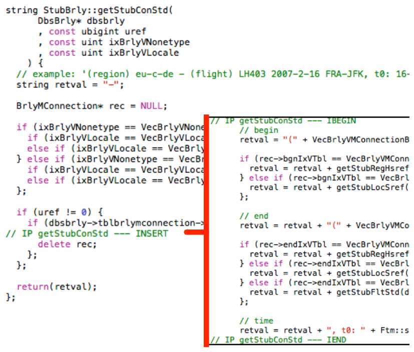
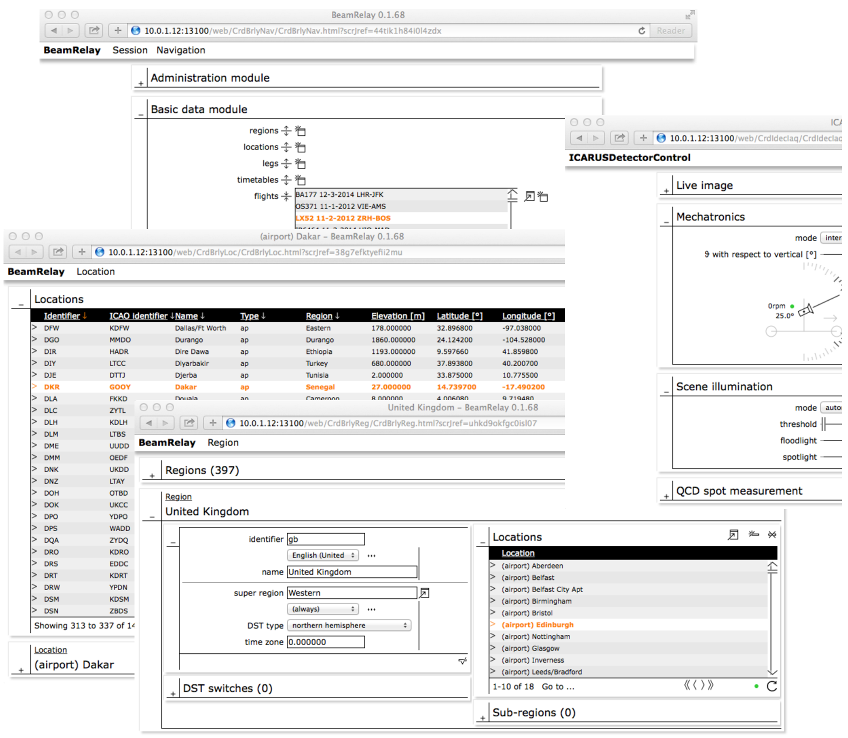

```
string StubBrly::getStubConStd(
      DbsBrly* dbsbrly
      , const ubigint uref
      , const uint ixBrlyVNonetype
      , const uint ixBrlyVLocale
    ) {
  // example: '(region) eu-c-de - (flight) LH403 2007-2-16 FRA-JFK, t0: 16-
  string retval = "-";

  BrlyMConnection* rec = NULL;

  if (ixBrlyVNonetype == VecBrlyVNone
    if (ixBrlyVLocale == VecBrlyVLoca
    else if (ixBrlyVLocale == VecBrly
  } else if (ixBrlyVNonetype == VecBr
    if (ixBrlyVLocale == VecBrlyVLoca
    else if (ixBrlyVLocale == VecBrly
  };

  if (uref != 0) {
    if (dbsbrly->tblbrlymconnection->
// IP getStubConStd --- INSERT
      delete rec;
    };
  };

  return(retval);
};
```
```
// IP getStubConStd --- IBEGIN
      // begin
      retval = "(" + VecBrlyVMConnectionB

      if (rec->bgnIxVTbl == VecBrlyVMConn
        retval = retval + getStubRegHsref
      } else if (rec->bgnIxVTbl == VecBrl
        retval = retval + getStubLocSref(
      };

      // end
      retval = retval + "(" + VecBrlyVMCo

      if (rec->endIxVTbl == VecBrlyVMConn
        retval = retval + getStubRegHsref
      } else if (rec->endIxVTbl == VecBrl
        retval = retval + getStubLocSref(
      } else if (rec->endIxVTbl == VecBrl
        retval = retval + getStubFltStd(d
      };

      // time
      retval = retval + ", t0: " + Ftm::s
// IP getStubConStd --- IEND
```

**Figure 3:** Insertion point in C++

An important aspect of code generation is the harmonic co-existence of automatically generated code and manually written code. In WhizniumSBE projects, this concerns the sources for the *engine/operation engine* and for the web-based UI. Generally, all source code files generated by WhizniumSBE can be edited to contain custom code at predefined *insertion points*. Figure 3 outlines this for the manual implementation of a *stub*.

