AMBIENT ASSISTED LIVING, AAL

JOINT PROGRAMME

ICT-BASED SOLUTIONS FOR ADVANCEMENT OF OLDER PERSONS' INDEPENDENCE AND PARTICIPATION IN THE "SELF-SERVE SOCIETY"

D3.4

System Integration and Tests

Project acronym:	**GeTVivid**
Project full title:	**GeTVivid - Let's do things together**
Contract no.:	**AAL-2012-5-200**
Author:	**PLUS, EVISION, IRT, ISOIN, USG**
Dissemination:	**Public**

TABLE OF CONTENTS

TERMINOLOGY & ABBREVIATIONS

ACS ..... Appointment Coordination System

Docker ..... Visualization framework

HbbTV..... Hybrid Broadcast Broadband TV

Python ..... A general-purpose, high-level programming language

SCRUM..... An agile software development framework

Selenium..... A portable software testing framework for web applications

SqlAlchemy..... Python SQL toolkit and Object Relational Mapper

SSF ..... Second Screen Framework

TDD..... Test-Driven Development

QUnit ..... A JavaScript unit test framework

1. EXECUTIVE SUMMARY

1.1 Link with the objectives of the project

The involved partners adapt and integrate the developed software and hardware used on the GeTVivid platform prototype. The prototype is tested by the technical partners in T4.5 and T4.3 to ensure a high quality and robustness for the evaluation with end users in WP2. The tests are executed on different levels of the architecture to guarantee a good test coverage of the provided features. Careful testing and thorough integration of all system components are the foundation for a good user experience. It is therefore crucial for the developed prototype and the project itself.

1.2 State of the art

According to Schach [2011] there a three different approaches to integration: bottom-up, top-down and sandwich integration. Due to the system architecture bottom-up integration will be used. This means, that the backends are tested and integrated before the frontends are implemented, tested and integrated.

Beck [2003], [2004] present unit tests as part of Test-Driven Development (TDD). All components of the GeTVivid platform are tested by (semi-)automated unit tests. Most partners make use of the TDD paradigm. It is not mandatory, though.

Molyneaux [2009] suggests a six step program for performance testing:

1. Pre-engagement requirements capture
2. Test environment build
3. Transaction scripting
4. Performance test build
5. Performance test execution
6. Analyse results, report, retest

All six steps are considered during the load, stress and endurance testing of the GeTVivid platform.

2. INTRODUCTION

The GeTVivid platform has a distributed system architecture, which is discussed in D3.1. Figure 1 illustrates the basic system conception. The orange area marks the development version of the platform (used by the partners to test out new features online). The pink boxes divide the system into separately testable components. Sections 3.1 to 30 cover these components developed by the individual partner. Sections 4.1 and 4.2 discuss the integration task. Three different service backends (sections 7 to 16) take care of persistent data storage and also contain most of the business logic. Incoming HTTPS requests are mapped to corresponding backends via reverse proxy. HbbTV and Mobile clients send requests to the proxy server. Both share a common library, which takes care of client-server communication and authentication. The distributed architecture allows to introduce software tests on multiple levels, basically separate testing for each component and hierarchy level.

Section 3.1 covers the testing of the Amiona backend and frontend (Appointment Coordination System). These tests are maintained, performed and documented by the University of St. Gallen (USG).

Section 3.2 elaborates all testing related to the profiling component. Besides unit tests it also includes theoretical background on testing profiling algorithms. The profiling component as well as the related tests are developed by Ingenieria y Soluciones Informaticas del Sur, S.L (ISOIN).

All additional functionality apart from profiling and the features offered by Amiona are included in the core services developed by the Paris Lodron University of Salzburg (PLUS). Refer to Section 3.3 for the testing related to this backend component.

The HbbTV and Mobile clients are developed by the Institut für Rundfunktechnik GmbH (IRT) and Hövener & Trapp Evision GmbH (EVISION). Sections **Fehler! Textmarke nicht definiert.** and 3.5 are dedicated to the testing of these two frontends. The main target platform for the HbbTV client during the field trials is a set-top-box developed by TARA Systems (see http://www.tara-systems.de/). Hence, section **Fehler! Textmarke nicht definiert.** also discusses testing and integration issues related to this set-top-box.

The community manager web interface is an administrative tool for the platform operator. It is developed and tested by PLUS. See Section 30 for details concerning the test and integration approach.

The overall tests are divided into automated integration tests and manual system tests. The first part is related to the development of the client-server communication infrastructure, which is developed by PLUS. Detailed information on the testing approach and the used testing framework are documented in Section 4.1. The integration testing covers all aspects of the system accessible to HbbTV and Mobile client. Section 4.2 describes the manual testing approach of the performed system tests. These tests cover the system from a user perspective. Therefore this can be considered the final phase of technical testing.

The development of the GeTVivid platform is organized in “sprints” and is inspired by the SCRUM development process. At the beginning of each sprint a list of tasks is defined for all involved parties. The defined tasks are selected in a manner that they are completable within one sprint. Each sprint within the GeTVivid project starts and ends with the monthly technical partner telco. The sprints are identified by the corresponding year plus month (e.g., Sprint 2014-08). The manual tests are performed after each sprint. All features (e.g., “Create an Informal Short-term Offer”) that pass the manual system test are considered complete and ready to use.

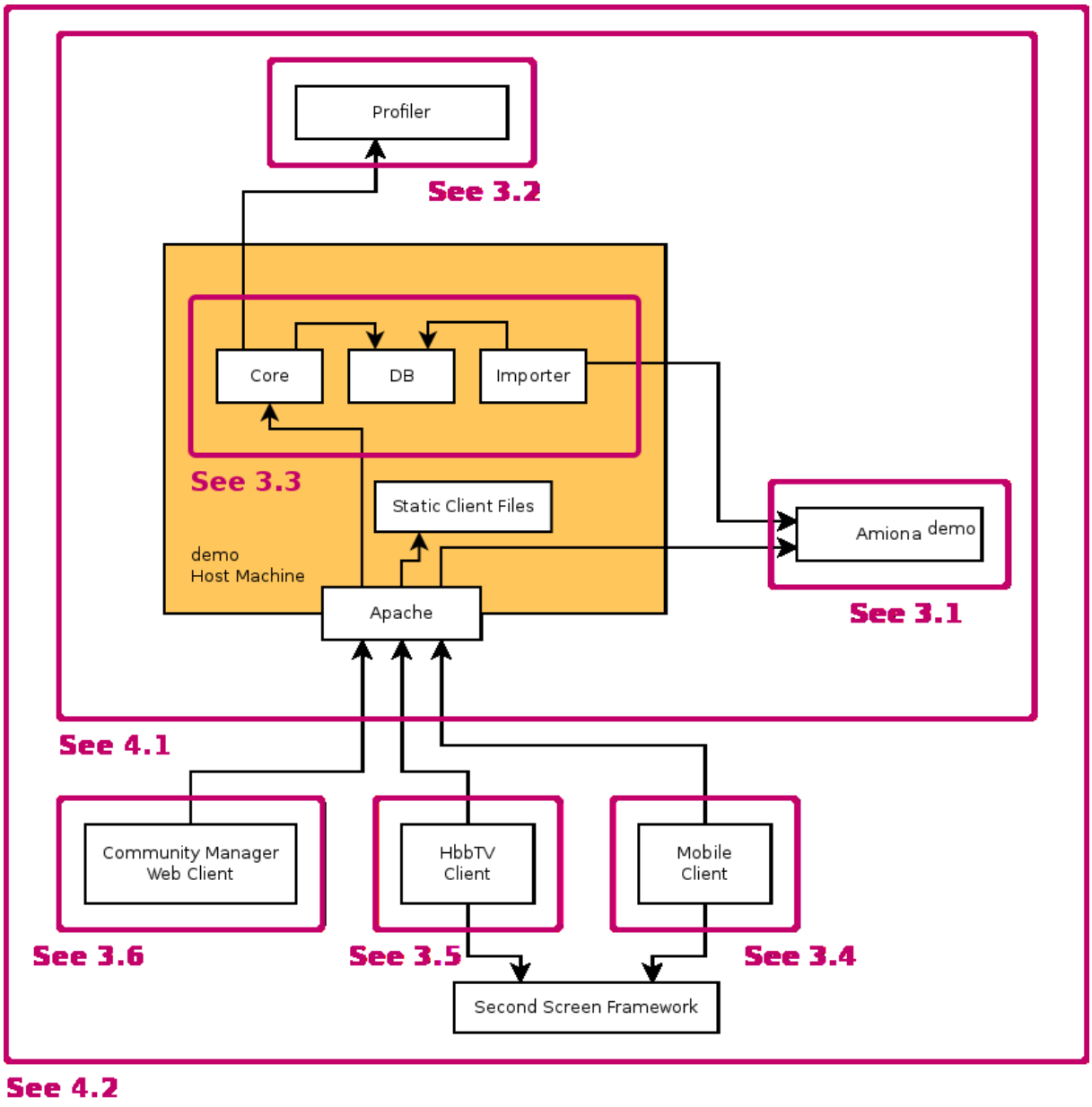

Figure 1: Test structure based on architecture. Numbers refer to sections in this document.

3. COMPONENT TESTS

The following sections cover the GeTVivid platform components developed by the technical partners. The ACS is developed by USG, the Profiler by ISOIN, core functionality as well as the community manager web interface by PLUS, the HbbTV client by IRT, and the Mobile client by EVISION. The author of each section is the partner responsible for the component. Each section discusses the used testing approach and its results.

