Horizon 2020
European Union Funding
for Research & Innovation

In2Rail

In2Rail

Project Title:	**INNOVATIVE INTELLIGENT RAIL**
Starting date:	01/05/2015
Duration in months:	36
Call (part) identifier:	H2020-MG-2014
Grant agreement no:	635900

Deliverable D8.8

Integration Test Plan for Application Framework and Constituents

Due date of deliverable	Month 36
Actual submission date	13-04-2018
Organization name of lead contractor for this deliverable	SIE
Dissemination level	PU
Revision	Final

Authors

		Details of contribution
Author(s)	**Siemens AG (SIE)** Stefan Wegele	Chapters 1-9
Contributor(s)	**Ansaldo STS (ASTS)** Gian Luigi Zanella Matteo Pinasco	Review and discussions
	AZD Praha s.r.o (AZD) Martin Bojda Michal Zemlicka	Review and discussions
	Bombardier Transportation (BT) Martin Karlsson Roland Kuhn	Review and discussions
	CAF Signalling (CAF) Carlos Sicre Vara de Rey	Review and discussions
	HaCon (HC) Sandra Kempf Rolf Gooßmann Mahnam Saeednia	Review and discussions
	Thales (THA) Christoph Bücker	Review and discussions

1. Executive Summary

In In2Rail WP8 a standardized ICT structure for Rail Services is specified. This new standardised infrastructure for the future Traffic management systems will not only reduce costs for software development due to standardised interfaces. It also opens a new market for small innovative services building together a traffic management system (TMS).

This will result in the future that instead of one vendor for the entire TMS, who often owns the source code and is able to maintain and evolve the system functionality for 20-25 years, now modules (services) from several vendors are involved.

Therefore a new role is getting more importance: the system integrator, who would take the responsibility to select appropriate solutions on the market, install and manage the communication and execution platform, and maintain the solution for the mentioned period of time. As part of this process the testing process plays a crucial role for establishing a stable system setup.

This document represents a possible testing approach consisting of two major steps:

- the module vendor implements unit, function and integration tests and delivers them together with the module. A successful test run would prove that the system integrator was able to integrate the module into TMS according to specification of the module vendor;
- the system integrator develops system tests, evaluating the entire system functionality independently and at a high level.

For both tasks the appropriate approaches are shown, which automate testing on the one hand and reduce the development costs through usage of COTS testing frameworks on the other hand.

This document is not standardising the testing approach of the TMS as the number of possible implementations is huge, but gives a direction, how the standardised TMS infrastructure can be efficiently exploited for test automation. The test examples in the document are dedicated to the software developers and architects, who would easily identify the basic concepts and solutions from the source code snippets in Appendixes A-C.

TABLE OF CONTENTS

2. Abbreviations and acronyms

Abbreviation / Acronyms	Description
AF	Application Framework, either a standard for plug-and-play service management developed in In2Rail (WP8) or a specific implementation of this standard.
API	Application Programming Interface.
COTS	Commercial off the shelf: available software as a general module.
DLL	Dynamic-link library, executable code, which can be linked to the application at run time.
HTTP	Hypertext Transfer Protocol is the main protocol for data communication for the World Wide Web.
ICT	Information and Communication Technology
IL	Integration Layer, either standard for communication platform for future TMS developed in In2Rail (WP8) or a specific implementation of this standard.
JavaScript	It is a high-level interpreted programming language.
IMDG	In Memory Data Grid, a data management technology for replicated object states.
JAR	Java Archive is a package file format aggregating Java classes with configuration and meta information.
JSON	JavaScript Object Notation, a human readable format for specification of objects.
Protobuf	Protocol Buffers is a method for serializing of structured data.
REST	Representational State Transfer, architectural style for development of distributed applications.
RTC	IBM Rational Team Concert, a versioning system.
RTTP	Real time traffic plan
SO	Shared Object – the extension of dynamic linked libraries in Unix world.
TMS	Traffic Management System
XML	Extensible Markup Language

3. Background

This document represents the next step in system design after detailed specification of the TMS platform in [In2Rail D8.3], [In2Rail D8.4], [In2Rail D8.6], [In2Rail D8.7] and after developing the application code in the proof of concept prototype in [In2Rail D7.5] in the framework of the project entitled “Innovative Intelligent Rail” (Project Acronym: In2Rail; Grant Agreement No 635900).

The document provides concepts for the right part of the system development process: Unit, integration and partly operational testing (s. Figure 3.1).

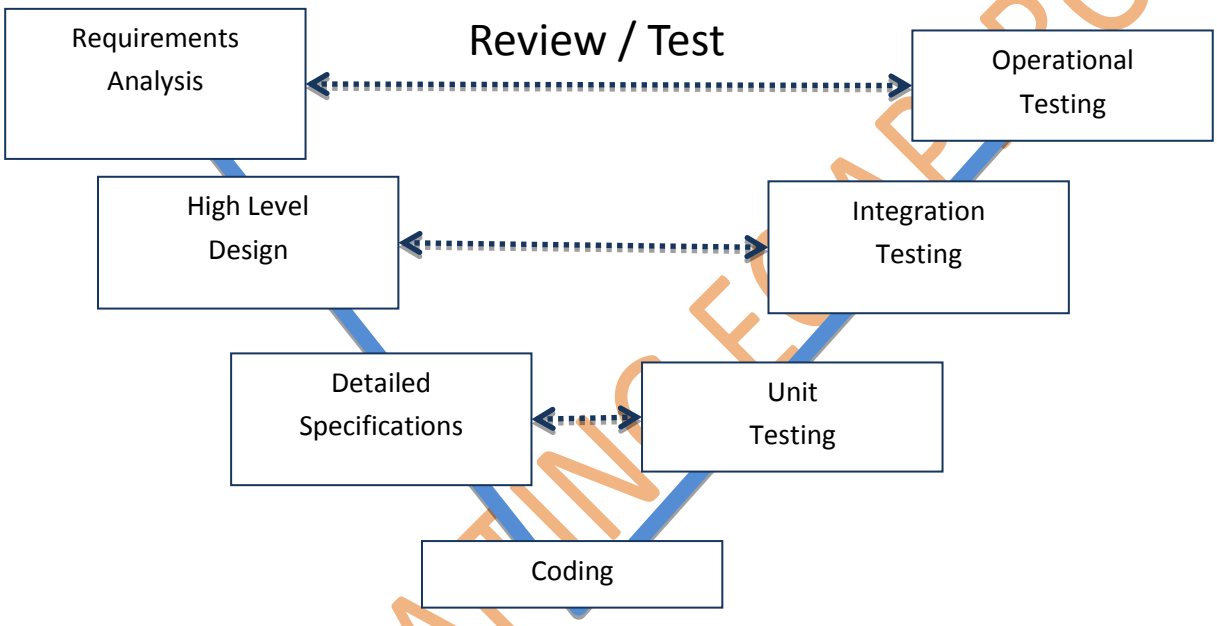

Figure 3.1: V-Model of the software development process

The content of this document was developed in parallel with the software development of the proof-of-concept-prototype described in [In2Rail D7.5], so the approaches presented here were developed and applied for the testing of the proof-of-concept prototype. This close cooperation allowed high maturity level of the testing approach for the following projects in Shift2Rail especially during development of technical demonstrators.

4. Objective / Aim

The overall objective of Work Package 8 – (WP8) – is to provide the specification of the architecture, protocols, and functional description of the required services. They should allow a seamless integration within TMS of:

- external systems like Crew Management, Fleet Management, and Maintenance Management etc.;
- TMS-specific applications provided by different suppliers, e.g. Timetable management, Automatic Route setting (ARS), Forecast, Decision support system, Task management, Route cause analysis etc.

Objective of this deliverable is to provide:

- a description of the testing approach for services managed by Application Framework;
- a description of common integration tests to be evaluated during test phase.

In opposite to the deliverables [In2Rail D8.4] and [In2Rail D8.7], where the specifications for Integration Layer and Application Framework are provided, this document represents one possible approach to implementing testing in the context of IL and AF. Although only a description of integration testing was planned for this document the nature of Integration Layer and Application Framework allows the automation of several kinds of tests: module, integration, function and system tests. In the following sections different aspects of testing will be considered.

5. Introduction

This document covers a very specific area in software engineering. To be useful in future EU-projects it goes deep into the specifics of the software testing in connection with architectural patterns published in other deliverables. Therefore it is dedicated to software architects, software developers and software testing engineers. To understand this document it is advisable to read [In2Rail D8.3] first.

*Testing is the **process** of executing a program with the intent of finding errors. [Myers et al 2011].*

The test process is strongly integrated into the software development process (see Figure 5.1).