*Insertion points* are comments in the respective programming language. Their full advantage is revealed on version stepping or source code tree iteration, which is required every time the model is updated. In the process, WhizniumSBE scans the existing sources for edited *insertion point* content, generates new sources based on the updated model and re-inserts manual code found previously. Besides the `INSERT/IBEGIN/`



BeamRelay 0.1.68
10.0.1.12:13100/web/CrdBrlyNav/CrdBrlyNav.html?scrJref=44tik1h84i0l4zdx

**BeamRelay** Session Navigation

Administration module

Basic data module

regions
locations
legs
timetables
flights: BA177 12-3-2014 LHR-JFK / OS371 11-1-2012 VIE-AMS / **LX52 11-2-2012 ZRH-BOS**

(airport) Dakar – BeamRelay 0.1.68
10.0.1.12:13100/web/CrdBrlyLoc/CrdBrlyLoc.html?scrJref=38g7efktyefii2mu

**BeamRelay** Location

Locations

| Identifier | ICAO identifier | Name | Type | Region | Elevation [m] | Latitude [°] | Longitude [°] |
|---|---|---|---|---|---|---|---|
| DFW | KDFW | Dallas/Ft Worth | ap | Eastern | 178.000000 | 32.896800 | -97.038000 |
| DGO | MMDO | Durango | ap | Durango | 1860.000000 | 24.124200 | -104.528000 |
| DIR | HADR | Dire Dawa | ap | Ethiopia | 1193.000000 | 9.597660 | 41.859800 |
| DIY | LTCC | Diyarbakir | ap | Turkey | 680.000000 | 37.893800 | 40.200700 |
| DJE | DTTJ | Djerba | ap | Tunisia | 2.000000 | 33.875000 | 10.775500 |
| **DKR** | **GOOY** | **Dakar** | **ap** | **Senegal** | **27.000000** | **14.739700** | **-17.490200** |
| DLA | FKKD | Douala | ap | Cameroon | 8.000000 | 4.006080 | 9.719480 |
| DLC | ZYTL | | | | | | |
| DLH | KDLH | | | | | | |
| DLM | LTBS | | | | | | |
| DME | UUDD | | | | | | |
| DMM | OEDF | | | | | | |
| DNK | UKDD | | | | | | |
| DNZ | LTAY | | | | | | |
| DOH | OTBD | | | | | | |
| DOK | UKCC | | | | | | |
| DPO | YDPO | | | | | | |
| DPS | WADD | | | | | | |
| DQA | ZYDQ | | | | | | |
| DRO | KDRO | | | | | | |
| DRS | EDDC | | | | | | |
| DRT | KDRT | | | | | | |
| DRW | YPDN | | | | | | |
| DSM | KDSM | | | | | | |
| DSN | ZBDS | | | | | | |

Showing 313 to 337 of 14

Location
(airport) Dakar

United Kingdom – BeamRelay 0.1.68
10.0.1.12:13100/web/CrdBrlyReg/CrdBrlyReg.html?scrJref=uhkd9okfgc0isl07

**BeamRelay** Region

Regions (397)

Region
United Kingdom

identifier: gb
English (United
name: United Kingdom
super region: Western
(always)
DST type: northern hemisphere
time zone: 0.000000

DST switches (0)

Locations

| Location |
|---|
| (airport) Aberdeen |
| (airport) Belfast |
| (airport) Belfast City Apt |
| (airport) Birmingham |
| (airport) Bristol |
| **(airport) Edinburgh** |
| (airport) Nottingham |
| (airport) Glasgow |
| (airport) Inverness |
| (airport) Leeds/Bradford |

1-10 of 18 Go to ...