3.1 ACS

3.1.1 Testing approach

The Appointment Coordination System (ACS) consists of two main components: a decoupled backend and a web-based frontend (also see D3.3 for further respectively more detailed information). The backend can be seen as the actual core of the system, being used only as a single instance even for larger applications. Frontends, on the other side, can occur in multiple versions in parallel (all connected to a joint backend instance). As defined by the user stories of the project (see Appendix D3.1), the ACS standard frontend is planned to be used solely by professional service providers, having additional frontends like the TV and mobile client connected to the shared backend instance for other stakeholders. Associated with the logical separation of frontend and backend are two rather diverse testing techniques, namely manual to semi-automated frontend tests and fully-automated module tests for the backend. Both approaches will be further explained in the following sections.

3.1.1.1 Frontend testing

ACS's standard web-based frontend is a responsive HTML5 component. The selection of this set of technologies is based on two reasons: First, HTML is the overall standard for web- respectively browser-related applications and version 5 the latest release. Second, responsiveness has been used in order to address certain peculiarities of different devices (standard computer, smartphones, etc.) without eventually developing multiple UIs.

Owed by this selection, however, is an increased complexity for automated tests. While the variability of the UI (due to responsiveness) as well as differences regarding the presentation of a certain HTML snippet in various browsers eventually argues for the application of automated tests, both aspects in combination increase the complexity significantly. This is, beside others, due to the fact that e.g. absolute positions of control elements cannot be used as they might differ or change along with the screen size, and the selection of control elements solely based on IDs is not always possible as functionalities are sometimes summarized under an additional hierarchy-level for mobile devices (e.g., calendar functionalities or the list of requests are not directly available).

In order to take adequate account of both, the complexity of automated tests, and the necessity and duty of ensuring the functionality, a hybrid (automated and manual) yet rather unstructured approach is used for frontend-testing. Based on the web-browser automation and testing tool Selenium (http://seleniumhq.org), a collection of test-cases has been implemented respectively recorded and is also further maintained. These tests refer to rather simple yet crucial use-cases like the registration process and are developed – where possible and meaningful – simultaneously with the development of the respective functionality. The entire collection is tested prior to each release that is deployed to productive instances of ACS; adapted subsets are also applied on a daily

basis during sprints in order to avoid long bug-fixing phases at the end of each sprint.

After the automated tests, nearly the whole set of available functionalities is tested manually as last step of each sprint phase and hence again prior to each release on productive instances. Cases for manual tests are derived as a first step from ACS's underlying process engine (hence the component defining which actions should be allowed based on which preconditions). Figure 2 shows an exemplary extract of possible paths of the standard request process. Additionally to this visual representation, a guideline further detailing cases (e.g., defining positive and negative scenarios in terms of compulsory fields) is used as a checklist.

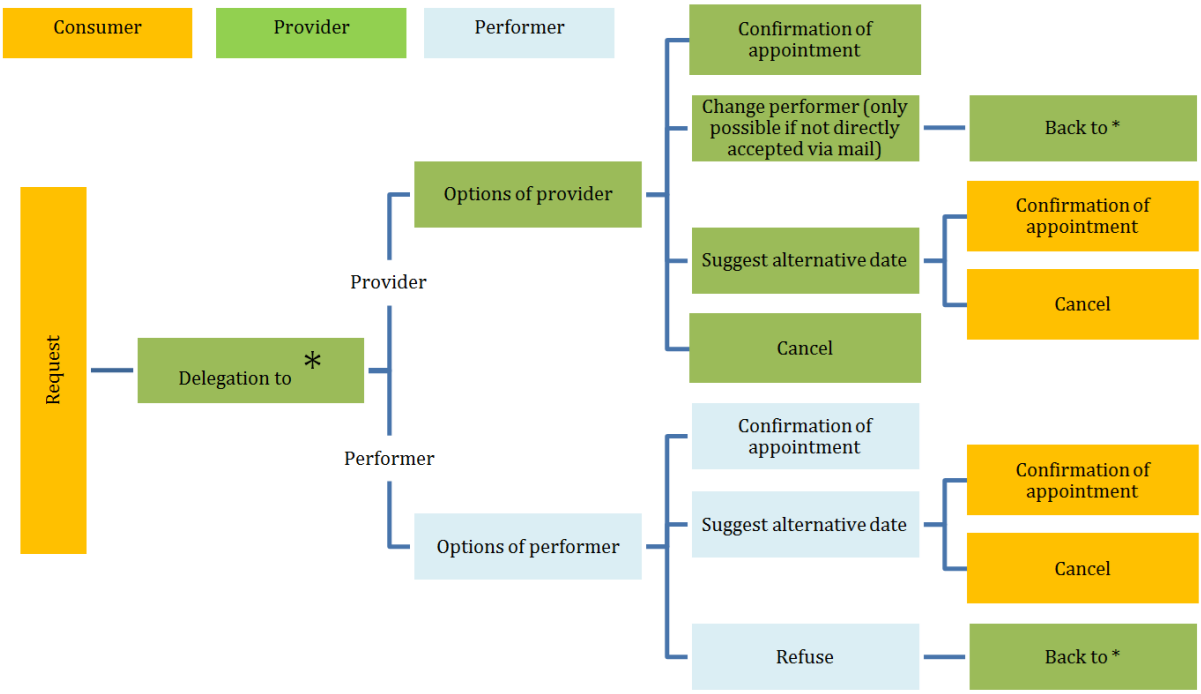

Figure 2: Extract of test paths derived from ACS's process engine

Even though this hybrid end especially the manual approach might seem to be too unstructured, it is necessary for two reasons:

1. ACS offers functionalities, which are very difficult and complex to cover in automatic test cases (e.g., the postponement of appointments over the turn of the years).
2. Automatic test cases can only be used in order to test based on determinism. In order to consider e.g. the meaningfulness of system messages, captions etc., manual tests will always be required.

Frontend related tests are only performed for functionalities that are eventually covered by the ACS frontend. Additional project-specific functionalities (e.g., the demand-driven process) are only available via third-party frontends.

3.1.1.2 Backend testing

The ACS backend represents a Python based application, running as a central server-solution. As SCRUM is applied as the fundamental process model respectively development framework, the development process corresponds to test-driven approaches with continuous integration. This means that for backend related tasks, tests are

designed and built prior to the actual development and are hence immediately available when a new functionality is finished. This approach is considered appropriate and crucial for two reasons: First, in practice the agile environment tends to be applied as excuse for missing specifications (which makes testing practically impossible as the anticipated behaviour of the system is not defined). By developing test-cases prior to the actual functionality, developers and project managers are forced to properly think through process variants and dependencies and are hence defining a minimum level of specification. Second, especially in the field of agile adaptions and changes, dependencies and coherencies are very difficult to track. In such situations existing module tests help to at least ensure the correct in terms of expected system behaviour.

However, considering the effort that is required in order to strictly follow the idea of having (preferably complete and covering) test-cases prior to each development step, the idea has been partly softened for ACS. This can be seen, e.g., by the achieved coverage (see section 10). Yet it is planned to enrich the test-base retrospectively and therefore facilitating further developments.

Within the test environment, the deduction and preparation of cases follows the structure of ACS (see D3.3 for further details). Referring to the range of functions of the backend and the logical separation of data, process logic, and web-services, three main testing scopes can be identified:

1. Data Layer

   Package: acs.models

   Data layer related business entities are tested considering their mapping functionalities from the relational model on the database level to the object model within the applications data processing based on Python and SqlAlchemy as a mapping tool. Tests are covering both directions: persisting data and retrieving data.

2. Web Services

   Package: acs.views

   The majority of web-services representing the business backend of ACS are covered by unit tests. Exceptions for this strategy are almost solely caused by web-services which are tightly coupled with external dependencies and hence not injectable to the unit tests in a clean way. Technically this means that the test preparation costs would significantly exceed potential benefits.

3. Appointment Coordination Base Process

   Package: acs.views.appointment_process

   Having the appointment coordination process as the core operation of ACS, the processes and modules related this functionalities are tested in a higher level of details.

All tests are – just as ACS itself – written in Python code and conducted in the following way:

1. Manually added by the developer as a precondition for the eventual implementation of functionalities (TDD).
2. Automatically performed by the build server (Jenkins). A test run is started after each change introduced to the main code branch. In case of failures, reports are sent directly to all developers.

3.1.2 Intermediate results

3.1.2.1 Unit tests

The following intermediate results refer to: March 2015. The results refer further to the backend testing and hence, to results of automated unit tests. Results of frontend tests are not presented here as issues are directly reported to the developers and usually solved prior to the respective release.

Figure 3 shows the coverage rate based on the number of statements per module. In contrast to lines of code (LOC), statements refer to the instructions made and hence to calculations (in a broader understanding) that have to be performed.

A coverage rate of 100%, e.g., means that each statement of the module is covered by at least one test-case, but it might be covered multiple times. A lower coverage rate, on the other hand, is not implicitly a sign of potential quality issues. Module 10 for example refers to an authentication process that has been replaced during the project by a new one, which is fully covered by test cases (Module 23).

