Elektrobit

Software architectural design document

Version: 0.1.8

Status: Draft

Release date: 2019-11-29

Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany
Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333
Email: info.automotive@elektrobit.com

Technical support

Support Phone	Support email	Support Website
Phone: +49 (9131) 7701-7722	Email: support.testlab@elektrobit.com	http://www.elektrobit.com/support

Legal disclaimer

Confidential and proprietary information.

Document history

Version	Date	Description	Editor
0.1.0	2019-11-06	Initial	Steffen Pankratz
0.1.1	2019-11-12	Web GUI documentation added	Johannes Förstner
0.1.2	2019-11-13	Scene REST API documentation added	Sebastian Böttigheimer
0.1.3	2019-11-13	Upload Client documentation added	Sebastian Böttigheimer
0.1.4	2019-11-13	IaC documentation added	Johannes Förstner
0.1.5	2019-11-13	Test Lab REST API documentation added	Rohit Prashar
0.1.6	2019-11-13	Analyzer batch node documentation added	Daniela Morales Tolentino Zang
0.1.7	2019-11-25	Versioning chapter added	Johannes Förstner
0.1.8	2019-11-29	Analyzer and Mobile App chapters added	Johannes Förstner

Table of Contents

1. Context view

This chapter shall give a rough overview about the functionality of the described software system in the system context.

EB Assist Test Lab is intended to be used for ingesting, analyzing and maintaining both virtual and real test drive data in a cloud-based environment. EB Assist Test Lab consists of a large variety of different software components.

Core software components:

- Analyzer
- Analyzer batch node scripts
- Android Mobile App
- Test Lab REST API
- Scene REST API
- Upload Client
- Web GUI

Internal software components for development:

- IaC (build and deploy scripts)

1.1. Software system context

Template instantiation instructions
Use this chapter to describe the whole system and illustrate how the software system described by this SWAD is placed into the overall system. It is recommended to use an *UML component diagram* or *deployment diagram* to visualize the context of the software system. A context can consists of other software system, 3 rd party components and also real users of the system.

1.2. Functional overview

This chapter describes main functionality of the software system.

Template instantiation instructions
Use this chapter to describe the main functionality that shall be provided by the software system. - In case the description of each function is sufficiently long, create a dedicated section for each function.

1.3. Environment

This chapter describes the environment of the software system (for example operating system, programming language, frameworks).

Template instantiation instructions
Use this chapter to declare: - which programming languages and language features may be used - which OS interfaces may be used - what external libraries or frameworks will be used or are allowed to be used - relevant open source modules - binary types - like 32bit/64bit

1.4. Configuration and application parameters

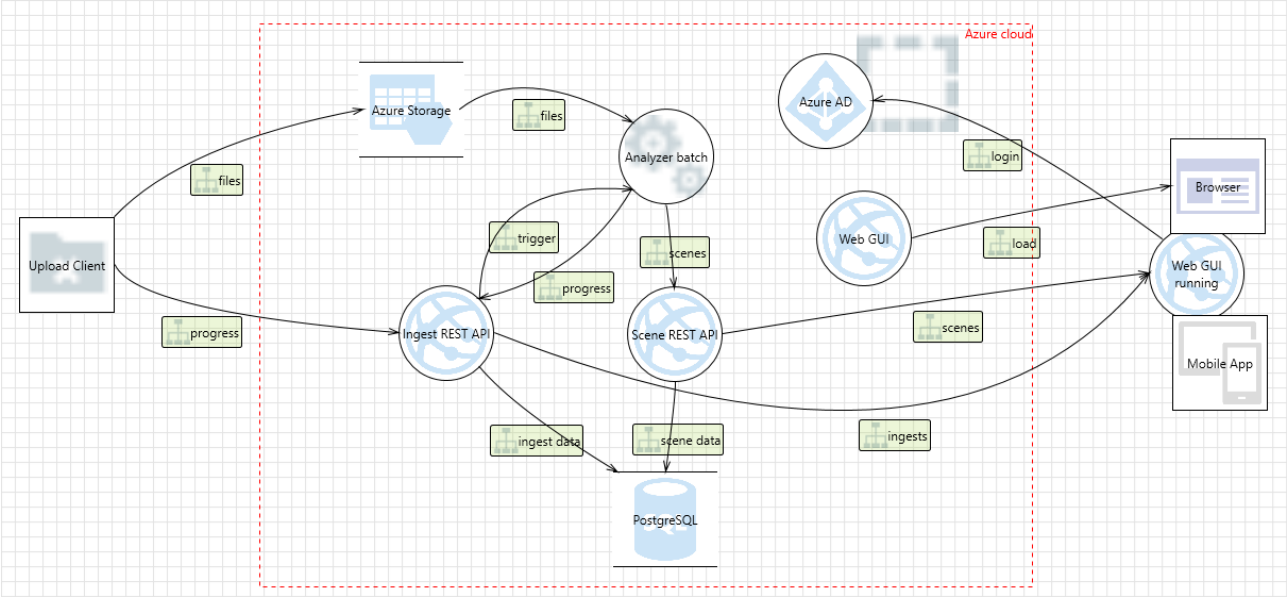
Template instantiation instructions
If applicable: Define a list of possible configuration parameters, a description of the variabilities they introduce and their possible value ranges.