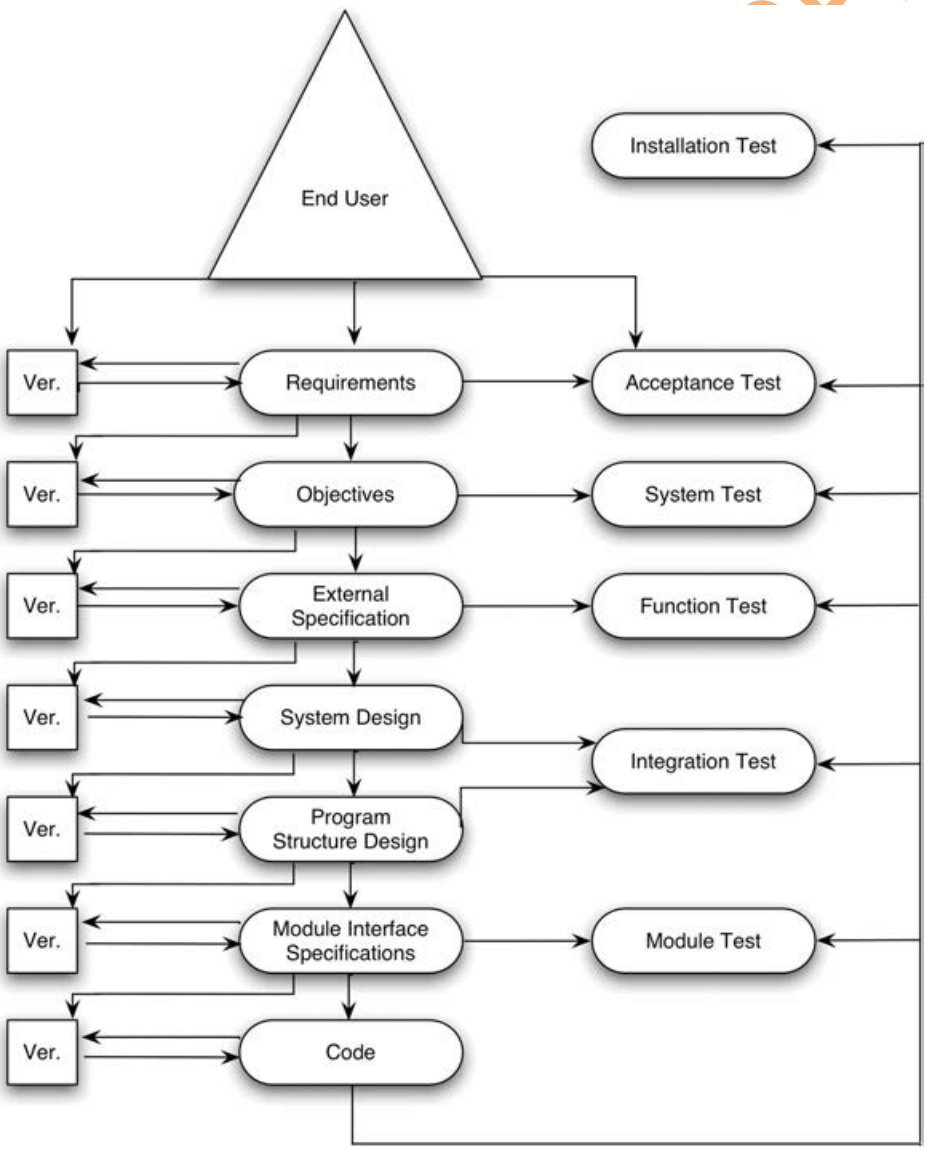

Figure 5.1: Correspondence between development and testing process [Myers et al 2011]

The single test steps are distinguished not only on the input/output relations and their position in the development process. The single test steps are dedicated to finding different kinds of errors. They have the following responsibilities:

- Module Test is a process of testing the individual sub-programs, subroutines, classes, or procedures in a program. The purpose of a module test is to find discrepancies between the program's modules and their interface specification [Myers et al 2011];
- Integration Test is the test of correct interaction among all components or services within a system;
- Function Testing is a process of attempting to find discrepancies between the program and the external specification;
- System Test is the process of finding discrepancies between the program and its original objectives. To formulate the test cases [Myers et al 2011] proposes to use the user documentation;
- Acceptance Test is a process of comparing the program to its original requirements and the current needs of its end users. In case of contracted program, the contracting organisation performs the acceptance test by comparing the program's operation to the original contract;
- Installation Test has the purpose of finding errors that occur during installation process.

A high degree of test automation is desired on all levels of testing since the effort of testing in terms of time and resources can be minimized by this measure. Integration Layer and Application Framework allow extensive automation in different steps of the test process. To show these possibilities the TMS prototype developed in the context of the "proof of concept" [In2Rail D7.5] will be used as a "system under test".

The TMS prototype follows the micro service architectural pattern [Cambell 2015] [AmundsenMclarty2016], therefore a typical service (program) contains often only one module. The structure of the TMS prototype is shown in Figure 5.2.

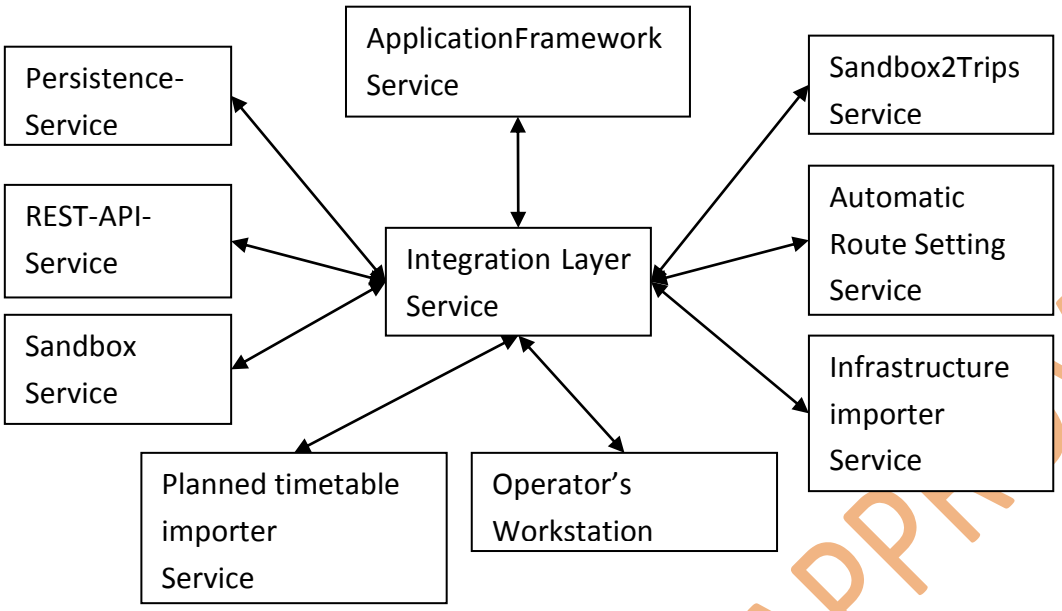

Figure 5.2: System structure of TMS prototype

As follows from the system architecture all services (programs) communicate with each other only by means of the Integration Layer service. They have no knowledge about the rest of the system: which services there are, if they are running or not, and if they work continuously or event/time based. The Integration Layer itself provides only data management and communication infrastructure. The data content is specified by Canonical Data Model described in CDM-Appendix of [In2Rail D9.1]. Specific data structures for Timetable, Infrastructure, Version Management are specified in [In2Rail D8.4] and [In2Rail D8.7].

The services in the prototype have the following responsibilities:

- Integration Layer provides the In Memory Data Grid (IMDG) functionality;
- the data is represented as key-value-pairs allocated into "containers" named topics (or maps);
- the clients (other services) connect to the Integration Layer by a dynamic library;
- the clients can create, read, update, and delete key-value-pairs,
- the clients can observe topics and will receive notifications on any change;
- IMDG builds a cluster resolving single point of failure for data management – as long as at least one node in the cluster is working, no data is lost;
- persistence service is responsible for storage of key-values located in "important" topics on persistent disk. In case of disaster (all nodes building the IMDG cluster go down) the persistence service shall recover the last known state;

- REST-API service provides a REST-interface [Masse2011] to the IMDG. It allows accessing the key-value-pairs in human readable JSON-Format [Bassett2015] using REST-API;
- Sandbox service manages concurrent change requests coming from different clients integrating them in to one data set modelled by a sequence of snapshots and delta-change-sets;
- Sandbox2Trips service extracts single trips building the operational timetable from a sandbox and publishes them on one of the topics. The automatic route setting service uses them to set the routes;
- Planned timetable importer is part of the existing TMS which converts proprietary timetable data into canonical data model-format;
- Infrastructure importer service converts Railml-2.3 topology data into canonical data model format and appends it to the production sandbox;
- Operator's Workstation contains several dialogs:
  - Timetable editor,
  - Track view,
  - Sandboxes view,

  and allows for several operators' concurrent modification of the Real Time Traffic Plan (RTTP);
- Application Framework service reads the desired state of all services (which services are running and with which topics they are communicating) from a special Topic and ensures that services managed by the Application Framework apply the desire state. To do so it starts, stops and monitors the managed services on a node-cluster (set of computers) managed by the Application Framework.

In the following sections the test steps from Figure 5.1 will be analysed in conjunction with the TMS prototype.

In short the testing approach used in the prototype development and proposed in this deliverable can be summarized as following:

- REST-API-Service allows usage of many REST-API-Testing frameworks from the market [Chakram2018][Django2018][Frisby.js2018];
- a test script inside such REST-API-Testing framework writes key-values over REST-API influencing services and reads key-values over REST-API analysing the service results;
- Application Framework starts/stops services under test;
- the Test framework runs the tests, collects the test results and represents them in an appropriate manner (e.g. a web-page-report).

From the variety of REST-API-Testing frameworks the open source framework Chakram was selected for the prototyping activities. In the following different test scripts will be analysed covering several steps of the testing process taking services from the prototype as examples.

6. Test environment

In most of test steps up to operational tests some components do not exist and must be emulated. Even during the acceptance test the connection to external systems could be missing. If the TMS is based on Integration Layer the standardised communication takes place there. From the testing point of view the Integration Layer provides crucial features described in the following.

6.1. Observability of communication

The Integration Layer provides access to the data by publish-subscribe principle. The "data universe" is separated in Topics. Each Topic represents a map of key-value-pairs. Any client with sufficient access rights is able to subscribe to any Topic and write to any Topic (see Figure 6.1).