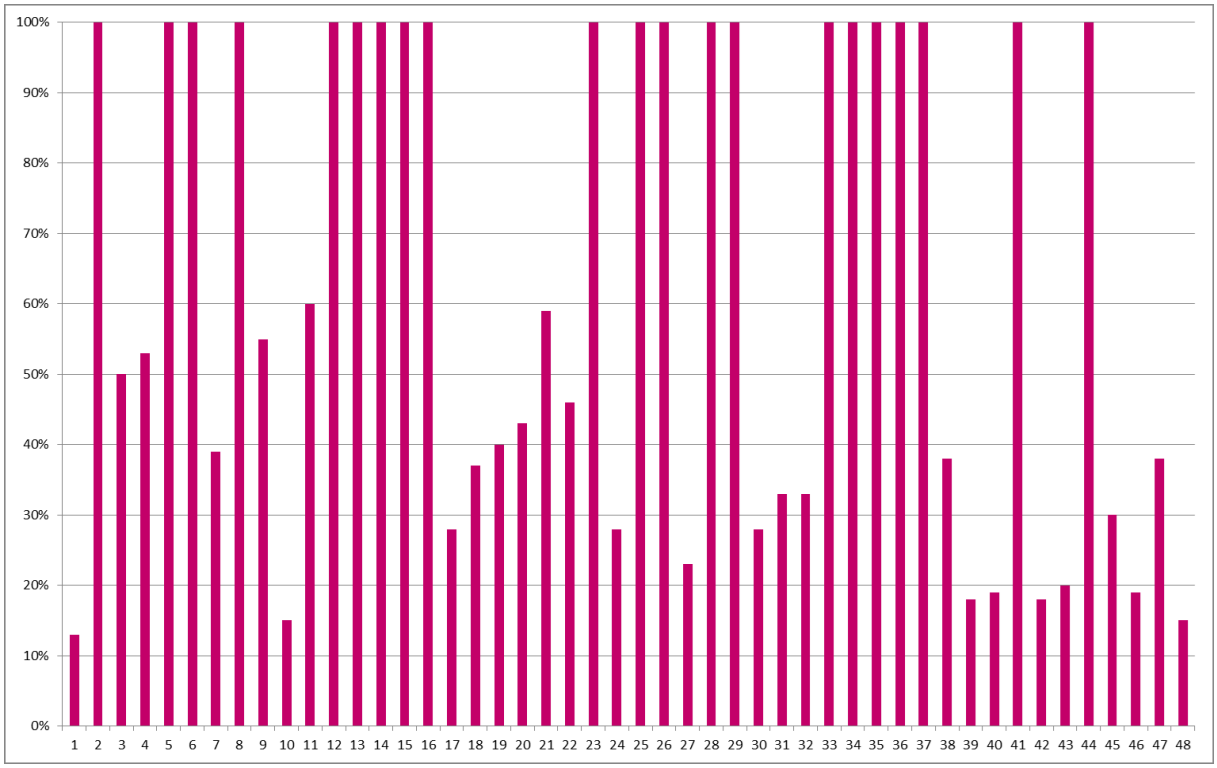

Figure 3: Intermediate results - module test coverage

The names of packages have been replaced by IDs in order to simplify the figure. Table 1 shows the mapping of IDs to package names.

ID	Name
[01]	acs
[02]	acs.foundation
[03]	acs.foundation.helpers
[04]	acs.foundation.ldap_cfg
[05]	acs.models
[06]	acs.models.abstract_models
[07]	acs.models.authentication
[08]	acs.models.enums
[09]	acs.models.helpers
[10]	acs.models.ldap_helper
[11]	acs.models.models
[12]	acs.models.tables
[13]	acs.scripts
[14]	acs.templates
[15]	acs.templates.mails
[16]	acs.tools
[17]	acs.tools.basic_authentication_policy
[18]	acs.tools.bearer_authentication_policy
[19]	acs.tools.cipher_helper
[20]	acs.tools.default_age_policy
[21]	acs.tools.default_password_policy
[22]	acs.tools.helpers
[23]	acs.tools.pyoauth2
[24]	acs.tools.request_helpers
[25]	acs.tools.singletons
[26]	acs.tools.templates
[27]	acs.tools.templates.mail_templating
[28]	acs.tools.templates.models
[29]	acs.tools.templates.models.appointment_process_email
[30]	acs.tools.templates.models.mail_message
[31]	acs.tools.templates.models.register_consumer_message
[32]	acs.tools.templates.models.register_provider_message
[33]	acs.tools.templates.models.system_mail
[34]	acs.tools.templates.models.system_mail.account
[35]	acs.tools.templates.models.system_mail.erp
[36]	acs.views
[37]	acs.views.appointment_process
[38]	acs.views.consumer
[39]	acs.views.consumer_management
[40]	acs.views.consumer_management_validation
[41]	acs.views.erp
[42]	acs.views.phone_managment
[43]	acs.views.provider_management
[44]	acs.views.tools
[45]	acs.views.tools.decorators
[46]	acs.views.tools.helpers

[47] acs.views.tools.validation_schema
[48] acs.views.tools.validation_tools

Table 1: ACS packages

3.1.2.2 Web services tests

Additionally to the (internal) unit tests, (external) web-service tests are performed. As counterpart to the internal testing, they are designed in order to simulate the interaction with other systems and hence focus primarily on third-party related functionalities. All other basic functions are already fully covered by the integrated frontend and backend tests. Where possible, test calls are chained for automation purposes (e.g. before a service is requested the service is created).

Table 2 gives a brief overview of the used web-service chains. The list is supposed to give an idea of the applied strategy. Due to the scope of the prepared chains and the redundancy of large parts of it (sub-chains like create service > retrieve service details etc. occur in many places), a complete list is not presented here. Call chains are described between “Unregistered Users” (U), “Consumers” (C), and “Providers” (P). Where multiple users with the same user role are utilized, they are distinguished by consecutive numbers.

Purpose	Call chains
Registration (especially for the distinction between professional and informal providers).	• U [Register consumer] > U [Login] > {err} > U [Activate consumer] > C [Login] • U [Register provider] > U [Login] > {err} > U [Activate provider] > P [Login]
User data management (especially for the distinction between professional and informal providers).	• C [Login] > C [Retrieve session information] • P [Login] > P [Retrieve session information]
Service management.	• P [Create professional service] > P [Retrieve service] > P [Edit professional service] > P [Retrieve service] > P [Delete service] • P [Create informal service] > P [Retrieve service] > P [Edit informal service] > P [Retrieve service] > P [Delete service] • C [Create informal service] > C [Retrieve service] > C [Edit informal service] > C [Retrieve service] > C [Delete service]
Demand management.	• P [Create demand] > P [Retrieve demand] > P [Edit demand] > P [Retrieve demand] > P [Delete demand] • C [Create demand] > C [Retrieve demand] > C [Edit demand] > C [Retrieve demand] > C [Delete demand] • C1 [Create demand] > C2 [Retrieve open demands] > C2 [Answer demand – provider triggered] > C1 [Accept]
Purpose	**Call chains**
Direct booking.	• C [Create bookable date] > C [Retrieve bookable date] > C [Edit bookable date] > C [Retrieve bookable date] > C [Delete bookable date] • C1 [Create service] > C1 [Create bookable date] > C1 [Link bookable date to service] > C2 [Retrieve bookable dates for service] > C2 [Request service bookable date]

	• C1 [Create service] > C1 [Create bookable date] > C1 [Link bookable date to service] > C1 [Edit bookable date] > C2 [Retrieve bookable dates for service] > {empty} > C1 [Link bookable date to service] > C2 [Retrieve bookable dates for service] > C2 Request service bookable date] • C1 [Create demand] > C1 [Create bookable date] > C1 [Link bookable date to demand] > C2 [Retrieve open demands] > C2 [Answer demand – divergent date] > {err} > C2 [Answer demand – bookable date]
Group management.	• C1 [Create group] > C1 [Create group-restricted service] > C2 [Retrieve service-details] > {err} > C1 [Add group member C2] > C2 [Retrieve service-details] • C1 [Create group] > C1 [Get members] > C1 [Create group-restricted service] > C2 [Request service] > {ERR} > C1 [Add group member C2] > C1 [Get members] > C2 [Request service] • C1 [Create group] > C1 [Create group-restricted demand] > C2 [Retrieve open demands] > {empty} > C1 [Add group members C2] > C2 [Retrieve open demands]

Table 2: Sample web service chains

3.2 Profiler

In computer science, profiling refers to the process of construction and application of profiles generated by computerized data analysis. This involves the use of algorithms or other mathematical techniques that allow the discovery of patterns or correlations in large quantities of data, aggregated in databases. When these patterns or correlations are used to identify or represent people, they can be called profiles. Other than a discussion of profiling technologies or population profiling, the notion of profiling in this sense is not just about the construction of profiles, but also concerns the application of group profiles to individuals, e.g. in the cases of credit scoring, price discrimination, or identification of security risks.

In this section it will be elaborated how the Profiler is tested in order to ensure that its services work properly. The test is done with different processes and techniques and also with different testing tools.

3.2.1 Profiling process

3.2.1.1 Testing the results of the data mining algorithms

In order to test the results of the data mining algorithms, a big choice of data sets is needed. A data set is just a set of data which represents a specific domain such as services suggestion based on the similarity between users or services suggestion based on the user's purchased history. A brief description of three different approaches are used in the testing process:

- **Testing with small datasets**: A small set of data is used in order to validate the results of the first testing phase. In this first phase is important to use an already classified data set or a known data set so the results of the data mining algorithms can be validated.

- **Testing with large datasets**: Different large data sets are used in this phase in order to validate the machine learning process with different sets of data. These large data sets give the opportunity to discover some errors that never occurs with small data sets. This task can be automatized (using JUnit in this case) so as to make the testing phase faster.
- **Compare with others data mining algorithms**: A comparison with other data mining algorithms is a mandatory task in order to validate the behaviour and the performance of the Profiling algorithms. These comparisons have been done with others data mining algorithms in the same scenario.

There are different data science repositories where there are a big variety of data set from different domains, which is an advantage because data mining algorithms can be tested with different kinds of data and different pre-processing approaches in order to know how achieve the best results or the lowest error rates. A lot of open databases are used for the same purpose[1]. A large data set is needed, because it will be verified that the patterns produced by the data mining algorithms are valid[2]. It is common for the data mining algorithms to have the patterns in the training set, which are not present in the general data set (this is called over-fitting). To overcome this, the evaluation uses a test set of data on which the data mining algorithms was not trained. The learned patterns are applied to this test set, and the resulting output is compared to the desired output. For example, a data mining algorithm trying to distinguish “spam” from “legitimate” emails would be trained on a training set of sample-emails. Once trained, the learned patterns would be applied to the test set of e-mails on which it had not been trained. The accuracy of the patterns can then be measured from how many emails they correctly classify.