Table 1.1. Configuration Parameters

Parameter	**Description**	**Value Range**
...	...	...

2. Cloud architecture

2.1. Main actors & actions



This diagram shows the main workflows of a running Test Lab.

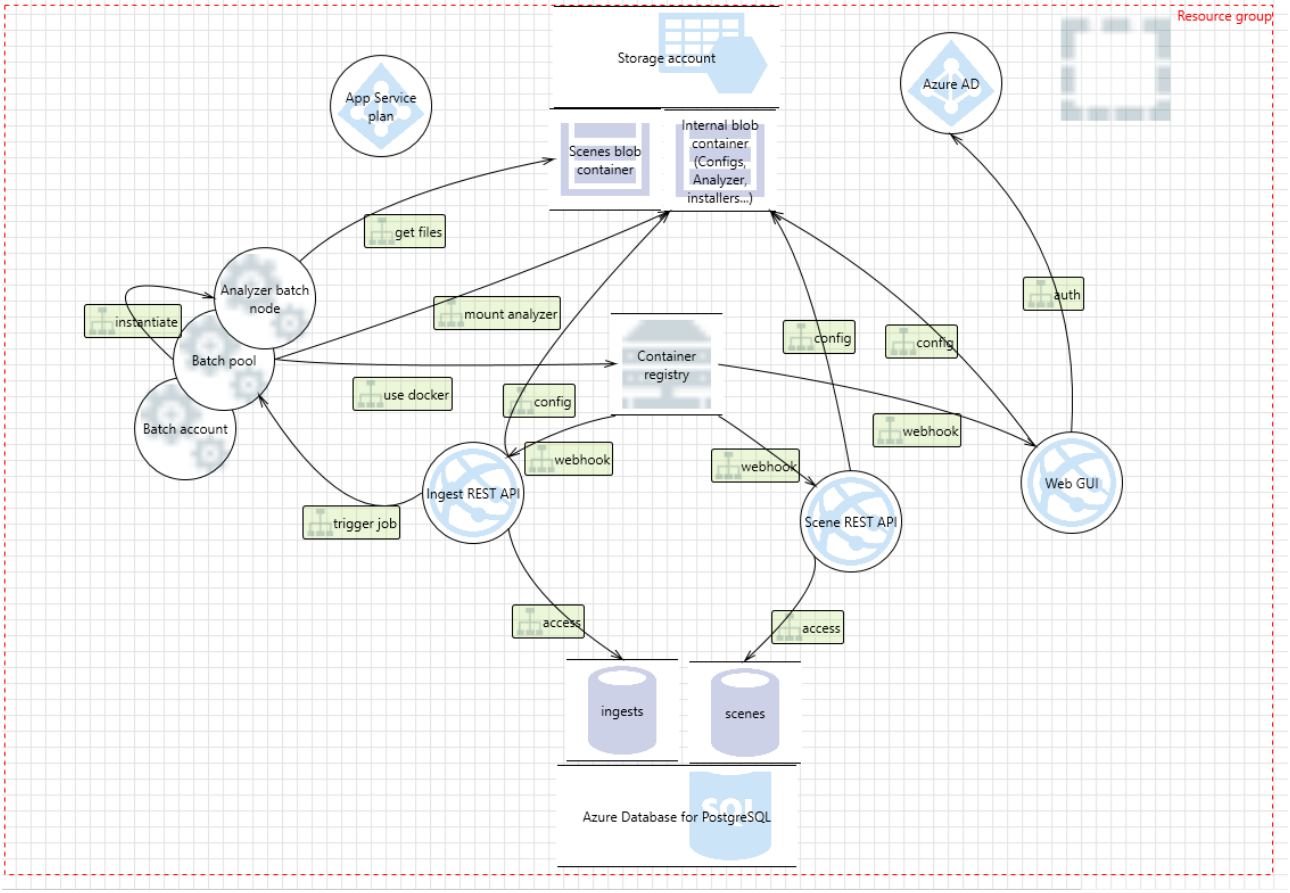
The Upload Client uploads scene files into the blob storage container and reports the progress to the Test Lab REST API.