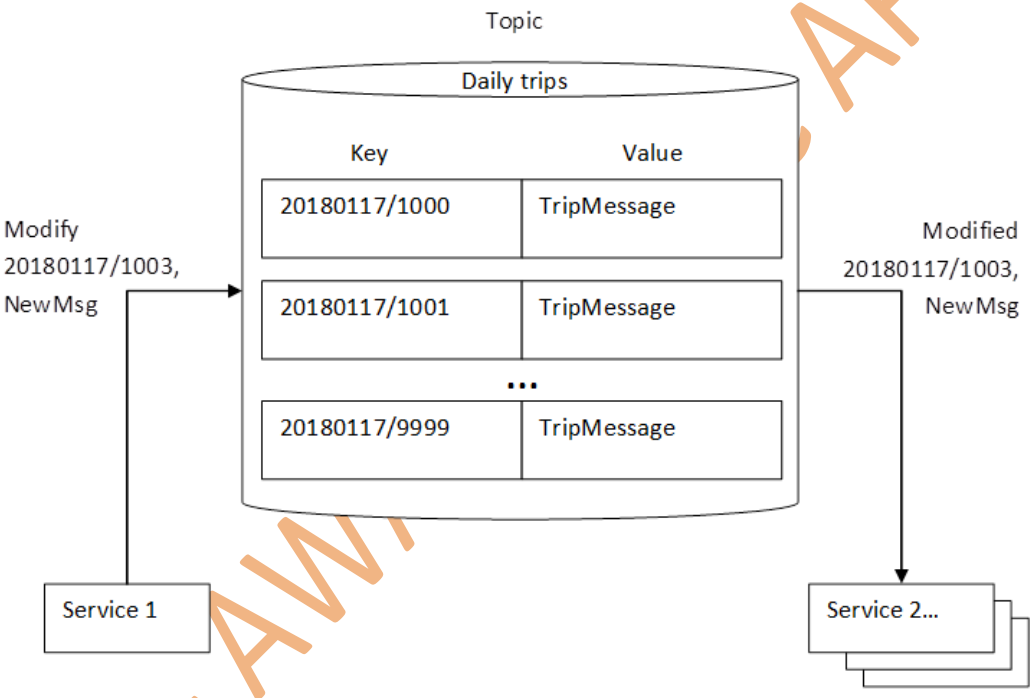

Figure 6.1: Topic example in Integration Layer

Any service influences the remaining part of the system only by publishing new states of key-value objects on Integration Layer. As a consequence to emulate the missing parts of the system it is sufficient to publish "appropriate" key-value-states on Integration Layer to allow running services to react. This reduces the test efforts quite strongly: instead of developing emulated subsystems in software, it is enough to formulate the messages in a test script and let the existing test framework emulate missing subsystems.

6.2. REST-based API

The main method to connect to Integration Layer represents a dynamically linked library (dll, so- or jar-library). The values in key-value-pairs are represented by a binary protocol e. g.

[Protobuf2017]. Both specialities prevent usage of general testing frameworks from the market:

- the API-calls to DLL required to access the data are not supported;
- a special treatment (encoding and decoding) of the messages is typically not supported.

To overcome these issues the Integration Layer includes a special service providing a REST-based API [Masse2011]. That means the test framework is able to create, read, update and delete any key-value-pair on Integration Layer using standardized HTTP-protocol solving the first issue with the DLL – it is not required for the testing (see Figure 6.2 below).

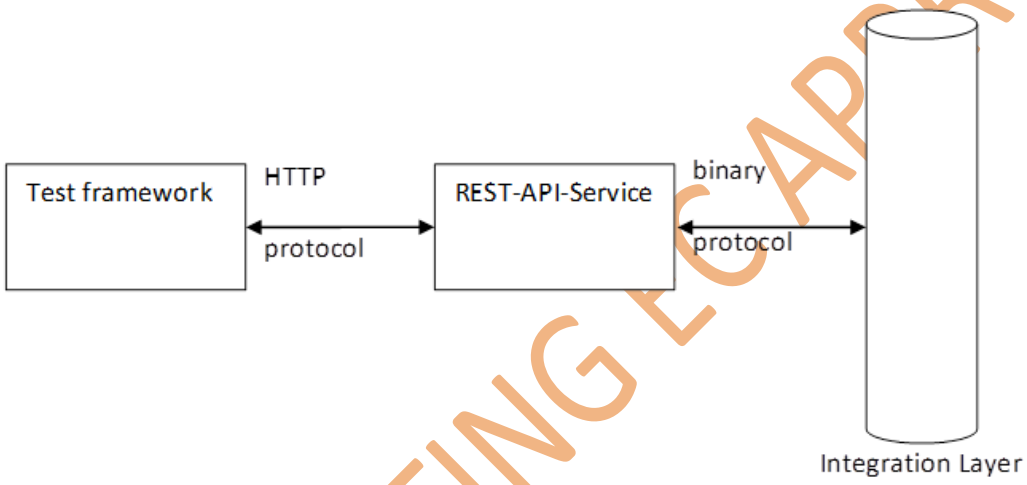

Figure 6.2: Access of Integration Layer through REST-API service

The REST-standard assumes the object representation either in XML or in JSON format. The REST-service converts binary representation of the value into JSON and back for the input. A test framework usable in this case is associated with the term “REST API Testing”. A web-search with this string returns over 6 Mio hits containing many tools and theoretical discussions on the subject.

The major drawback of this approach is the additional delay introduced by the REST-API-Service:

- the HTTP protocol with JSON data representation consumes about 10 times higher bandwidth in comparison to binary protocol with enabled compression;
- conversion JSON<->Protobuf could consume considerable computational resources on the node running the REST-Service. In bigger test cases several instances of the REST-Service might be required for load balancing.

As a consequence the REST-API cannot be used for performance evaluation of the Integration Layer. But it is still sufficient for testing the service performance as the communication part of the total service response time is often negligible.

The REST-API is planned to be used not only for testing purposes but for connection of "low-performance" services, where the throughput and round-trip times are not critical, e.g. a timetable import every four hours is often allowed to take several minutes, so the communication time of about one second is acceptable.

6.3. Test environment setup and organisation

For the proof-of concept in WP7 the open source REST-API testing framework Chakram [Chakram2018] has been selected. It is based on node.js [Cantelon et al 2017] which is a JavaScript framework. As JSON represents native JavaScript-Code the possibility to write test scripts in JavaScript simplifies handling of the IL-Messages: the logic can be applied on the input message directly and to send a new message a normal JavaScript-object can be used.

The plug-and-play infrastructure provided by the Application Framework opens new possibilities and challenges in system deployment: instead of integrated development of the entire big system by one vendor, services coming from several vendors can be integrated together. To ensure that the delivered service is installed properly, the service vendor should provide an installation test-service as a part of its delivery (see Figure 6.3).

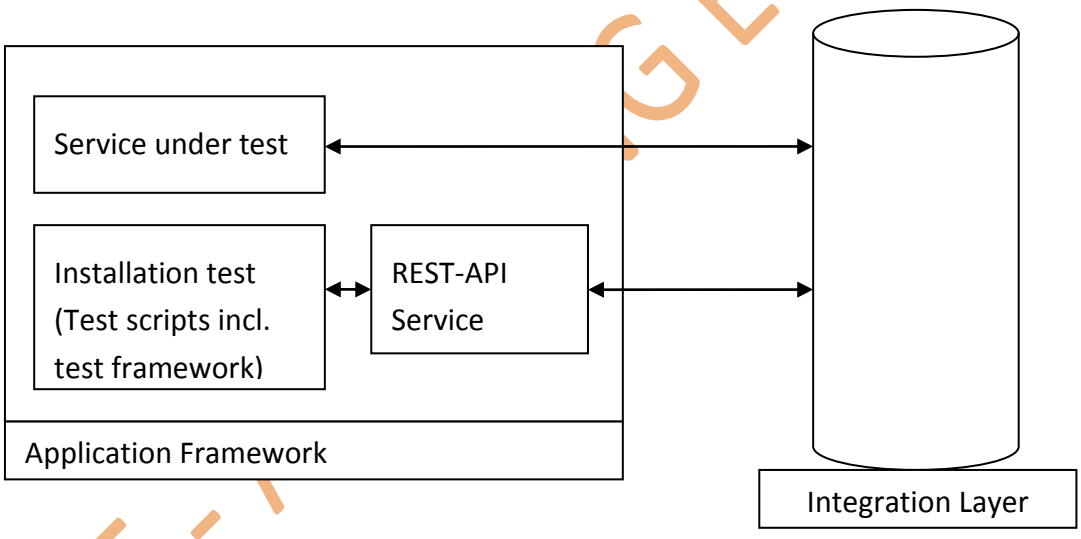

Figure 6.3: Test setup for productive service to be provided by service vendor

With increasing number of such "independent" services the availability of the automated (installation) tests appears to be crucial to keep the system stable and manageable.

6.4. Test implementation approaches

As shown in the previous chapters tests can be implemented using different approaches. An obvious one is to use an external testing framework to produce inputs, observe and analyse outputs of the service (Figure 6.4).

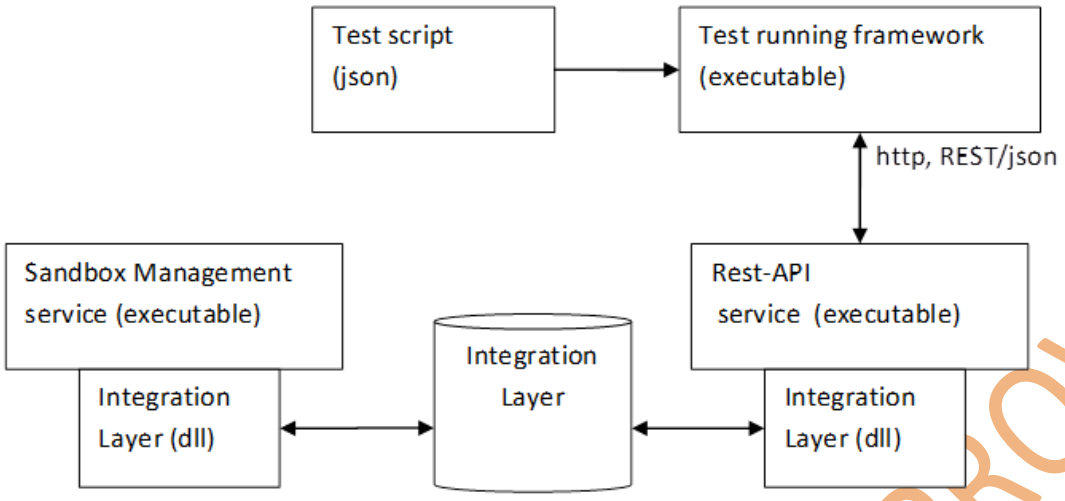

Figure 6.4: Test setup with an external testing framework

The main advantage of this approach is: it is independent from the programming language, operating system, CPU architecture etc. of the service under test. Therefore this kind of tests can be implemented and managed by the system integrator. As it is typically not the responsibility of the integrator to test the entire functional behaviour, most of the tests shall be implemented by the service vendor.