If the learned patterns do not meet the desired standards subsequently, it is necessary to re-evaluate and change the pre-processing and data mining steps. If the learned patterns do the desired standards, then the final step is to interpret the learned patterns and turn them into knowledge.

3.2.1.2 JUnit test

JUnit is a unit testing framework for the Java programming language. JUnit has been important in test-driven development. In this project JUnit is used in order to test the Profiling API, because there are a lot of Java methods with different functionalities. For each method of the Profiling API a JUnit test is introduced, which call the method and test its functionality.

JUnit allows the execution of Java classes in a controlled manner in order to assess whether the operation of each of the class methods behaves as expected. That is, according to an input value an expected return value is evaluated; whether the class meets the specification, then JUnit will return that the class method successfully passed the test; if the expected value is different than the one the method returned during the execution, JUnit will return an error in the corresponding method. JUnit is also a mean of controlling regression testing, necessary

[1]http://dev.mysql.com/doc/index-other.htm l MySQL has some samples database which are created by the MySQL documentation team and the can be used in tutorials, samples, and so forth.

[2]http://repository.seasr.org/Datasets/UCI/arff/ A data set repository where you can download different data set in order to test your algorithms.
https://archive.ics.uci.edu/ml/datasets.html Machine Learning Repository you can browse through 307 data set.
http://www.kdnuggets.com/datasets/index.html KDnuggets Data Set Repository
https://www.kaggle.com/datasets Kaggle Data set repository

when a part of the code has been modified and you want to see that the new code meets the above requirements and has not altered its functionality after the new amendment.

With this tool, the testing process can be automatized so testing the algorithms with a big amount of large datasets and get their results is a task that not require a lot of effort due to the possibilities given by the JUnit framework.

3.2.2 Profiling services

The current architecture of the Profiler has totally changed. Previously, three main components of the Profiler were developed:

- **GeTVivid Simulator**: it was an ACS simulator which was the data provider and it also tracked the user's behaviour.
- **Profiler Client**: it was a way to create an information flow between the GeTVivid simulator and the Profiler. The Profiler Client knew how to connect with the Profiler module and it is in charge of sending the user's data to the Profiler module. When the data sent to the Profiler has been analysed, the Profiler Client returns the results to the GeTVivid Simulator. The Profiler Client worked as a bridge between the GeTVivid Simulator and the Profiler module.
- **Profiler Module**: it is the core of the data mining process, the Profiler is the place where all the machine learning tasks are executed. The Profiler took the information from the Profiler Client and return a user's suggestion based on the analysis of the data sent by the Profiler Client.

Currently there is only one component, the Profiler API. The Profiler has evolved to an API which can be accessed through a set of web services using JSON as the exchange information format. A brief example of the architecture is shown in the Figure 4.

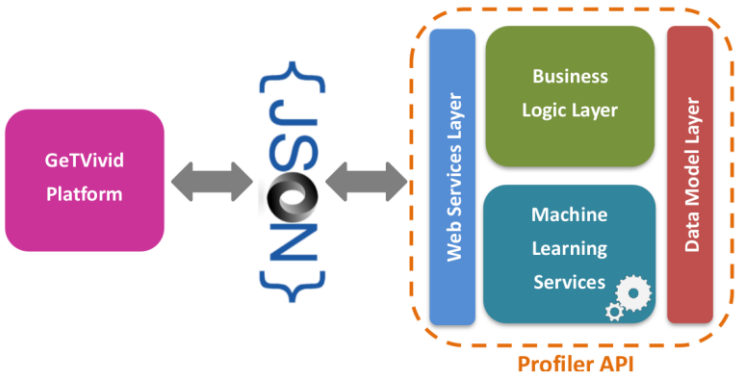

Figure 4: Current Profiler Architecture

The aim of the Profiler API is to ensure a good suggestion service to the user based on his behaviour in when the user is using the platform, the ACS tracks the user and store some user behaviour such as purchased offers or offers which were visited by the user. With all the gathered data the Machine Learning Services give a suggestion to the user.

To ensure that services work properly each of the services has to be tested independently using the JUnit Regression Testing Framework, which was introduced in Section 3.2.1.2. The following points will be checked for all services (see Table 3).

Request	Status Code	GeTVividAPIResponse
POST method correct credentials for the use of the service, well-formed XML document.	HTTP OK	CODE OK
POST method correct credentials for the use of the service, malformed XML document.	BAD REQUEST	ERROR CODE XSD
POST method using incorrect credentials for the service, well-formed XML document.	HTTP UNAUTHORIZED	null
GET method correct credentials for the use of the service, well-formed XML document.	HTTP OK	CODE OK
GET method correct credentials for the use of the service, malformed XML document.	BAD REQUEST	ERROR CODE XSD
GET method using incorrect credentials for the service, well-formed XML document.	HTTP UNAUTHORIZED	null

Table 3: Expected responses for all services

3.3 Core functionality

The core functionality covers all services that are not implemented by the ACS. It takes care of the integration of the profiling component. Apart from that, there are also endpoints for functionality related to the message service, the calendar, community manager related functionality, error logging, and other features required for integration.

3.3.1 Testing approach

The server component for core functionality is accessible through REST APIs, which can be invoked by the client applications. For this server component a separation into *data layer/repositories*, *application logic/services* and *REST service interface/controllers* was introduced:

The *data layer* relies on a local MySQL database. Some tables in the local database are frequently updated by the importer component, which keeps local data synchronized with the ACS database. For time sensitive data transactions like the validation of OAuth 2.0 session tokens corresponding REST services of the ACS are used. Thus it is necessary to introduce a more complex test environment where the underlying data layer can be changed in the remote database as well as in the local database. For this purpose a virtual test environment was set up with predefined databases in both the local and the remote database, which can be used to derive more sophisticated assertions for testing. For a more detailed description of the mentioned test environment see Section 4.1.1. The data layer will not be tested explicitly, since most of the database operations rely on Spring

Data JPA / Hibernate and have a very generic nature. Apart from that, an error in the data layer can be detected if the output of the service does not comply to the defined assertions.

The *application logic* contains the business logic needed for a REST service. Assertions about the output mostly depend on the data layer. The application logic contains for example error handling, permission management, process logic, a real-time message relay etc. Different error and border cases are tested explicitly during testing. Methods containing application logic are tested indirectly by invoking the REST service.

All tests run against *REST service interfaces*. Since the frontend clients, which access the HTTP REST APIs, are JavaScript applications, it seemed more convenient to create a JavaScript test suite instead of a pure Java approach like JUnit. At this point the unit tests for this component are incorporated into the client-side test suite, which is also used for automatic integration testing (see Section 4.1). The benefits are that no redundant test cases are introduced and that the test scenario is much closer to the production environment compared to isolated unit tests.

QUnit (http://qunitjs.com/), a JavaScript unit test framework, is used to implement all test cases in the test suite.

3.3.2 Results

The results of all passed unit tests are displayed in Figure 7. JaCoCo (http://www.eclemma.org/jacoco/), the successor of the popular code coverage tool EclEmma, is used to calculate the code coverage for each package. JaCoCo includes a Java agent that can be added as an argument to the startup command of the used Java Virtual Machine (JVM). This allows a code coverage analysis under circumstances nearly identical to the production environment. The results can be imported into most major IDEs to get a detailed view of covered code lines and instructions.

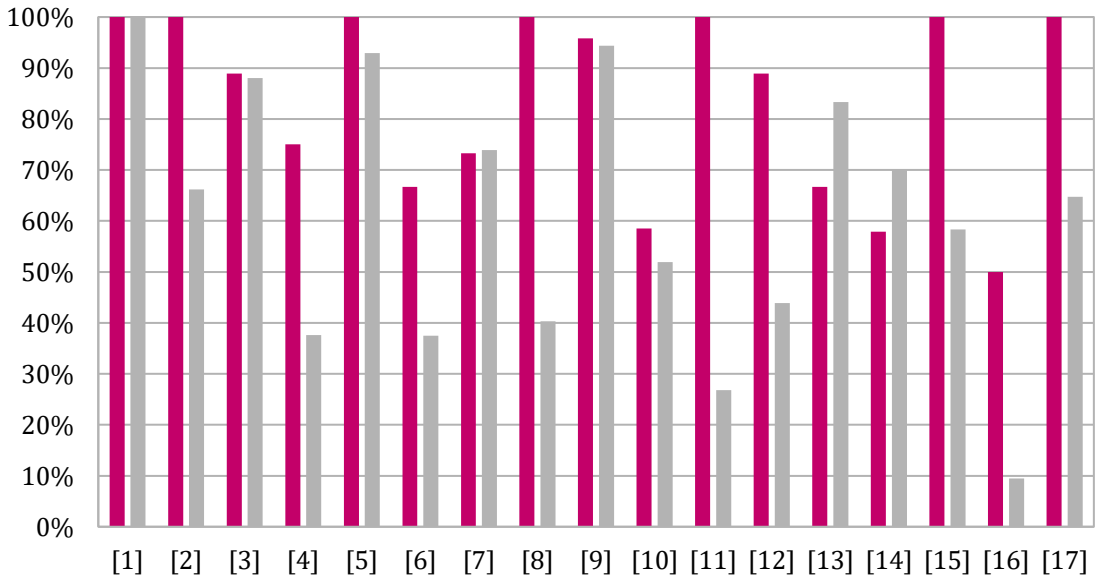

Figure 5: Line (■) and method (■) test coverage (1).