The Test Lab REST API triggers analysis of new scenes in the Analyzer batch node, which loads the scene files from the blob storage container, writes the scene data into the Scene REST API and reports the progress back to the Test Lab REST API.

The Web GUI is loaded into a browser (or running in a Mobile App) and logs in with Azure Active Directory.

The Scene REST API and Test Lab REST API provide their database content for display in the Web GUI.

2.2. Resources & dependencies



This diagram provides an overview of all cloud resources involved in the Test Lab and their interdependencies.

The PostgreSQL server hosts the databases for the Scene REST API and Test Lab REST API.

The Storage account has separate blob storage containers for customer data (scenes) and Test Lab internal data.

All App Services are running on an App Service plan.

The Container registry holds the docker images of the Web GUI, Scene REST API and Test Lab REST API, and also of the Analyzer batch node.

The docker images of the Web GUI, Scene REST API and Test Lab REST API are loaded into their App Services via webhooks.

The deployment-specific configurations of the App Services are held in the internal blob storage container.

To trigger an analysis, the Test Lab REST API triggers a job in the batch pool which is hosted by the Batch account.

The analyzer job instantiates the Analyzer batch node with the docker image from the Container registry and mounts the Analyzer binaries & script from the internal blob storage container. The running Analyzer batch node then loads the scene files from the customer blob storage container for analysis.

The Web GUI relies on Azure Active Directory for authentication.

3. Versioning

3.1. Test Lab product version

Test Lab uses the commonly known semantic versioning scheme of "major.minor.patch" version numbers.

Additionally, a build number can be attached. This number is uniquely generated by the build process (e.g. Jenkins job number) and can be linked to the git revisions of all contained repositories (e.g. from Jenkins job log file).

The Test Lab version is used for communicating the kind of change to the customer. It is therefore defined by Product Management / Product Order.

They are expected to be incremented & released about once a quarter, at most once a sprint.

A Test Lab version is defined by the list of component versions.

3.2. Component versions

Test Lab components all have their own version number, which also uses semantic versioning.

These versions are used internally to communicate changes within the team and are therefore defined by the team.

They can be incremented & released anytime; however the developers will avoid multiple releases within a sprint, because it would probably create too much friction.

They are displayed as detail information on the About Page, but mainly for support purposes; the customer is not expected to be interested in them.

The versions are used to keep track of important changes, and to define incompatibilities.

4. Scene REST API

This chapter shall give a rough overview about the functionality of the Scene REST API.

The Scene REST API exposes multiple interfaces over HTTP to scenes and campaigns which are stored in a database. These interfaces are used by clients such as the Web GUI. The API is written in PHP and is served by an nginx web server inside a docker container.

4.1. Component dependencies

To function, the Scene REST API needs a running PostgreSQL database server with a set up database. Data is retrieved from this database and provided in JSON format over the API's interfaces. Furthermore, it stores data it receives over the interfaces into the database.

4.2. Configuration parameters

The Scene REST API needs the following parameters to work properly. They are stored inside a configuration JSON file which is located in a blob storage container. An environment variable specifies the path to this configuration file which is then mounted into the docker container.

parameter name	**data type**	**description**
host	string	database server (host)
user	string	database user name
password	string	database user password
database	string	database name

4.3. Mounted files

The following files have to be mounted from a blob storage container into the docker container. The PHP script of the Scene REST API needs to have write access for the license and the signature files.

file name	**description**
license.lic	Test Lab license file
signature.sig	Test Lab license signature file
config.json	JSON file containing the configuration parameters from section 4.2, “Configuration parameters”

4.4. Involved cloud services

Some of the Scene REST API's cloud resources are shared with other Test Lab components. They do not need to be created new when deploying or updating this component. Already existing resources can be shared and reused. Other resources are component specific and are therefore only used by the Scene REST API itself.

resource	**type**
Resource group	shared
Container registry	shared
Storage account	shared
App Service plan	shared
App Service	component specific
Container registry webhook	component specific

4.5. Interfaces

The Scene REST API exposes a lot of interfaces over HTTP/HTTPS. All interfaces (except getApiVersion) are protected by JSON Web Tokens (Bearer authentication) and only respond with a valid Test Lab license. They also can be accessed with any HTTP request type other than OPTION. The input parameters have to be submitted in the query string, the responses are located in the response body in JSON format. The API includes a Swagger UI, describing all the interfaces in detail.