Sub-regions (0)

10.0.1.12:13100/web/CrdIdeclaq/CrdIdeclaq

**ICARUSDetectorControl**

Live image

Mechatronics

mode: inter
ϑ with respect to vertical [°]
0rpm
25.0°

Scene illumination

mode: auto
threshold
floodlight
spotlight

QCD spot measurement

**Figure 4:** Web-based UI examples ; auto-generated with the exception of a custom SVG control in the "Mechatronics" panel on the right hand side

IEND directives, BEGIN/END vs. RBEGIN/REND tokens allow for the replacement of code generated automatically. The *insertion point* functionality is complemented by an in-file directive to copy a file on version stepping and by a key file name to retain an entire folder.

## Web-based user interface

Standard features of the multi-language web-based UI, depicted in Figure 4, include a login screen and a *navigation card* with access to all other *cards*, grouped by *modules*. A record access history is included as well. Access rights can be defined on *card* and/or record level, both for individual users and user groups.

While custom *cards*, e.g. for hardware control, allow for freely definable *panels*, all database table-backed *cards* feature the same layout: the upper half is occupied by a list of records while the lower half shows record-specific data. Within the lower half, record manipulation and custom views are available on the left hand side. In particular, the *detail panel* offers a multitude of fine-grained display and editing options, depending on the nature of the table columns and relations concerned. The *panels* on the right hand side serve as the link to other *cards*, as established by underlying 1:N and M:N relations.

## API access

The philosophy behind API access to a WhizniumSBE *combined engine* for machine-to-machine (M2M) communication is that every view that is present in the web-based UI and every action that can be triggered e.g. by clicking, has an API library equivalent that exchanges the same XML data blocks as the web browser.

This method ensures that all access is authenticated, as the first step of API interaction is to start a session (login screen equivalent).

To effectively make use of the API library, it is also required to maintain a copy of certain XML data blocks received client-side – in the web-browser this is handld by the Document Object Model (DOM). Further, typical tasks, such as waiting for a record in a view and then add new sub-records to it, require multiple sequential interactions with the *combined engine*.

In M2M, the former can be replaced by a set of C++ runtime objects, and the latter can be expressed as an event-driven state machine.

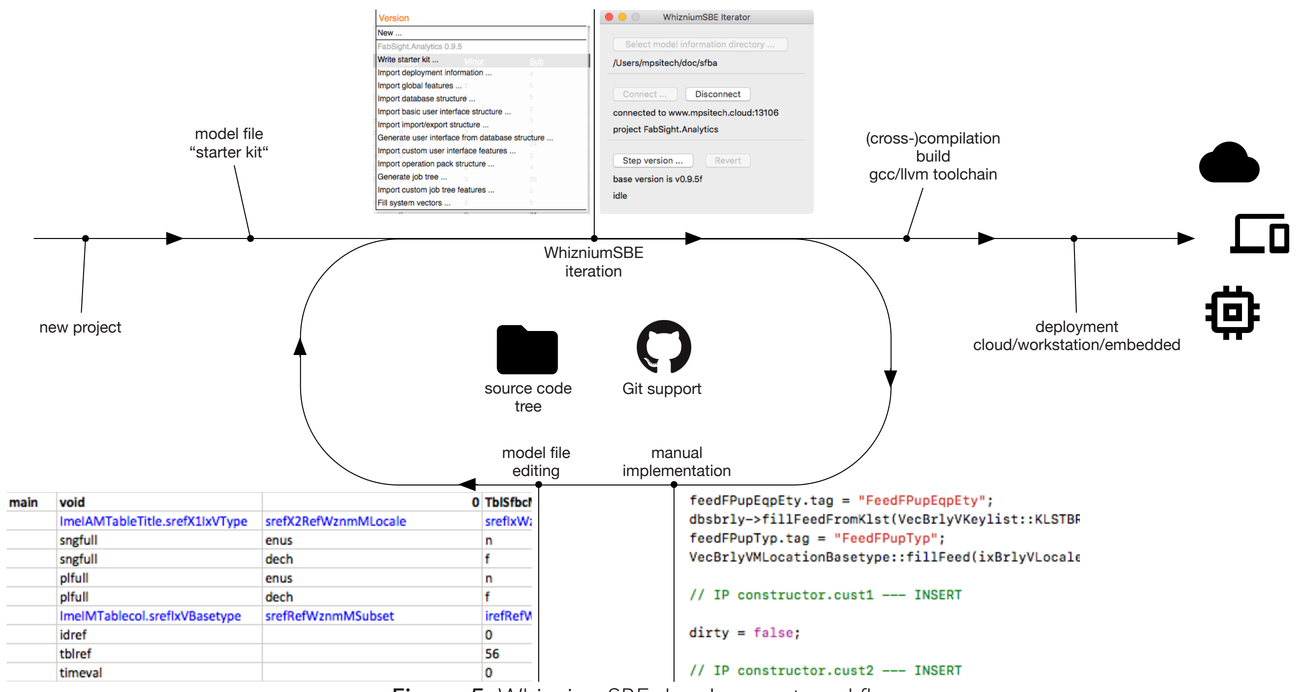
The accessor app generation feature in WhizniumSBE helps to write the code for this scenario with the input specified in the form of a separate, accessor app specific, model file.