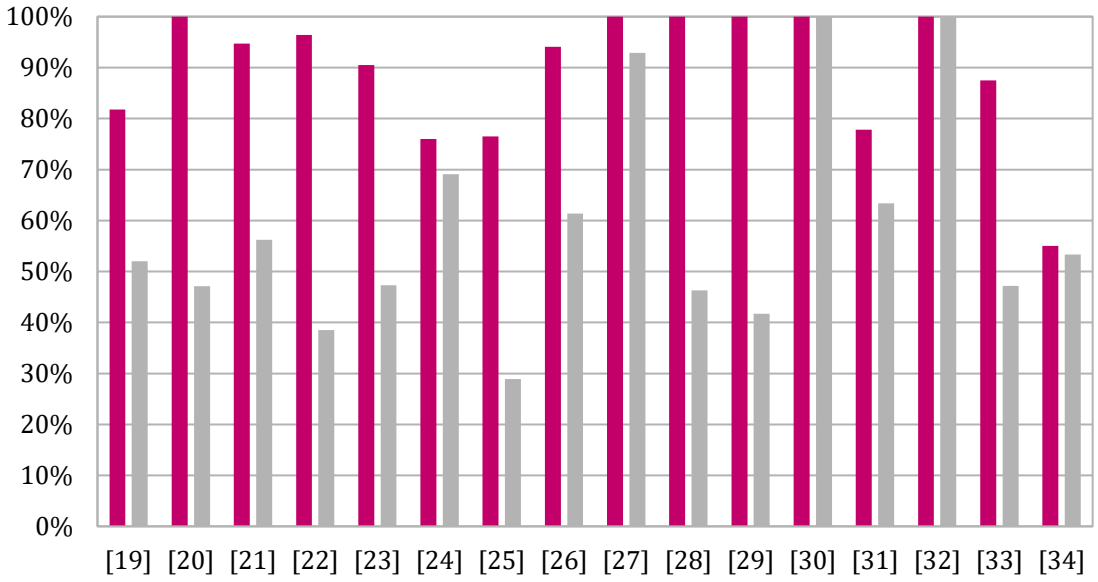

Figure 6: Line (■) and method (■) test coverage (2).

Both method and line coverage of the applied unit tests are visualized in Figure 5 and 6. All package names have been replaced with an index number to improve the legibility of the diagram. Refer to Table 4 for the full package names.

ID	Package
[1]	eu.getvivid
[2]	eu.getvivid.config
[3]	eu.getvivid.config.properties
[4]	eu.getvivid.endpoints
[5]	eu.getvivid.endpoints.models
[6]	eu.getvivid.queries.isoin
[7]	eu.getvivid.queries.isoin.models
[8]	eu.getvivid.queries.usg
[9]	eu.getvivid.queries.usg.models
[10]	eu.getvivid.repositories.models
[11]	eu.getvivid.security.auth.filters
[12]	eu.getvivid.security.auth.tokenauth
[13]	eu.getvivid.security.models
[14]	eu.getvivid.security.models.usg
[15]	eu.getvivid.services
[16]	eu.getvivid.services.admin
[17]	eu.getvivid.services.badges
[18]	eu.getvivid.services.badges.models
[19]	eu.getvivid.services.calendar
[20]	eu.getvivid.services.calendar.filters
[21]	eu.getvivid.services.calendar.functions

[22] eu.getvivid.services.calendar.models
[23] eu.getvivid.services.chat
[24] eu.getvivid.services.chat.models
[25] eu.getvivid.services.helpex
[26] eu.getvivid.services.helpex.filters
[27] eu.getvivid.services.helpex.functions
[28] eu.getvivid.services.suggestions
[29] eu.getvivid.services.suggestions.functions
[30] eu.getvivid.services.suggestions.models
[31] eu.getvivid.services.users
[32] eu.getvivid.services.users.filters
[33] eu.getvivid.tasks
[34] eu.getvivid.tools

Table 4: PLUS packages

The unit tests achieve a method test coverage of 86.82%.This also includes interfaces and classes used for data representation. Same applies to the line coverage with 61.31%. The remaining 38.69% belong to older versions of the used endpoints which are still in the code for compatibility reasons, but not tested by the most recent tests.

Figure 7: Results of core functionality unit tests.

3.4 HbbTV client

3.4.1 Test set-up and test approach

Although there are specific software tools for testing HbbTV applications in a PC browser, without using a TV set (e.g., FireHbbTV, a PC browser-plugin which emulates HbbTV devices), it is necessary to test on up-to-date end devices. Interoperability among TV hardware is a common problem, because runtime environments for these systems are not standardized and rely on conventional web technologies. For instance, there is a variety of

different TV browsers on the market with different implementations, leading to inconsistent behaviour of the applications with HTML and JavaScript.

Therefore the software tool BRAHMS (see Figure 8) is used in the evaluation process of the HbbTV client. It is a PC solution for creating and multiplexing MPEG transport streams for the DVB standard. Basically the HbbTV application URL is inserted in the broadcast stream via multiplexing (requires a hardware multiplexer card for the PC), which is afterwards played out to the TV set via an antenna cable. The TV receiver then extracts the URL and receives the data either via the broadband access or the data carousel of the broadcast stream (DSM-CC).

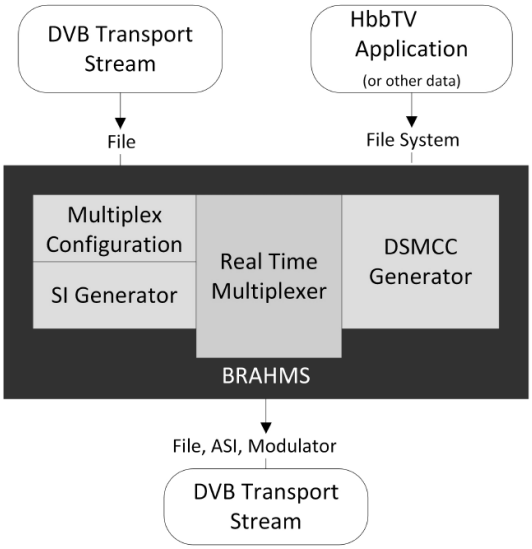

Figure 8: BRAHMS system overview

This set-up enables the developer to test in a real TV environment, making it easier to track bugs. Although most tests on various end devices sometimes result in the same outcomes, it is advisable to test on a variety of TV sets. The IRT owns a test lab with state-of-the-art TVs from the market leading manufacturers such as Samsung, LG or Sony, enabling the examination of a broad lateral cut of actual runtime environments used in actual TV implementations (see Figure 9).

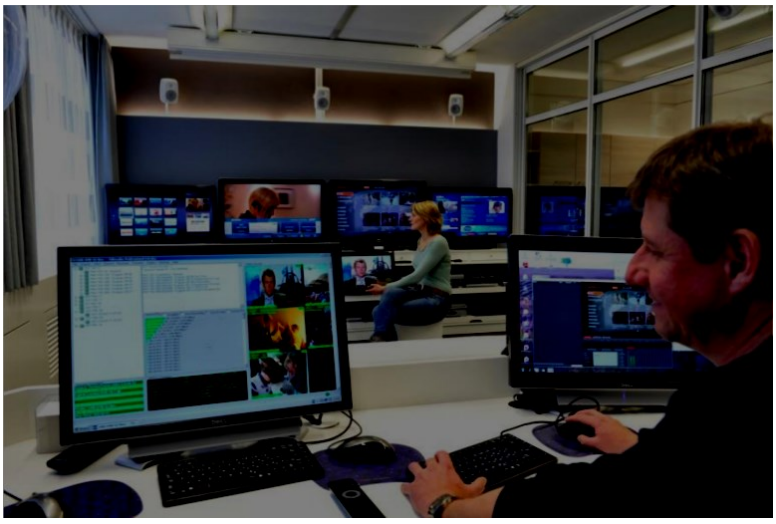

Figure 9: IRT test lab

Furthermore the IRT hosts a quarterly HbbTV interoperability workshop (see Figure 10), where representatives from over twenty different companies from across the value chain attend the event to evaluate current HbbTV applications and terminal implementations (including broadcasters, software providers and CE device manufacturers). This is an opportunity to test the GeTVivid applications on the latest end devices.

Figure 10: HbbTV interoperability workshop

3.4.2 Semi-automated testing

While the HbbTV Client is heavily focused on the frontend user interface, the basic script functionality can be tested by performing semi-automated tests on a low-level basis. These tests do not only allow to test the stability of the current system, but also ease the process of maintaining and extending the existing system without the fear of introducing new issues into the working application.

As mentioned before, testing the application thoroughly requires the application to be tested on a wide set of TV devices from various manufacturers. Unfortunately, there is no accepted standard for running unit (module) tests on HbbTV devices and the lack of a framework for doing so lead to the implementation of a custom test suite. The developed test suite can be run on any HbbTV device and can be operated with the remote control (see Figure 11). Due to the necessity of interaction with the test suite, the performed tests can only be considered semi-automated.

Figure 11: HbbTV test suite menu

Once a test has been selected, the corresponding unit tests will be executed on the HbbTV device itself (see Figure 12). This ensures that the functionality of the tested script is working as expected on the tested device or, if any of the tests failed, which part of the script is not working (see Figure 13). As a result of this process, the availability of the tested functionality can be considered certain and issues occurring on any tested device can be fixed more efficiently and effectively.