As the owner of the source code and the development environment, the service vendor has additional degrees of freedom for selecting an appropriate test setup. The service vendor is facing to competing test objectives:

- test-runs should be implemented efficiently (fast);
- even for small (micro) services the number of tests can reach hundreds;
- each build typically requires running all (or at least as many as possible) automated test. Unnecessarily long running (slow) tests would significantly reduce the effectivity of the developers, while they are waiting for the test results;
- test should be as similar as possible to the production use. In case of IL-based services, they should be connected to the IL later used in the production, which requires setup and management of an integration Layer for each testing environment.

A possible solution for this situation is shown in Figure 6.5. As the IL-based services expose its functionality over IL the tests scripts can be implemented as API-testing over IL.

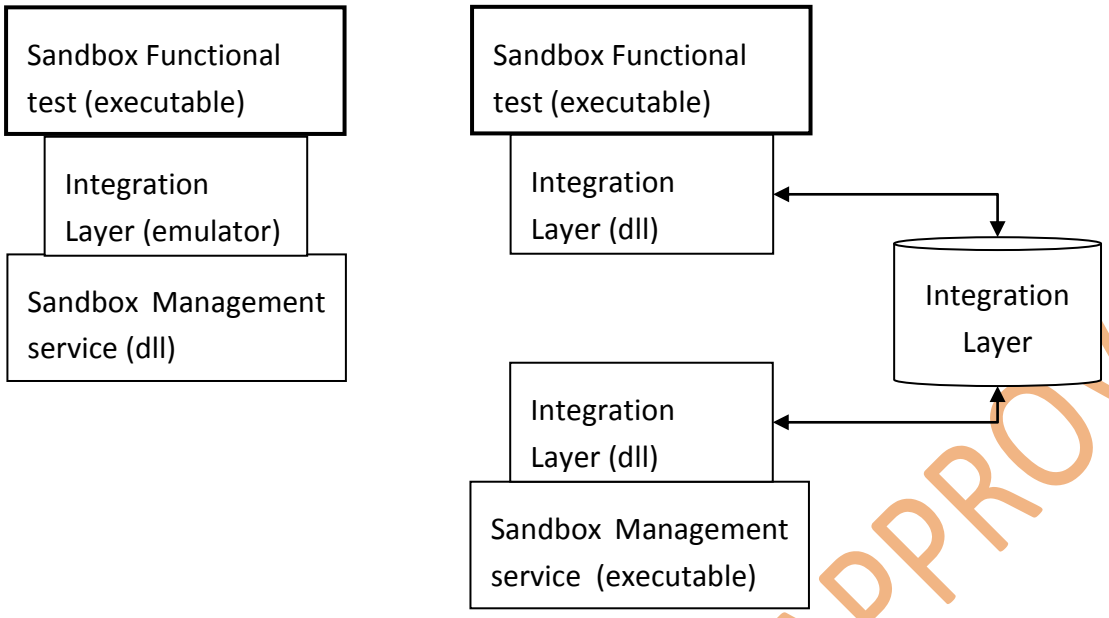

Figure 6.5: Reuse of function test for "fast" and for "productive" test runs

If the testing code is not aware of the implementation of the IL, two test setups can be constructed by the build environment:

- connecting test code through the emulated Integration Layer with a library of the service under test allows a very fast test execution: no overhead for communication with an external IL and time required for starting/stopping/managing the external IL;
- connecting the same test code to the productive IL allows reusing hundreds of test cases on productive environment and can be delivered together with the service implementation. This test setup is able to implement unit, integration and functional tests in micro service architecture.

In the following examples for test implementations for different services in the In2Rail-Proof-of-concept-prototype will be discussed.

7. Module tests

Module testing (also called unit testing) is a process of testing the individual components building a program: class, method, library (a set of classes). The purpose of the module testing is to compare the function of the module with an existing specification either as a functional or interface specification.

Module testing covers “private” modules known only to the service vendor. The functional or interface specifications are results of internal development steps during the service development. As the module test strategies are very vendor specific this chapter gives only a short overview on the module testing and provides some hints, how to increase test execution performance in conjunction with Integration Layer.Consider a program as a combination of modules with some call hierarchy a possible program structure can look like Figure 7.1.

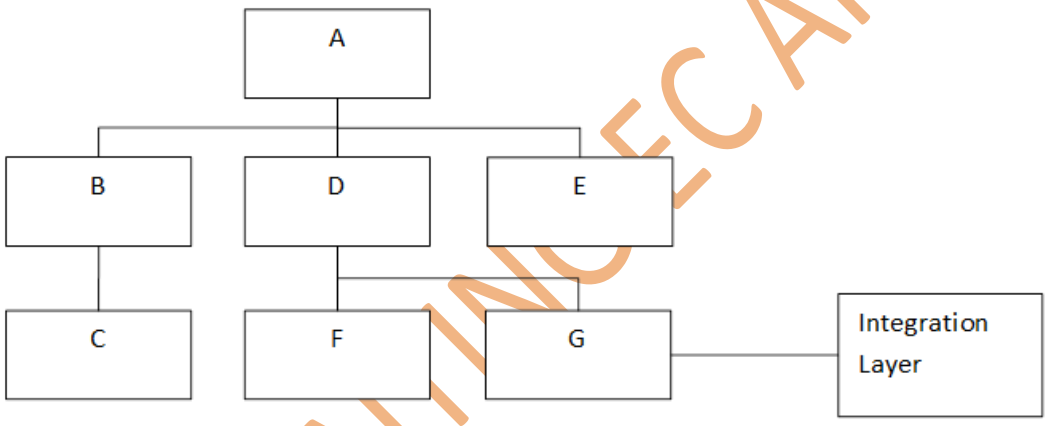

Figure 7.1: Sample program with connection to Integration Layer

Module A uses (calls) modules B, D, E. Module G interacts with the Integration Layer.

Module test scripts are typically combined to test suites, which are executed by some test framework like Boost Test Library, Google Test, and JUnit. In the context of larger software projects a big amount of such test scripts (several hundreds) are coming together.

The Module tests are typically repeated as “regression tests” – a build of a new version is considered as successful only if the associated unit tests don’t find errors. As a consequence of the large number of tests and frequent test executions the test execution time plays significant role for software development especially at the end of the project. To reduce execution time and simplify test setup a local “Test-version” of the Integration Layer could be created. As the object oriented interface of IL is specified in [In2Rail D8.4], the local IL represents a managed set of maps (which are typically part of standard libraries in C++, Java…). Compiled as a library it can be used in test running suites reducing execution time by at least one order of magnitude.

Another advantage of a generic implementation for the Testing-IL is the possibility to reuse it in many module-tests for programs using IL. Instead of a local Test-version of the IL a server-

less implementation can be used, e.g. DDS based wrapper [In2Rail D8.3]. This approach extends the test execution time as it tries to find connected systems during login procedure. Afterwards it works the same way as the local IL.

8. Integration Tests

Integration Testing represents a step in the test process where single modules/programs are combined and tested as a group. The purpose of this step is to find errors in the interaction between modules/programs. In [Myers et al 2011] the integration testing is not considered as a separate testing step, as it is an implicit part of the incremental module testing.

In this document a combination of services/programs is considered under integration tests, while modules of a single program are not taken into account.

Two kinds of integration testing can be identified:

- component integration testing, with the purpose to expose faults in the interfaces and interaction between integrated components;
- system integration testing, testing the integration of systems, including interfaces to external systems (organisations).

In the following only the component integration testing is considered. As the Integration Layer is used only for components integration no generic approach can be provided for system integration testing.

The main advantage of the Integration Layer represents the fact that no components interact with each other directly, but only by means of IL. This reduces the component integration tests to the test of a single service integrated into the Integration Layer (Figure 8.1).

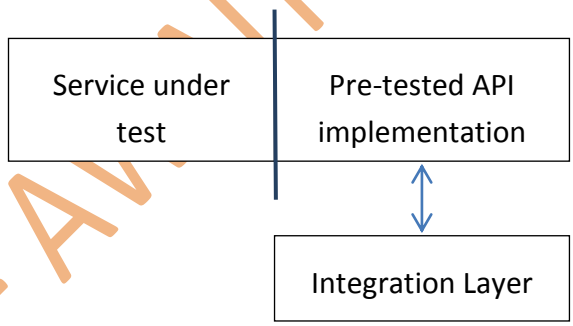

Figure 8.1: Integration test setup of a service within Integration Layer

As the integration of API with the Integration Layer is already tested by the IL-vendor, the integration test is responsible for correct usage of the API by the service under test. This is typically part of the API implementation, which always validates the service requests for correct message format.

In a future TMS it is assumed that the life cycle of some services will be managed by Application Framework. To allow that functionality the Application Framework should interact with these services using a mean other than IL. Therefore the main use case for the integration testing would be the integration of a managed service into the Application Framework (see Figure 8.2).

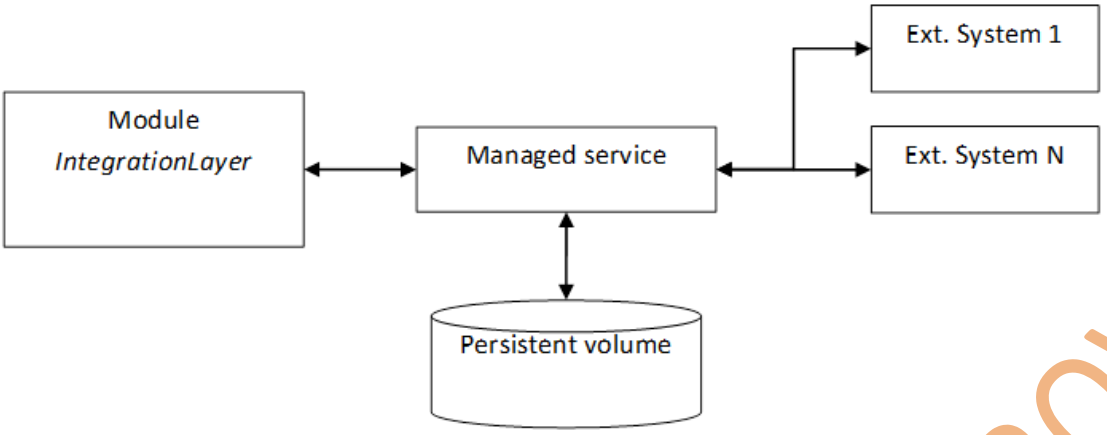

Figure 8.2: Integration of a service into Application Framework

In the proof-of-concept prototype the Application Framework was implemented based on Docker Swarm technology [Farcic2017].