5. Web GUI

5.1. Overview

The Web GUI is the main interface of the Test Lab for its users.

Here, they can:

- Browse the scenes, incl. search and details
- Create campaigns to request missing scenes
- View the status of incoming scenes
- Upload scenes from browser
- View open campaigns to record missing scenes
- View registered recording devices
- Register notifications for finished uploads & finished ingestions

Access to the Web GUI is protected by authentication (except public pages).

5.2. Architecture

The Web GUI is programmed in TypeScript using the Angular framework.

The compiled distribution is served by a webserver.

It is deployed in the Azure cloud as a docker that runs in an App Service.

Furthermore, it is part of / compiled into the Mobile App.

Internally, there are separate Angular modules & components for every page, and also components used across multiple pages.

For communicating with the APIs, there are services, split into lower-level restapi services and higher-level data services.

Used libraries include: leaflet, ngx-translate, bootstrap, cordova, d3, mapbox-gl.

5.3. Involved Cloud Services

- App Service
- App Service plan
- Container registry
- Container registry webhook
- Azure Active Directory
- Storage account
- Resource group

5.4. Interfaces

The Web GUI interacts with other components in the following ways:

- AAD (Azure Active Directory) authentication: for login (using msal-angular)
- Scene REST API: for browsing scenes & campaigns
- Test Lab REST API: for viewing incoming scenes (polling) and registered devices; for accessing files in storage (download scene files, download installers of Mobile App & Upload Client)
- External: OSM & mapbox: for displaying GPS information

The Web GUI gets compiled into the Mobile App.

6. Upload Client

This chapter shall give a rough overview about the functionality of the Upload Client.

The Upload Client is used to ingest files from a folder into Test Lab. It uses the same Test Lab REST API as the Web GUI to report the progress of uploading files. The client is written in Python and is provided as a Microsoft Windows installer including all dependencies.

6.1. Component dependencies

To function, the Upload Client only needs a running Test Lab REST API. It is used to request SAS tokens and report the file upload progress to the blob storage container. During the Upload Client installation, the user has to provide the URI to the Test Lab REST API and the folder containing the scene files. After they are validated, they both are stored into a configuration file.

6.2. Configuration parameters

The Upload Client needs the following parameters to work properly. They are filled during installation and stored inside a configuration JSON file at the installation destination folder.

parameter name	**data type**	**description**
rest_api_service_url	string	Test Lab REST API URI (with protocol but without path)
upload_file_directory	string	absolute path to the folder containing scene files (with escaped backslashes)
max_nr_of_parallel_uploads	integer	number of parallel uploads
delete_files_after_upload	boolean	if the files shall be deleted after a successful upload
max_nr_of_connections	integer	number of connections per file upload

6.3. Involved cloud services

The Upload Client only requires at least one storage account to upload files to.

resource	**type**
Resource group	shared
Storage account	shared

6.4. Interfaces

The Upload Client only communicates with the Test Lab REST API and the Storage account over HTTP requests.

7. Test Lab REST API

7.1. Overview

The Test Lab REST API in the Test Lab is used to keep track of all the current scene ingestions. The Test Lab REST API provides SAS (shared access signature) token which client uses to upload scene data to the storage account, each upload is registered by using the client id. When uploads are started, it keeps informing the web gui and the mobile app regarding current upload status. Moreover, all upload related information is stored by the Test Lab REST API in the database. When all uploads are finished it triggers analyzer job and save all the progress in the database.

7.2. Test Lab REST API Architecture

Test Lab REST API is written in Python 3.6. It is packed as docker image served as docker container on the Microsoft Azure cloud. The azure web app service serves the docker container.

The external libraries which are used in the ingest are as follows:

- flask: Micro web framework in Python.
- azure-storage-blob: Microsoft's object storage library for Azure cloud.
- flask-cors: Flask extension for handling the Cross Origin Resource Sharing (CORS)
- flask-jwt-extended: Flask extension for json web tokens
- sqlalchemy: Library for databases and ython interface.
- psysopg2: Library for database integration
- pyfcm: For firebase cloud messaging
- pyjwt: Library for encoding and decoding JSON Web Tokens (JWT)
- requests: Used for making the HTTP requests
- pyqrcode: Module for generating the QR code
- jinja2: Web template engine for Python

7.3. Cloud Services

The cloud services used for the Test Lab REST API are listed below:

Table 7.1. Cloud resources

Azure Resources	
Resource group	Container for holding the Test Lab REST API resources
App Service plan	Plan for app service
App Service	For serving the docker container
Azure Active Directory	For Authentication
Batch account	For creating the batch jobs
PostgreSQL Database	Used to store all the information regarding uploads
Container registry	For storing the Test Lab REST API docker images
Container registry web-hook	For deploying the docker images from container registry to the web app

7.4. Interfaces

The Test Lab REST API interfaces with the other software components are described as follows:

- Analyzer: For creating batch jobs for analyzer and monitoring its work progress
- Ingest database: For updating the scene related information and analyzer progress
- Upload Client: For providing SAS (shared access signature) token to upload scene data and monitoring uploading activity
- SMTP Server: For sending the emails regarding uploads
- Google Firebase: For sending the notification on the Test Lab mobile app
- Web GUI: Provides the information regarding current uploads and ingestions

8. Analyzer batch node

8.1. Overview

In order to process data after file upload, the Test Lab REST API uses an Batch account to run Docker containers with the Analyzer app in parallel.

In this context, the behavior of each node is defined by the module Analyzer batch node.

8.2. Architecture

The Analyzer batch node runs inside a Docker container in a Azure batch node. It is implemented in Python and includes modules for the following tasks:

- Run the Analyzer for "scene files" (i.e. data files, with the possibility of running for multiple, split-scene files) and all related "analyze files" (annoxml, xml etc.) iteratively.
- Write scenes, label groups and labels provided by Analyzer to Scene REST API
- Write the progress of the Analyzer file processing to Test Lab REST API

8.3. Component dependencies

To function, the Analyzer batch node needs the Scene REST API and Test Lab REST API to be running and working.

8.4. Configuration parameters

The Analyzer batch node needs the parameters below to work properly. These are provided as environment variables.

parameter name	data type	description
INGEST_REST_API_URL	string	URI of Test Lab REST API
SCENE_REST_API_URL	string	URI of Scene REST API
SCENE_FILE_URI	string	URL to "scene file"
ANALYZE_FILE_URI_LIST_STRING	string	URL to "analyze files"

parameter name	data type	description
UPLOAD_ID	integer	Upload ID
AZ_BATCH_TASK_WORKING_DIR	string	Path to the Azure batch nodes working directory

8.5. Involved cloud services

The Analyzer batch node requires one storage account to download Analyzer binaries from.

resource	type
Resource group	shared
Storage account	shared
Batch account	shared
Container registry	shared

8.6. Interfaces

The Analyzer batch node is executed by shipyard, which is called by the Test Lab REST API. It communicates to Scene REST API and Test Lab REST API.

9. Analyzer

9.1. Overview

The Analyzer is the binary that processes the scene files.

It is written in C++.

It has a plug-in architecture so that customers can write extensions.

More information is available in the documentation of this component.

10. Mobile App

10.1. Overview

The Mobile App is an app for mobile devices that encapsulates the Web GUI.

It is written using Cordova, which enables us to develop OS independently. Currently Android is supported.

The Web GUI is using responsive design down to mobile screen resolutions.

In a few places, the Web GUI has Mobile App specific code that is activated by comment-switching during build.

The Mobile App provides additional functionality like QR code scanning.

11. Infrastructure as Code

11.1. Overview

The IaC (Infrastructure as Code) sets up the cloud infrastructure with the Test Lab components.

It allows clean installation as well as upgrading an existing deployment.

It can install/upgrade the whole Test Lab product as well as single components.

11.2. Architecture

The IaC is a script written in Python.

The Test Lab components need to be available pre-built.

IaC uses terraform to create and alter the resources in the Azure cloud.

The script itself is intended to run locally.

More details once we find out what we implemented XD

11.3. Involved Cloud Services

The IaC is not running in the cloud itself, but affects all cloud services used by the Test Lab components (see the component sections).

11.4. Interfaces

11.4.1. Used APIs

The IaC script might access the Scene REST API, Test Lab REST API and other cloud resources in a minimally-invasive way as a smoke test to verify successful deployment.

11.4.2. Inputs & Outputs

Input artefacts:

- Web GUI: docker container
- Scene REST API: docker container & database schema definition
- Test Lab REST API: docker container & database schema definition
- Analyzer batch node: Analyzer binaries & batch node script files
- Mobile App APK and Upload Client installer

Output artefacts:

- Config files
- Database backups
- Log / report file

11.4.3. Command-line interface

The IaC is a command-line script.

The usage documentation can be printed by running `iac --help` or also `iac` without parameters.