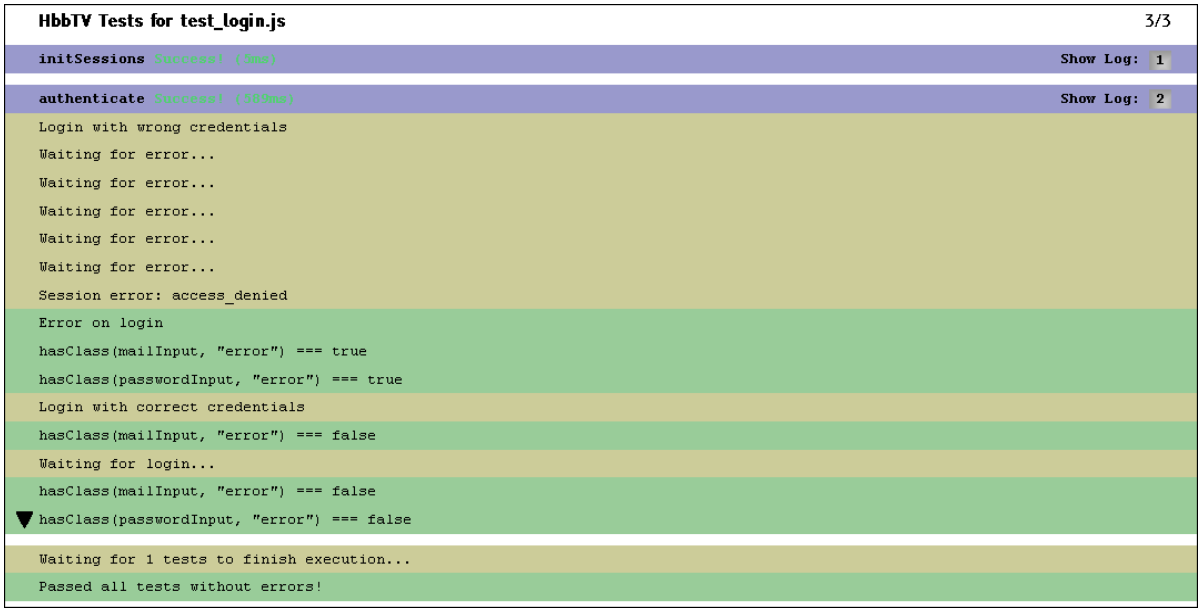

Figure 12: HbbTV unit test

For the GeTVivid application itself, a set of test cases was defined, that are executed on a wide set of HbbTV devices currently on the market. Moreover, they are performed with new HbbTV device prototypes during the Interoperability workshop (see 3.4.1), to ensure compatibility with the latest devices.

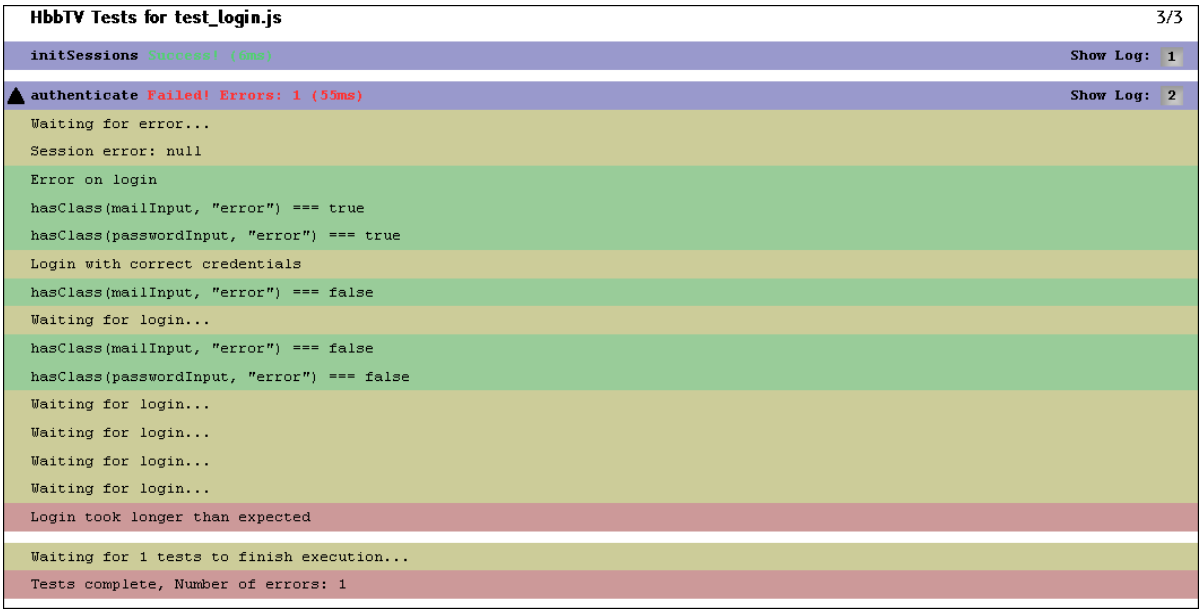

Figure 13: Failed HbbTV unit test

3.4.3 Manual testing

In addition to the semi-automatic testing procedure, manual testing allows to detect problems that cause visual inconsistencies or arise only under certain circumstances that are difficult to emulate. Usually when testing HbbTV applications on actual HbbTV devices (instead of emulators), debugging is not an option, as the devices do not provide any interface to do so. However, using the set-top-box prototype of TARA Systems GmbH, manual testing is greatly alleviated, as the box provides a debugging interface that can be used to debug HbbTV applications running on the box itself (see Figure 14). The interface provides several options, e.g. starting an HbbTV application from a URL (without any broadcast signal), remote control input and even step-by-step debugging of applications.

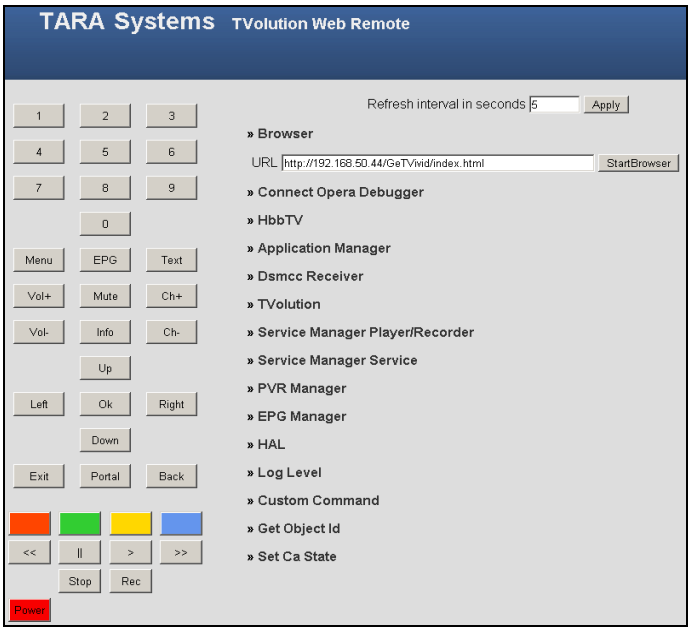

Figure 14: TARA Systems debugging interface

The interface is accessible through a browser on any computer in the same network as the box. The browser allows monitoring the state of elements in the application in order to ensure that everything is working as expected. Figure 15 depicts the monitoring of elements for the GeTVivid application.

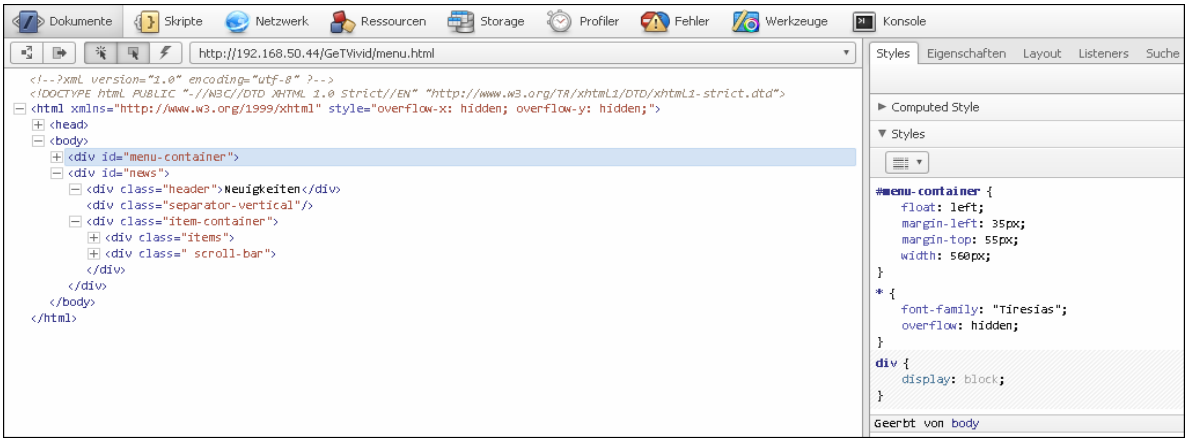

Figure 15: Manual element monitoring

Through step-by-step debugging, it is also possible to view messages of errors caused by the application or execute certain commands in the application running on the box. Some corner cases that only occur under certain circumstances can therefore be evoked more easily. Errors that occur in any of the manual test cases are raised in the debugger and can simply be traced back to the actual line of code that caused the error (see Figure 16).

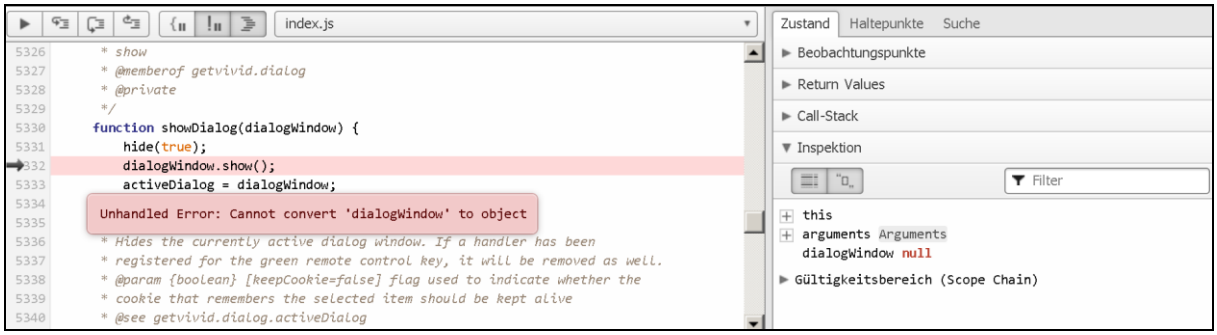

Figure 16: Error in a browser's debugger

3.4.4 Results

3.4.4.1 Semi-automated testing results

Several minor issues that are not detected during manual testing are usually identified by the semi-automated testing procedure. The source of these corner cases can be quickly determined using the test suite and all known bugs that have been detected have been fixed. By now there are test cases defined for all of the modules integrated in the GeTVivid application. The test coverage is depicted in Figure 17.