It provides code generation for integration into native Linux, Windows .NET and MacOS projects by using the respective C flavors. The delivered code also comes with the required platform-specific HTTPS communication methods.

## Development workflow

The full project workflow is illustrated in Figure 5. Its main cycle consists of manually editing the source code tree, adapting the model files, and feeding updated versions of both into WhizniumSBE, which in turn establishes the next iteration of the project. WhizniumSBE features built-in Git synchronization with remote repositories.

It is noteworthy that WhizniumSBE itself is a WhizniumSBE developed service. As a result, its web-based UI features can be accessed via API, as evidenced in the top center of Figure 5: a project iteration requires multiple model file uploads, clicks and then finally, source code downloads. For convenience, all this functionality was integrated into a native MacOS tool, WhizniumSBE Iterator, which allows version stepping in six clicks.


new project
model file
“starter kit“
Version
New ...
FabSight.Analytics 0.9.5
Write starter kit ...
Import deployment information ...
Import global features ...
Import database structure ...
Import basic user interface structure ...
Import import/export structure ...
Generate user interface from database structure ...
Import custom user interface features ...
Import operation pack structure ...
Generate job tree ...
Import custom job tree features ...
Fill system vectors ...
WhizniumSBE Iterator
Select model information directory ...
/Users/mpsitech/doc/sfba
Connect ...
Disconnect
connected to www.mpsitech.cloud:13106
project FabSight.Analytics
Step version ...
Revert
base version is v0.9.5f
idle
WhizniumSBE
iteration
(cross-)compilation
build
gcc/llvm toolchain
deployment
cloud/workstation/embedded
source code
tree
Git support
model file
editing
manual
implementation

| main | void | 0 | TblSfbcl |
|---|---|---|---|
| | ImeIAMTableTitle.srefX1IxVType | srefX2RefWznmMLocale | sreflxW |
| | sngfull | enus | n |
| | sngfull | dech | f |
| | plfull | enus | n |
| | plfull | dech | f |
| | ImeIMTablecol.sreflxVBasetype | srefRefWznmMSubset | irefRefV |
| | idref | | 0 |
| | tblref | | 56 |
| | timeval | | 0 |

```
feedFPupEqpEty.tag = "FeedFPupEqpEty";
dbsbrly->fillFeedFromKlst(VecBrlyVKeylist::KLSTBR
feedFPupTyp.tag = "FeedFPupTyp";
VecBrlyVMLocationBasetype::fillFeed(ixBrlyVLocale

// IP constructor.cust1 --- INSERT

dirty = false;

// IP constructor.cust2 --- INSERT
```

**Figure 5:** WhizniumSBE development workflow

## More information

- The ICARUS detector, 2016 – control software for a complex embedded system with visible and infrared image data processing capabilities.
- The FabSight project, MPSI Technologies 2018 – an IIoT Whiznium example from chip level to cloud-based data collection with muliple uses of the API functionality.
- The BeamRelay project, 2014 – a sophisticated simulation of commercial air traffic using distributed computing in the cloud.

MPSI TECHNOLOGIES

contact@mpsitechnologies.com
+49 175 918 5480
www.mpsitechnologies.com

Helene-Mayer-Ring 4, 04.19
80809 Munich
Germany