The Application Framework has the following responsibilities:

- deploy a service on the cluster (Docker Swarm uses an extra registry service as an image source);
- provide a set of values to the service through environmental variables to allow the service to start interaction with Integration Layer;
- if needed provide persistent volume(s) and notify the service about its "position" through environmental variable;
- provide a port mapping between the service ports and the host ports. The main use case for the port mapping is connection of external systems to the managed service;
- ensure that required number of service instances is running;
- monitor the service instances – activity, centralised logging, fail/restart history etc.

The Application Framework is controlled using Integration Layer. On a specific topic "AFDesiredStates" on the Integration Layer key-value-pairs represent required states of the services. The Application Framework compares the desired state with its observations and initiates actions if it detects differences. The Application Framework publishes its observations on "AFCurrentStates"-topic. Access of the AF-functionality through the "AFxxxStates"-Topics allows using the same testing framework as for function tests of the single services (s. Figure 8.3).

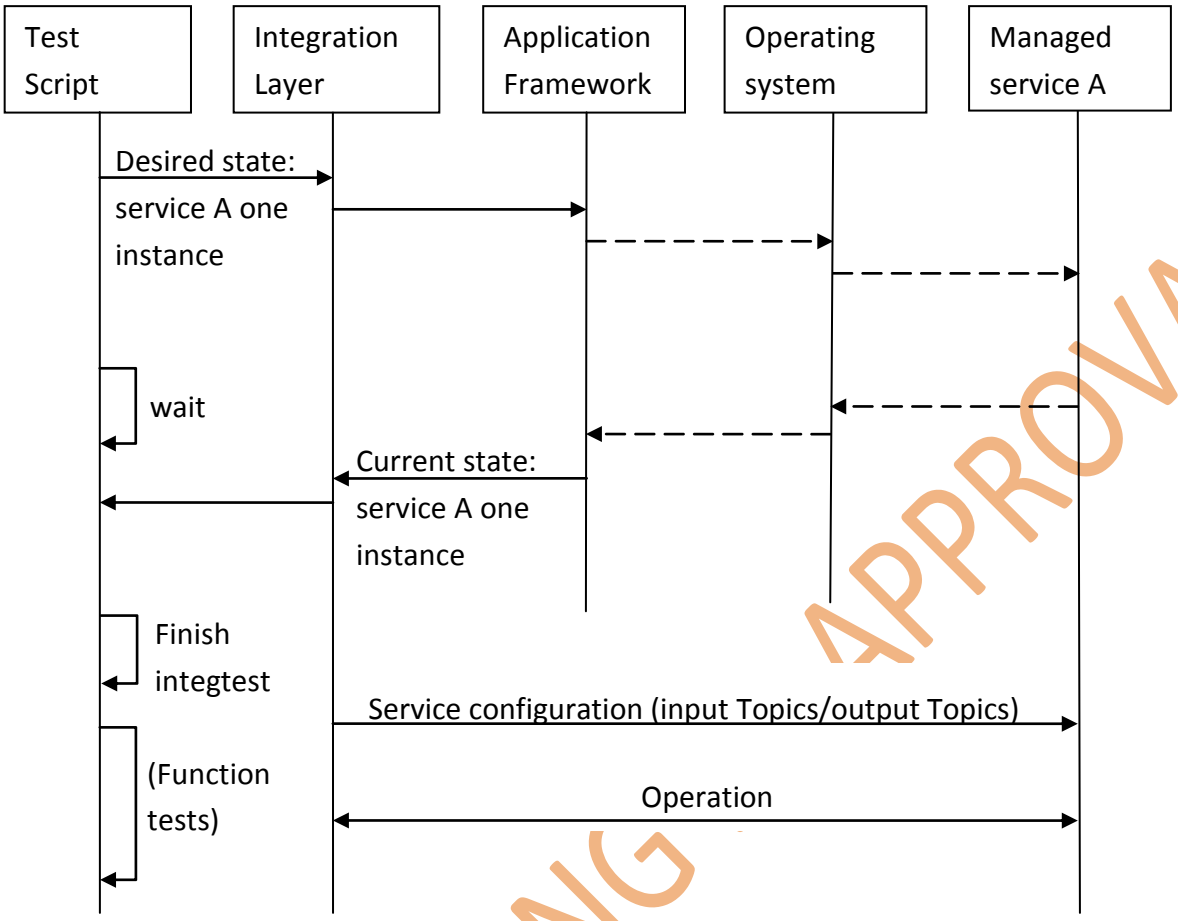

Figure 8.3: Example of a sequence diagram for an integration test

8.1. Use case “Persistence service”

In the following a test script for integration test of the persistence service is shown. The “desired state” for the service is in the file “*persistence_service_spec.json*”

```
{
	"Name" : "persistence_service",
	"ContainerSpec" : {
		"Image" : "registryhost/persistence_service:2.1",
		"Mounts" : [{
			"Source" : "data",
			"Target" : "/var/data",
			"Type" : "volume",
			"Consistency" : "consistent"
		}],
		"Env" : [
			"topicsListTopic=internalTopics"
		]
	}
	"Placement" : {
		"Constraints" : ["node.persistence_role==true"],
		"Platforms" : [{
			"Architecture" : "x68_64",
			"OS" : "linux"
		}]
	},
	 "Mode" : {
		"Replicated" : {
```

```
                "Replicas" : 1
            }
        }
}
```

The service specification follows Docker-Swarm API, assuming that the mapping to other similar products would be easily possible https://docs.docker.com/engine/api/v1.35/#operation/ServiceCreate. The content of the request is defined in Table 8.1.

Attribute	Description
ContainerSpec.Name	Name, version and location of the image.
ContainerSpec.Mounts	List of "disks" provided to container from the host
ContainerSpec.Mounts.Source	Name of the volume on the host. Docker and Kubernetes provide an abstraction for Volume – named storage, which can be a special service on the cloud or a normal directory on the host.
ContainerSpec.Mounts.Target	Directory at which the volume is mounted inside of container.
ContainerSpec.Mounts.Type	Type can be either `volume` (abstraction) or `bind` which represents a normal directory on host .
ContainerSpec.Mounts.Consistency	`Consistent` means that data written by the service is immediately provided to the host to be written in the volume. If the loss of couple of seconds of data is acceptable the load introduced by the persistence_service can be reduced by allowing the system to provide written data in batches with a value `delegated`.
ContainerSpec.Env	This attribute allows providing environmental variables to the container. In this case the topic for the list of topics is specified, used by the service to identify topics to be persisted.
Placement.Constraints	Allows specifying on which types of hosts the service shall be executed. In this case only hosts with the label `node.persistence_role==true` are allowed .
Placement.Platforms.Architecture	The service inside the container represents a binary executable, which is specific to the processor architecture. If the Docker Swarm Cluster contains heterogeneous nodes the architecture requirements must be specified.
Placement.Platforms.OS	Specifies required Operating System on the host.
Mode.Replicated.Replicas	Specifies the number of instances of the service, which shall be executed concurrently. In case of persistence_service one instance is typically sufficient.

Table 8.1: Attributes description for persistence_service

A test script using Chakram-API-Test-Framework [Chakram2018] covering integration of the persistence service into Application Framework can be as follows.

Integtest_spec.js

```
var call = require('chakram');
expect = call.expect;
var delay = require('timeout-as-promise');

//reading persistence service specification (s. above)
serviceSpec = require('persistence_service_spec.json');
//configuration for hosts/urls/etc required for the test
testConfig = require('testConfig.json');
appUrl = testConfig.APP_URL;

//start test suite
describe("Integration test for persistence_service", function () {
   //max waiting time for http response
   this.timeout(1000);

   it('start and stop persistence service', function () {
      return call.post(appUrl + "/afdesiredservices/values/", serviceSpec)
      .then(function(r1) {
         expect(r1).to.have.status(200);
         //waiting testConfig.delay (5 seconds) until the service
         //is asynchronously installed and started
         return delay(testConfig.delay);
      })
      //after waiting time read current state of persistence_service
      .then(function() {
         return call.get(appUrl +
"/afactualservices/values/persistence_service");
      })
      //if ok, delete the persistence service
      .then(function(r2) {
         expect(r2).to.have.status(200);
         //stop the service by deleting its key from desired services
         return call.delete(appUrl +
"/afdesiredservices/keys/persistence_service");
      })
      //wait 5 seconds
      .then(function(r3) {
         return delay(testConfig.delay);
      })
      //check that persistence_service is not there any more
      .then(function() {
         return call.get(appUrl +
"/afactualservices/values/persistence_service").then(function(r4) {
            expect(r4).to.have.status(404);
         });
      });
   });
});
```

9. Function tests

According to [Myers et al 2011]:

The purpose of a function test is to show that a program does not match its external specifications.

In the following two services will be analysed as examples for the function tests:

- persistence service representing a constituent of the Application Framework;
- sandbox management service representing a constituent of the Integration Layer.

First the requirements of the services will be summarized representing *external specification* of the program. Then the possibilities for implementing function tests for each service will be shown.

9.1. Persistence service

As a first example, the persistence service from the proof-of-concept prototype is considered (see Figure 5.2). The persistence service has only one responsibility: Restoring the content of the Integration Layer upon request (requirement 5.2.2.5 in [In2Rail D8.1]).

The Integration Layer manages its data in an In Memory Data Grid (IMDG). As long as sufficient number of nodes running the IMDG are online all the data is safe even in case of failure of some nodes executing IMDG. The persistence service has a role of a backup system for the case of major disaster (failure of all nodes of the Integration Layer). During the service development it can be used to setup initial system configuration.