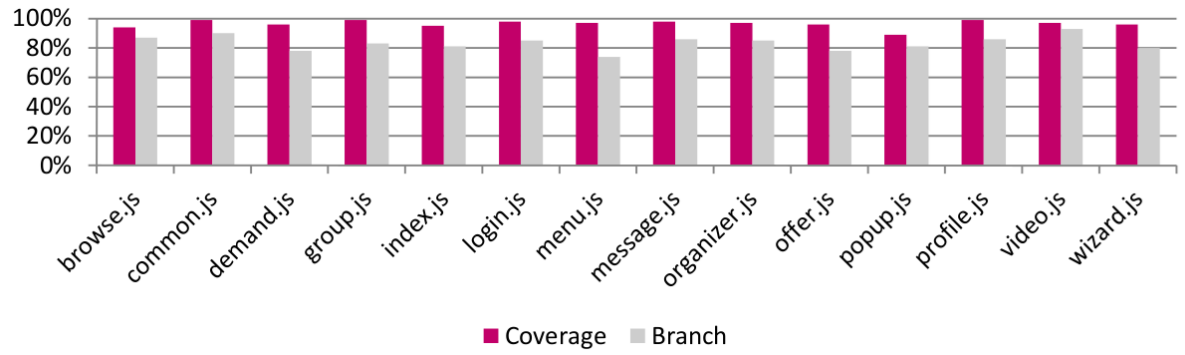

Figure 17: Module test and branch coverage

3.4.4.2 Manual testing results

Using the manual testing approach, several inconsistencies on HbbTV devices could be detected, that lead to problematic behaviour within the GeTVivid HbbTV client. The most important issues are listed below.

- Device-dependent user input mechanisms: Some manufacturers provide input mechanisms in HbbTV applications as a native widget, whereas others do not. This can be a problem, as there is no reliable technique to determine whether the user of the GeTVivid HbbTV client can enter text or not. Moreover, when providing input mechanisms specific to the HbbTV client, one has to ensure that these mechanisms do not interfere with the HbbTV device's input mechanisms.
- Application appearance: Styling information used to define the appearance of the GeTVivid HbbTV client on an HbbTV device may be altered by the device manufacturer. For instance, buttons will have an extra border around them, which causes them to take up more space than originally intended. If multiple buttons in a single page take up more space, some elements of the page may not be entirely visible anymore. Therefore, the styling information has to be provided in a way to ensure that the page is displayed correctly even under these circumstances.
- Navigation: Although the element layout for each page is the same across HbbTV devices, the navigation between elements works differently for different devices. Some devices, for example, will select the element right to the current element, even though the user intended to navigate downwards using the "down arrow"-key. Other devices will actually navigate to the intended element. This unexpected behaviour can be prevented by explicitly stating the navigation options for each navigable element on each page.
- Video playback: Playback of on-demand content is mandatory for HbbTV devices. GeTVivid makes use of that feature by providing a set of tutorial videos that help the user getting started. However, the remote control is used differently across devices when it comes to video playback, e.g. by providing separate buttons for play and pause in contrast to combining both features in one button. This and other subtleties were only discovered by manually testing and could be taken care of by covering all possible usage scenarios.

3.5 Mobile client

3.5.1 Testing approach

In accordance to the project proposal, the Mobile client is meant to be a “Second-Screen-Device” for the GeTVivid system running on an HbbTV set at the home of elderly people. In this role it is not only a display screen, but more the main input and controlling device for the whole system. It gives the user a more convenient way of interaction with the system than the remote control is able to provide.

In order to reach this goal the mobile application needs to work hand-in-hand with the application running on the television. In fact hand-in-hand means a full synchronization between the clients is necessary. So the architecture of the Mobile client is analogue to the architecture of the HbbTV client, using the Second-Screen-Framework for the synchronization. For more details about the architecture please refer to D3.1, D3.2 and D4.1.

The Mobile client uses a web-application sharing some web-components and libraries with the HbbTV application. This means that all business logic is on the backend side and is tested there and not inside the client itself. So the testing of the Mobile client was focused mainly on the frontend design and usability for elder people. Although both clients sharing widely the same web-based core application, there are still a lot of differences between them. Each client is optimized for completely different hardware devices with completely different input and controlling methods. This means the semi-automated test procedures run for the core application, including the JavaScript functionality, were widely the same as they are described in the HbbTV section. The testing of the individual input and controlling features of the both clients was done individually for each kind of client and device.

On the Mobile client side these tests were performed manually. Structured testing routines and checklists helped to keep track during the manual testing procedures. Since a SCRUM oriented development approach was used in the project, the definition of these checklist was based on the feature description in the corresponding user story. Each user story was related to a certain feature of the system in which the desired behaviour of the function was described.

In the first step, these tests were operated on hardware emulators of the development system, which are able to emulate almost every release of the operating system and hardware specification. This was the fastest and most convenient way of testing during the development process. The second step was the testing on real hardware devices. These tests proved the real end-user experience with full sensomotoric input and feedback. The tests were also run on devices with the same hardware specifications and version of operating system as they were used for the end-user test during the evaluation period. Both testing routines were run very frequently during the implementation of a new user story. Depending on complexity of the actual tasks, up to 100 test cycles of the first level could be easily recognized per day. Once a new feature had passed both levels with satisfying results, the whole application was tested again. This final internal test proved, if the new feature had any side effects on the existing features and if all possible ways of “mis-usage” were handled correctly, for example input of characters where numbers are expected and vice versa.

3.5.2 Results

Because of the high frequency of iteration cycles between development, testing, and fixing no statistic results on the frontend testing were evaluated. Nevertheless the issues that occurred during the development and deployment process of the Mobile client can be classified basically in the following categories:

- **Application appearance:**

  For the first versions of the Mobile client styling information like Cascading Style Sheets (CSS) used to define the appearance on the HbbTV device caused some unexpected behavior or appearance on the mobile device. For example, due to the fact that the special font type of the TV client is not available for the mobile application, the usage of a different font caused some unexpected line breaks, which ruined the whole interface design.

  After first experiences with this kind of issues, a general style-guide for each of the clients was worked out. In this style-guide all aspects of the UI components were defined for each of the clients. The usage of the style-guide did avoid nearly all issues of this kind for the later versions of the Mobile client.

- **Navigation and input behavior:**

  The Mobile client application is free from the navigation limitations of the HbbTV client, like the focus frame controlled by the arrow keys of the classic remote control. Data input and navigation is done via the touch sensitive display. This is why all navigation and input components of the client must be checked for proper functionality and behavior. The tests of the first versions of the client showed a general issue with the usage of the OS specific touch keyboard. The virtual touch keyboard is displayed in case a text input is required. So it happened that the keyboard had covered up the input field itself, the way that the user was not able to see anymore what he was typing. A general solution for this mayor issue was found by adding special input pages for any type of input. This kind of input logic was applied for all following versions of the Mobile client.

  Instead of the TV remote control, the mobile application is controlled via direct touch interaction. The implementation of this different interaction method is the biggest difference between the clients and had caused a lot of JavaScript exceptions in the beginning. After fixing all of these general errors, the Mobile client was able to be controlled just by touch-clicking the interface buttons.

  For the scrolling of all kinds of lists, like for example the "open offers" list, the semantic logic of the swipe action needed to be adapted to the GeTVivid needs. Finding the right settings for the swipe action parameters took a lot of trial-and-error testing, but led to an optimized solution for the special requirements of the project. So all input and interaction issues were fixed in a proper way until the development process.

- **Device and operating system (OS) specific problems:**

  Even if all versions of the Android operating system should be backward compatible, the deployment of the newest OS versions had in fact a big influence on the behavior of the GeTVivid application. So allot of new issues were caused by the change from Android 4.4.x “KitKat”, which was the actual Android version at the beginning of the development phase, to the current version Android 5.x.x “Lollipop”. Especially the scrolling method needed a complete re-implementation, as well as the media player for

the GeTVivid video tutorials. As a consequence a flawless operation of the Mobile client can now only be guaranteed for the latest version of the OS (V5.x.x).

Similar issues occurred during the usage of hardware devices of different manufacturers. Namely the "Samsung Note" tablet series caused some malfunctions during the usage of the mobile user interface, again mainly scrolling and the media player were affected. Since this issues were caused by the manufacturer himself, by alternating the Android OS in a way that is not conforming to the official standards, the mobile app could not address this kind of proprietary sub-standard. As a consequence of this third party influences, the Mobile client can only be supported for standard conforming devices, like Google Nexus or Asus ZenTab. Nevertheless, the majority of devices on the market are conforming to the standards, but from a developer perspective it is impossible to guarantee a flawless operation of the application on all available devices.

- **Proper backend connection:**

  Every new feature and function related to the backend, even if the libraries were already tested on other test stages, could cause issues and must be checked on the mobile end-user device. The testing and bug fixing of the interaction between the client side and the server side was a crucial parts of the development and testing procedure because the components of other partners were involved. Most common issues of this category were adapting the settings of the client according to the server side. The debugging process was mastered in excellent collaborative way between the project partners, so that all issues could be solved fast and easily.

- **Synchronization between the system clients (Second-Screen-Framework):**

  Since the way of text input on the Mobile client differs from TV client, the synchronization of the input was a challenging task in many ways. The text input is synced via SSF messages every 3 seconds while the user is typing. Testing of synched text in addition to make sure that the navigation of the screens afterwards is not changed was always a must to verify. Furthermore out-of-synch problems occurred because of SSF messages originated by the action were not sent properly or were sent more than once. Having in mind that one single SSF message failure causes the two clients to be out of synch and once they out of sync it is very difficult for the user to recover the proper synchronization for both clients, all possible functions had to be tested very carefully on both clients to make sure no missing or double send messages occur. This was the major testing task for final versions.