To fulfil the assigned functionality the persistence service:

- observes the list of topics available in IMDG;
- subscribes to topics annotated as “PERSISTENT” and stores their current state on disk;
- if started with the request “Restore”, reads stored key-value-pairs from the disk and publishes them on Integration Layer;
- if started without the request “Restore” adjusts outdated key-values on disk to the current state of IL.

The test script for the function test is similar to the one for integration test.

9.1.1. Test case 1

A high level test description is provided in Table 9.1 and the single test steps are listed in Table 9.2.

Test case 1 – Functional test for persistence service	
Precondition	▪ Redundant servers running IL are started ▪ Application Framework is started ▪ REST-API service is started ▪ Image for the Persistence Service is pushed into service registry of the Application Framework
Test Description	Test the main function of persistence service to restore last state of the Integration Layer
Expected Test Case Result	All topics annotated with PERSISTENT durability are restored after simulated crash of Integration Layer.

Table 9.1: Test description for backup-test of persistence service

Test Step	Action	Expected Result
1	Start Persistence Service	Persistence service is running.
2	Create TestTopic1 with persistency and data type 1	TestTopic1 is available.
3	Create TestTopic2 with persistency and data type 2	TestTopic2 is available
4	Fill TestTopic1 and TestTopic2 with pseudo-random data for defined time interval including all CRUD operations.	TestTopics1 and TestTopic2 have key-values.
5	Stop Persistence Service	Persistence Service is not running
6	Remove TestTopic1 and TestTopic2	TestTopic1 and TestTopic2 are not available.
7	Start Persistence Service with “restore”-request	Persistence Service is running, TestTopic1 and TestTopic2 are available and contain the same key-values as after step 4.

Table 9.2: Test steps for backup-test of persistence service

9.1.2. Test case 2

In this test the failover functionality of the Persistence Service shall be tested. In case of a failure the service shall not modify any value on IL after restart, but it shall be able to recover the Integration Layer if restarted with “restore request”.

A high level test description is provided in Table 9.3 and the single test steps are listed in Table 9.4.

Test case 2 – Functional test for persistence service

Precondition	▪ Redundant servers running IL are started ▪ Application Framework is started on one node ▪ REST-API service is started ▪ Image for the Persistence Service is pushed into service registry of the Application Framework
Test Description	Test for failover of the persistence service itself.
Expected Test Case Result	After failover the persistence service does not modified anything on Integration Layer is able to restore IL after one minute latest.

Table 9.3: Test description for failover

Test Step	Action	Expected Result
1	Start Persistence Service	Persistence service is running
2	Create TestTopic1 with persistency and data type 1	TestTopic1 is available
3	Create TestTopic2 with persistency and data type 2	TestTopic2 is available
4	Fill TestTopic1 and TestTopic2 with pseudo-random data for defined time interval including all CRUD operations.	TestTopics1 and TestTopic2 have key-values
5	Kill the processes assigned to Persistence Service	Persistence Service is not yet running
6	Repeat step 4	TestTopic1 and TestTopic2 have different content as after step4
7	Check the state of Persistence Service	Persistence Service is running as it is restarted by Application Framework
8	Stop Persistence Service	Persistence Service is not running
9	Remove TestTopic1 and TestTopic2	TestTopic1 and TestTopic2 are not available on IL
10	Start Persistence Service with “restore”-request	Persistence Service is running, TestTopic1 and TestTopic2 are available and contain the same key-values as after step 7.

Table 9.4: Test steps for persistence service failover test

9.2. Sandbox management service

The sandbox management service provides a versioning control for data requiring transactional behaviour (requirement 2.3.3 in [In2Rail D8.1]). A good metaphor represents file versioning systems like Git, Mercurial, and IBM Rational Team Concert (RTC). The versioning system RTC even has a term “stream” representing a sequence of modification. It builds the basis for cooperation between developers and teams [RTC2018].

The sandbox management service keeps a list of snap-shots and delta updates consistently synchronising change requests from concurrent client applications (see Figure 9.1).

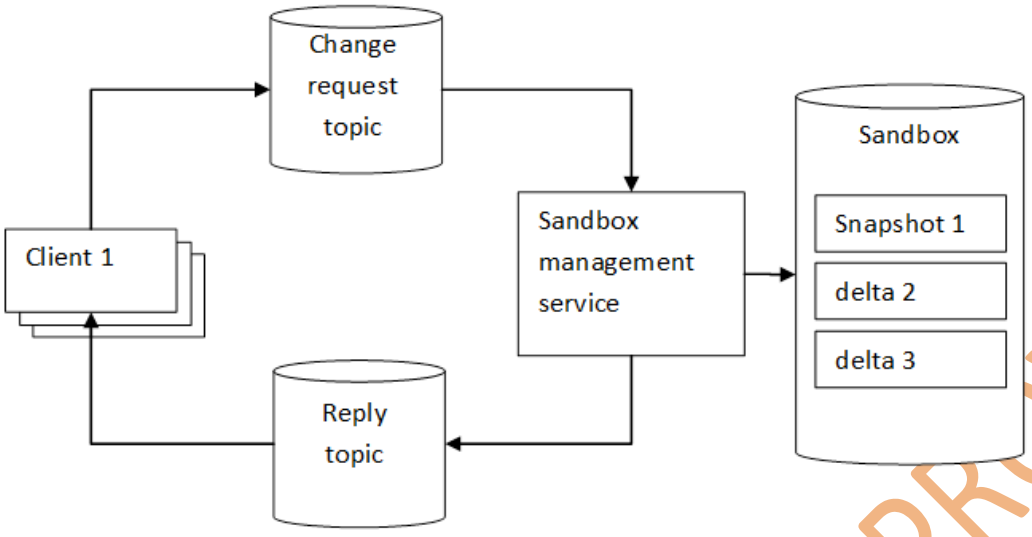

Figure 9.1: Sandbox management service

Depending on the configuration of the Sandbox Management Service can apply additional logic to the change requests:

- validation steps to check access rights of the client to specific part of the model;
- merging of change requests to already applied deltas, if it is acceptable for the managed sandbox.

The validation process can be externalized to an additional service, which observes the "ChangeRequestTopic" and puts results of validation to "ValidatedRequestTopic", observed by Sandbox management service.

In the following two subsections of the function "Creation of a new sandbox" will be shown. In Appendix A a test script evaluates concurrent change requests. In Appendix B a model based functional test for the remaining functions is presented:

- appending new change requests;
- merge change requests if possible;
- undo one or more change requests;
- accept sandbox;
- cancel sandbox.

9.2.1. Major functions testing

The high level of the test description is defined in Table 9.5. The required test steps are shown in Table 9.6.

Test case 1 – Functional test for Sandbox Management Service

Precondition	▪ Redundant servers running IL are started ▪ Application Framework is started ▪ REST-API service is started ▪ Image for the Sandbox Management Service is pushed into service registry of the Application Framework
Test Description	Test creation of a sandbox, handling of concurrent change requests, handling of undo requests, accept sandbox, remove sandbox.
Expected Test Case Result	The changes introduced to the test sandbox appear in the parent sandbox.

Table 9.5: Test description Sandbox management service

Test Step	Action	Expected Result
1	Start Sandbox Management Service	Sandbox Management Service is running.
2	Request to create a new sandbox	Three additional topics are created: ▪ Change Requests ▪ Replies ▪ Sandbox itself
3	Emulate concurrent requests from three clients with pseudo-random requests including Undo-Requests	Consistent state of the sandbox, with reasonable replies.
4	Request to accept the sandbox	The change sets of the sandbox are combined together and appended to the parent sandbox.
5	Request to remove the sandbox	Topics assigned to the sandbox are empty and removed.

Table 9.6: Test steps for sandbox management service

9.2.2. Failover functionality

Main functions defined above are relatively easy to test. The next important aspect of the testing is the failover-functionality. The Sandbox Management Service is intended to be state-less – the entire state is represented on the Integration Layer to any point of time. To achieve high performance implementation developers often use caching strategies and "copy" the IL-state in the local memory of the application.

The Sandbox Management Service requires transactional behaviour, e.g. if it creates a new sandbox is shall open three different topics, and append the new sandbox configuration into the sandbox-list in a fourth topic. If one of the steps fails, the already implemented steps must be rolled back. The most interesting aspect in this context represents a service/node crash and restarting a new service instance on some other node. In this case the Sandbox Management Service must identify the current state and continue with implementation of

already started transactions. The logic behind this process could be quite complicated, so a set of dedicated test cases must be provided.

The Sandbox Management Service manages several transactions in parallel:

- sandbox creation/removal;
- management of one sandbox – accepting change requests from concurrent users;
- management of sandbox acceptance – "moving" the content from one sandbox to another one.

One possible state sequence for sandbox creation process is shown in Figure 9.2. The Sandbox Management Service can fail/crash at any arrow connecting activities in the sandbox creation process. Failures during the activities are managed by Integration Layer – it ensures, that value modification message is either accepted properly or ignored.

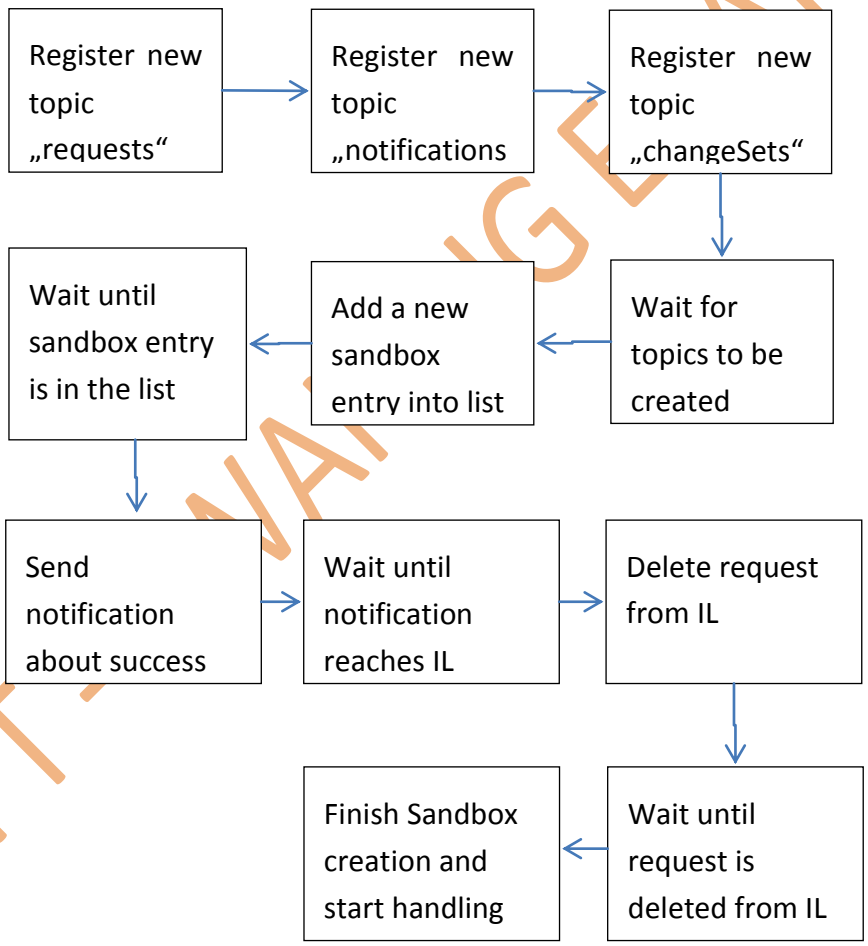

Figure 9.2: Steps in the process of sandbox creation

In Figure 9.2 ten arrows are available, so the newly started Service shall be able to continue the sandbox creation at any of them. An additional difficulty is that, the service implementation could have a different sequence of the steps, e. g. first create a changeSets-Topic and then the Notifications-Topic, so the number of possible states at the service start

is 2^4=16: each of the six aspects (topics, entry in the sandbox list, notification etc.) is either there or not. An exhaustive test would require coverage of each of them.

As the Sandbox Management Service manages three transactions concurrently (listed above) the number of possible states increases further to theoretically 3^16. Assuming that the process handling is independent from each other it is a reasonable assumption to consider only 3*16=48 states. Creation of such initial states can be easily automated, so running of 50 test cases for failover functionality for this crucial service is still reasonable.

A general approach in this case is to prepare the state on the Integration Layer, start the Sandbox Management Service and analyse the result after some time. Examples of such tests are shown in Appendix C.

10. System tests

According to [Myers et al 2011]

"the purpose of a system test is to show that the product is inconsistent with its original objectives."

The main issue in this definition is the term "objectives" as the documents describing objectives of a product do not contain precise description of the product's external interfaces needed to define test scripts. Therefore [Myers et al 2011] proposes to use the user documentation to formulate the test cases: "design the system test by analysing the objectives; formulate test cases by analysing the user documentation".

At the current state of the In2Rail project, the objectives of IL are described in [In2Rail D8.3], [In2Rail D8.4] and objectives of AF in [In2Rail D8.5] and [In2Rail D8.7]. The user documentation for IL and AF is not available as it depends on specific products selected as basis for IL and AF. The test cases were defined based on the selected products Hazelcast for IL and Docker Swarm for AF.

The system tests comprise several test categories (see Table 10.1). Some of the tests for the prototype in the proof-of-concept were automated in the project, while others were implemented manually.

Category	Description	Test method in In2Rail WP7
Facility	Ensure that the functionality in the objectives is implemented.	Manual
Volume	Subject the program to abnormally large volumes of data to process.	Automated
Stress	Subject the program to abnormally large loads.	Automated
Usability	Determine how well the end user can interact with the program.	Manual
Security	Try to subvert the program's security measures.	Ignored
Performance	Determine whether the program meets response and throughput requirements.	Automated
Storage	Ensure the program correctly manages its storage needs, both system and physical.	Automated
Configuration	Check that the program performs adequately on the recommended configurations.	Automated
Compatibility	Determine whether new versions of the program are compatible with previous releases.	Manual
Installation	Ensure the installation methods work on all supported platforms.	Manual
Reliability	Determine whether the program meets reliability specifications such as uptime and MTBF.	Automated
Recovery	Test whether the system's recovery facilities work as	Manual

Category	Description	Test method in In2Rail WP7
	designed.	
Maintenance	Determine whether the application correctly provides mechanisms to yield data on events requiring technical support.	Manual
Documentation	Validate the accuracy of all user documentation.	Ignored
Procedure	Determine the accuracy of special procedures required to use of maintain the program.	Ignored

Table 10.1: Categories of test cases according to [Myers et al 2011]

A possible test case is shown in Table 10.2. From the table follows that this kind of tests is planned to be implemented manually at least in the context of the proof of concept. In this test aspects facility, performance, and usability were evaluated.

Test script 1 – System test					
Test Step	**Action**	**Expected Result**	**Result OK/nOK**		**Remark**
1	Start 10 operator's workstations	Operator's workstations are running, the users can login in. The views are empty.	☐	☐	
2	Start Integration Layer	IL is running. Running operator's workstations not the status "connected".	☐	☐	Stress-test for IL-login
3	Import offline timetable	The IL-Explorer shows the published request for timetable import. Operator's workstations show nothing.	☐	☐	
4	Start timetable management service	Timetable management is running. Operator's workstations show initial offline timetable available.	☐	☐	
5	Start persistence service	The persistence service is running. On the assigned volume there are files for each "persistent" topic on IL.	☐	☐	
6	Create production timetable out of the planned timetable on one of the operator's workstations.	All operators' workstations are able to show production timetable.	☐	☐	
7	Start Automatic route setting	ARS is running. Current production trips are imported.	☐	☐	
8	Start traffic emulation service	Emulation system is started. The operator's workstations show current state of traffic – train positions, signal aspects, switch positions.	☐	☐	

Test script 1 – System test					
Test Step	**Action**	**Expected Result**	**Result OK/nOK**		**Remark**
9	Start further 10 operators' workstations	The new workstations have the same state as already started once. The CPU load on nodes running Integration Layer increases by factor 2 maximum. No additional delays are detectable for manual operations on workstations.	☐	☐	
10	Switch off one of the nodes running Integration Layer	The CPU load on remaining node increases by factor 2 maximum. No client observed lost connection. All clients continue operation without interruption.	☐	☐	

Table 10.2: Example test script for manual system test of Integration Layer

Using the setup in Figure 10.1 automated tests for Volume, Stress were created by providing emulated load from operator and traffic processes.

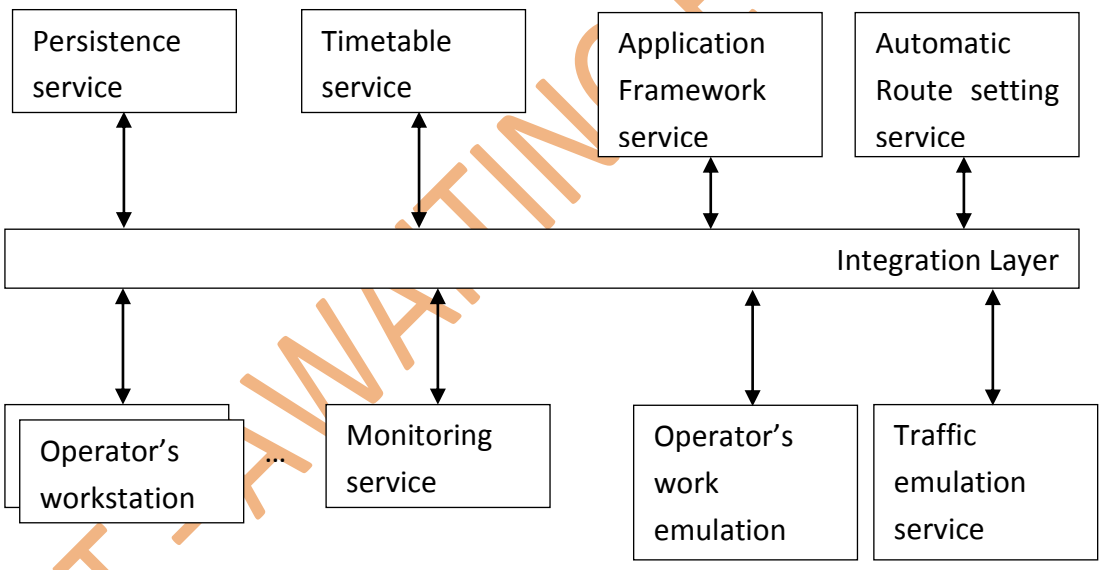

Figure 10.1: Setup for testing IL-objectives

The achieved performance was evaluated by the monitoring service, which observed:

- performance of the services by compares time intervals between published requests and responses on the integration layer;
- performance of the Integration Layer by comparing time stamps issued by the message writer and message arrival times at the service.

Achieved results were then compared with the system requirements defined in [In2Rail D8.1].

11. Conclusions

Hard and expensive testing of a multi-vendor distributed system can gain a great advantage from the selected architecture for future TMS. Using a simple test script it allows to automatically deploy, configure, connect, stimulate and observe the behaviour of the building blocks. This introduces repeatability of the testing process, reduces update times and the overall life-cycle costs.

Standardisation of the communication technology opens the TMS market for new functions coming from different vendors. But only the automated tests allow a reasonable integration and management of a multi-vendor installation in context of system with high availability, high performance, high security, and high safety requirements.

The market of testing automation solutions is big as well. COTS tools allow structuring test cases, simplify observing the system behaviour and provide extensive representation of the test results. To enable this functionality the Integration Layer with a high performance specific API is extended with a REST/JSON-Interface. Most of the unit, integration and functional test can be covered through this interface. To evaluate volume, stress and performance aspects in system tests the high performance API is still the preferred solution.

The next step in the testing automation would be integration of the testing results into Integration Layer. If each vendor delivers an installation test together with his service, the test results representation shall be harmonized and “stored” in Integration Layer. This would create an impression of an integral system hiding “specialities” of each service vendor from the maintenance team.

The future Shift2Rail projects will cover different aspects and functionalities building a future TMS. This document shows to the developers how to take advantages of the TMS-architecture to simplify and automate testing. This could build a basis for demonstrating the achieved TRL.

12. References

[AmundsenMclarty2016]	M. Amundsen, M. Mclarty: Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly, 2016
[Bassett2015]	L. Bassett: Introduction to JavaScript Object Notation, O'Reilly, 2015
[Cambell2015]	E. Cambell: Microservices Architecture: make the architecture of a software as simple as possible, 2015.
[Cantelon et al 2017]	M. Cantelon, A. Young, M. Harter, TJ _Holowaychuk, N. Rajlich : Node.js in Action, Second Edition.
[Chakram2018]	http://dareid.github.io/chakram/
[Django2018]	http://www.django-rest-framework.org
[Farcic2017]	V. Farcic: The DEVOPS 2.1 Toolkit: building, testing, deploying, and monitoring services inside Docker Swarm Cluster. Leanpub, 2017
[Frisby.js2018]	www.frisbyjs.com
[In2Rail D7.5]	In2Rail Project, Grant Agreement H2020-MG-2014-635900 – D 7.5 Evaluation of the proof of concept – 30/04/2018 http://www.In2Rail.eu/
[In2Rail D8.1]	In2Rail Project, Grant Agreement H2020-MG-2014-635900 – D 8.1 Requirements for the Integration Layer – 30/10/2016 http://www.In2Rail.eu/
[In2Rail D8.3]	In2Rail Project, Grant Agreement H2020-MG-2014-635900 – D 8.3 Description of Integration Layer and Constituents – 30/04/2018 http://www.In2Rail.eu/
[In2Rail D8.4]	In2Rail Project, Grant Agreement H2020-MG-2014-635900 – D 8.4 Interface Control Document for Integration Layer Interfaces, external/Web interfaces and Dynamic Demand Service– 30/04/2018 http://www.In2Rail.eu/
[In2Rail D8.5]	In2Rail Project, Grant Agreement H2020-MG-2014-635900 – D 8.5 Requirements for the Generic Application Framework – 30/09/2016, http://www.In2Rail.eu/
[In2Rail D8.6]	In2Rail Project, Grant Agreement H2020-MG-2014-635900 – D 8.6 Description of the Generic Application Framework and its constituents – 30/07/2017, http://www.In2Rail.eu/
[In2Rail D8.7]	In2Rail Project, Grant Agreement H2020-MG-2014-635900 – D 8.7 Interface Control Document (ICD) for Application-specific Interfaces – 30/07/2017, http://www.In2Rail.eu/
[In2Rail D9.1]	In2Rail Project, Grant Agreement H2020-MG-2014-635900 – D 9.1 Asset status representation– 30/04/2018, http://www.In2Rail.eu/
[Masse2011]	M. Masse: REST API Design Rulebook, O'Reilly, 2011.
[Myers et al 2011]	G.J. Myers, C. Sandler, T. Badgett: The Art of Software Testing. 3rd Edition, Wiley, 2011
[Protobuf2017]	https://developers.google.com/protocol-buffers
[RTC2018]	https://www.ibm.com/us-en/marketplace/change-and-configuration-management

13. Appendix A

Example of an REST API tests evaluating sandbox-service.

```
var call = require('chakram');
expect = call.expect;

testConfig = require('../config.json');
expected = require('./../response.json');

describe("manage topics in Domain", function () {
    this.timeout(100000);
    var i = 0;

    it('get list of topics', function () {
        return call.get(testConfig.APP_URL +
"/internalTopics/values/").then(function(r) {
       expect(r).to.have.status(200);
       console.log("Response time " + r.responseTime + " ms");
     });
  });

  it('create sandbox Adam and write two values', function() {
     var createSBCmd = {sandboxId:"Adam", command:"CREATE_EMPTY_SANDBOX",
createInfo:{name:"Adam", persisted:false, basisSandbox:"OnPP"}}
     return call.post(testConfig.APP_URL +
"/SBMgmtCommandsOnPP/values/rcmd", createSBCmd)
     .then(function(r) {
       if(r.status != 200)
         console.log("Error reason: " + r.response.body);
       expect(r).to.have.status(200);
       return call.get(testConfig.APP_URL + "/SBMgmtListOnPP/values");})
     .then(function(r) {
       expect(r).to.have.status(200);
       var r2 = r.response.body;
       expect(r2.length).to.equal(1);
       var r3 = r2[0];
       expect(r3.key).to.equal("Adam");
       expect(r3.value.name).to.equal("Adam");
       expect(r3.value.basisSandbox).to.equal("OnPP");
       var addTTMCmd = {nextChangeSetId:0, cs: {
         timestamp:"1511132365",
         sender:"TestScript", commands:[{
           objectRef: "/",attributeId:2, value: {
                 stringValue:"testTTM5"
           }
         }]
       }};

       return call.post(testConfig.APP_URL +
"/SBRequestsOnPPAdam/values/r5", addTTMCmd).then(function(r) {
```

```
          if (r.status != 200)
            console.log("Error reason: " + r.response.body);
          expect(r).to.have.status(200);
          expect(r.responseTime).to.be.below(130);
          addTTMCmd.nextChangeSetId = 0;
          return call.get(testConfig.APP_URL +
"/SBChangeSetsOnPPAdam/values").then(function(r) {
            expect(r).to.have.status(200);
            expect(r.responseTime).to.be.below(130);
            var r2 = r.response.body;
            expect(r2.length).to.equal(1);
            var r3 = r2[0];
            expect(r3.value).to.deep.equal(addTTMCmd.cs);
            addTTMCmd.nextChangeSetId = 1;
            addTTMCmd.cs.timestamp = "1511132366";
            addTTMCmd.cs.commands[0].value.stringValue = "testTTM6";
            return call.post(testConfig.APP_URL +
"/SBRequestsOnPPAdam/values/r4", addTTMCmd).then(function(r) {
                  expect(r).to.have.status(200);
                  expect(r.responseTime).to.be.below(130);
                  return call.get(testConfig.APP_URL +
"/SBChangeSetsOnPPAdam/values").then(function(r) {
                        expect(r).to.have.status(200);
                        expect(r.response.body.length).to.equal(1);

  expect(r.response.body[0].value).to.deep.equal(addTTMCmd.cs);
                        expect(r.responseTime).to.be.below(130);
                  });
            });
          });
        });
      });
    });
});
```

14. Appendix B

Example of functional test implementation in C++, able to run with emulated IL and productive IL.

```
BOOST_AUTO_TEST_CASE(testCreationOfSBEntries)
{
   boost::asio::io_service io_service;
   {
      ImdgClientLocal client(io_service);
      Service service;
      service.setClient(&client);
      int result = service.start(config, io_service);
      boost::asio::deadline_timer testTimer(io_service);
      TestTaskQueue taskQueue(&testTimer);
      taskQueue.addTask(1000,
                        new CreateSandbox(&client, "Paul", "OnPP"));
      Model model;
      TestTask *mkCmd = new CreateSandboxCommand(&client,
                        "SBRequestsOnPPPaul", &model);
      TestTask *undoCmd = new UndoSandboxCommands(&client,
                        "SBRequestsOnPPPaul", &model);
      taskQueue.addTask(6000, mkCmd);
      for (size_t i = 1; i != 100; ++i)
      {
         int r = rnd(0, 10);
         if (r == 1)
            taskQueue.addTask(100, undoCmd);
         else
            taskQueue.addTask(rnd(10, 100), mkCmd);
      }
      taskQueue.addTask(1000, new CheckSandboxCommands(&client, "Paul",
                  "OnPP", &model));

      taskQueue.addTask(10, new CreateSandbox(&client, "OnPP", ""));
      Model parentModel;
      taskQueue.addTask(100, new AcceptSandbox(&client,
                  "SBRequestsOnPPPaul", &model, &parentModel));
      taskQueue.addTask(100, new CheckSandboxCommands(&client,
                  "OnPP", "", &parentModel));
      taskQueue.addTask(10, new CheckSandboxCommands(&client,
                  "Paul", "OnPP", &model));

      taskQueue.run();

      io_service.run();
   }
}
```

15. Appendix C

In the following the pseudo-code is shown as an example for test implementation for failover testing on Sandbox Management Service.

```
//a set of initial conditions at starting of the service after crash
for (int testCase = 0; testCase != 16; ++testCase) {
  initTestCase();
  runTestCase(testCase);
  clearTestCase();
}

//single run of one initial condition
void runTestCase(int testCase) {
  bool requestTopicAvailable = testCase & 0x1;
  bool notificationTopicAvailable = testCase & 0x2;
  bool changeSetsTopicAvailable = testCase & 0x4;
  bool sandboxEntryAvailable = testCase & 0x8;

  if (requestTopicAvailable)
      topicsList->putValue(requestTopicName, requestTopicConfig);
  if (notificationTopicAvailable)
      topicsList->putValue(notificationTopicName, notificationTopicConfig);
  if (changeSetsTopicAvailable)
      topicsList->putValue(changeSetsTopicName, changeSetsTopicConfig);
  if (sandboxEntryAvailable)
      sandboxList->putValue(sandboxName, sandboxConfig);

  //start the service
  SandboxService srv;
  srv.start();

  sleep(0.1sec); // to establish connection to IL

  //run the service by issuing a request message
  mgmtRequestTopic->put(someKeyString, createSandboxRequest);

  sleep(0.1sec); // to implement all required steps

  //validate the state on IL
  ASSERT(topicsList->find(requestTopicName) == requestTopicConfig);
  ASSERT(topicsList->find(notificationTopicName) ==
notificationTopicConfig);
  ASSERT(topicsList->find(changeSetsTopicName) == changeSetsTopicConfig);
  ASSERT(sandboxList->find(sandboxName) == sandboxConfig);
  ASSERT(mgmtRequestTopic->find(someKeyString) == nil);
  ASSERT(mgmtNotificationTopic->find(someKeyString) == successfullRequest);
}
```