- **Usability issues regarding elderly people:**

  Basically this was not part of the internal testing process of the Mobile client, because it is part of the end-user test run during the project. Nevertheless, during development and testing it was helpful to keep always in mind that the end-users are elderly people, most probably not familiar with apps and web-applications. So the appearance and the wording of possible error messages or even the possibility of an easy recovery, in case of an unexpected failure, were designed carefully and optimized during different feedback cycles. A similar issue of this category were for example latency issues caused by an unstable Internet connection or other hardware related problems. Issues of this kind are not easy to handle or to fix, because they are mostly not under direct control of the developers. The only way to deal with these issues is to avoid them beforehand as good as possible.

3.6 Community manager web interface

The community manager web interface is the latest addition to the GeTVivid platform. This client reuses most of the endpoints that were already introduced for Mobile and HbbTV client. Thus, no dedicated endpoint tests are necessary. The interface logic as well as client-side user input validation are tested manually. Table 5 and 6 contain a short functionality and test overview for the community manager web interface.

Functionality	Test result
Dashboard	
Active demands overview	Passed
User behavior overview	Passed
System log overview	Passed
Messaging overview	Passed
Latest entity comments	Passed
User management	
Create user	Passed
Read/list informal or professional users	Passed
Update user	Passed
User behavior	Passed
User group management	
Create group	Passed
Read/list groups	Passed
Update group	Passed
Delete group	Passed
Messaging	
Send a message (to one or many)	Passed
Send an entity attachment	Passed
Receive a message	Passed

Table 5: Community manager web interface manual test (1).

Functionality	Test result
Help exchange management	
Create a demand/offer	Passed
Read/list demands/offers	Passed
Read/list active demands/offers	Passed
Update a demand/offer	Passed
Mark a demand/offer as inactive	Passed
Appointment coordination	
Perform a coordination step	Passed
Entity commenting	
Create a comment for a demand/offer/user	Passed
List comments for a demand/offer/user	Passed
Misc	
Log in as another user	Passed
Read/list system log entries	Passed

Table 6: Community manager web interface manual test (2).

Figure 18 shows a preview of the community manager web interface dashboard. Most features are represented by a dedicated panel. The menu items represent each one entry in the manual testing plan.

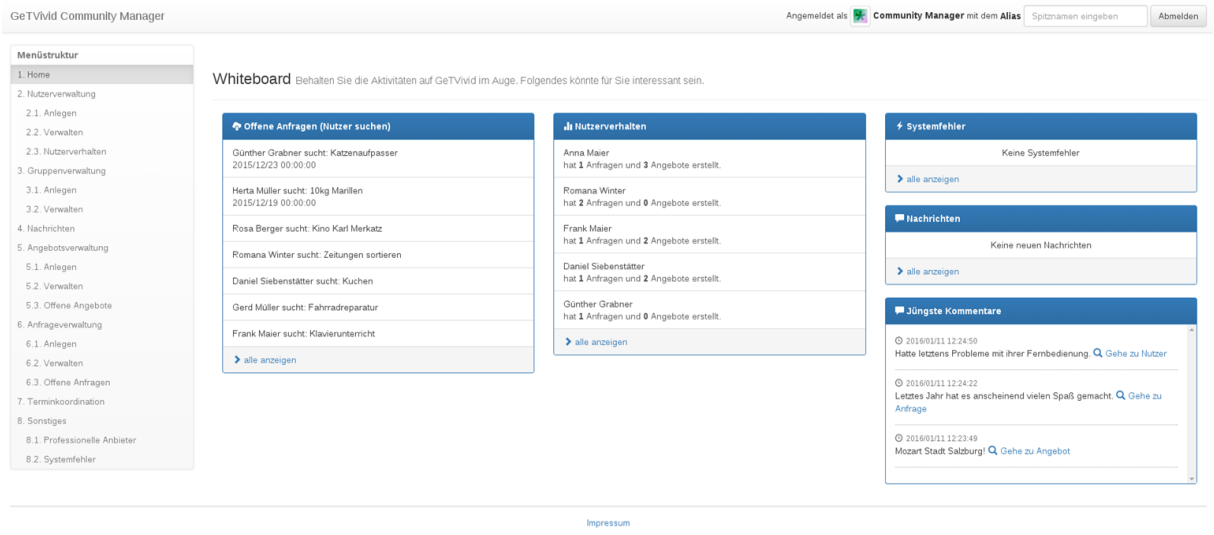

Figure 18: Screenshot of the community manager web interface.

4. SYSTEM INTEGRATION

The system integration brings all components of the previous sections together. The goal is to connect interfaces and use the same databases in all parts of the GeTVivid platform. This section is divided into automated integration tests and manual system testing. The integration tests check all endpoints accessible to the client applications and verify that a specific input leads to the desired output. Of course this can only be done in terms of data format checks and with very basic assertions about the actual data, because the system state is too complex to test with an automated test. For that reason there are also manual system tests, which use the actual end-user clients. The tester can then confirm that the outputs of the system are sensuous and the waiting time is within subjectively tolerable limits.

4.1 Automated integration tests

4.1.1 Test setup

The distributed system architecture and the amount of interactions between the system components make it difficult to test each individual component with isolated tests. A test system setup nearly identical to the production system is used for efficient automated integration testing. Since most system components are contained in Docker containers, it is possible to recreate production conditions in the test environment. Figure 19 illustrates the components of the test system. Note that it also shows components that will be discussed later in context of the sub system waiting time testing. HbbTV and Mobile client are both tested on the corresponding hardware (set-top-box and tablet) to simulate real frontend usage.

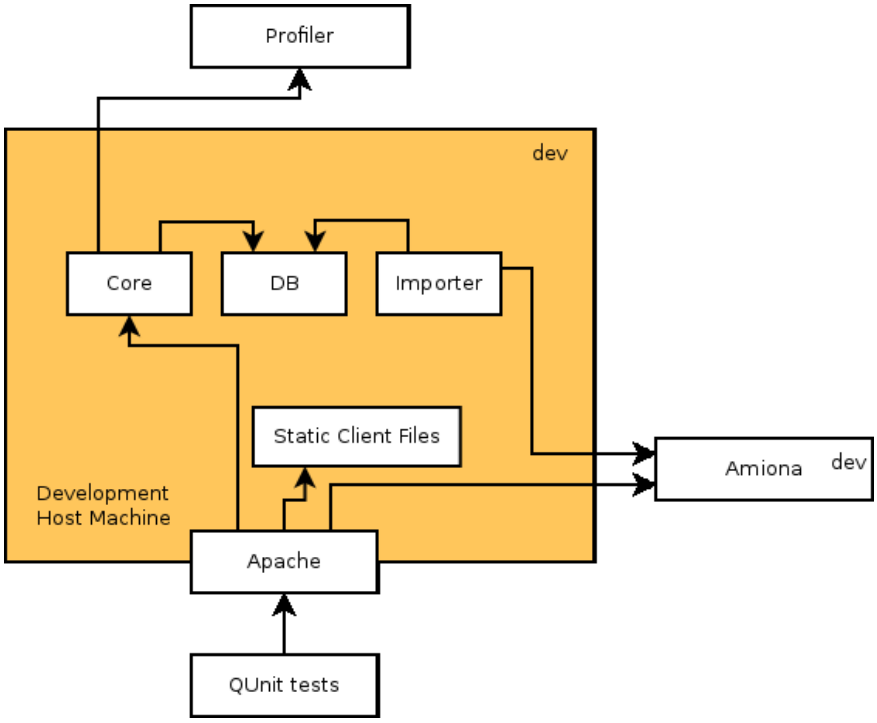

Figure 13: Test system setup

4.1.2 Integration test suite

The test suite is based on QUnit, a JavaScript framework for unit testing. All REST API endpoints are grouped functional identifiers (see Figure 20). Each test case consists of multiple requests and assertions that are based on the defined interface for each endpoint. Note that the measured time in the integration tests is the time it took all calls in each test case to be completed and processed. Also note that the number of requests differs between test cases and the option “Create Entities” is deactivated, hence all requests, which would lead to a new entity in the server's storage, are disabled.

The test suite also allows to activate fault injection. When activated, a server-side error (HTTP 500) is simulated. This is necessary to verify that error messages are correctly returned by the client-server communication library. This feature can also be utilized to test client-side error handling in the graphical user interface.

Figure 20: Screenshot of integration test suite.

Altogether the suite tests how all backend components work together. This also includes the ACS, the Profiler, the core functionality and the Apache reverse proxy configuration. The test suite basically takes the role of another client application and issues its calls directly to the client-server communication library, which is the foundation for all client-server interaction. With each deployment the test suite is executed in the local test setup as well as in the production environment.

4.1.3 Sub system response and waiting time test

Response time is an important factor when it comes to usability. More accurately it is not the response time, but the whole waiting time. Waiting time is the amount of time that passes between the request of an action and its completion. Naturally the waiting time will vary between different setups, largely depending on the Internet connection, current server and client load.

While client-side load is hardly traceable, both the connection to the server and the server load can be set up in a controlled environment in order to get conclusive results. Both Figure 21 and 22 shows the average waiting time for the endpoints listed in Table 8 and 9. Ideally all results would be below 100 ms. This is true for most of the platform's endpoints. Some of the more complex ones take up to 200 ms, which is still in reasonable boundaries. These usually combine the functionality that would else wise be provided by two or more endpoints. Two items are obviously well above the desired waiting time. [25] provides the functionality to suggest an Offer or Demand category based on a user-defined title (takes about 578.27 ms). It is not a vital feature for the platform's users, but it still offers support, because it may speed up the interaction process. Nevertheless there is room for improvement. [30] takes about 364.47 ms and allows the user to request an Offer. The action is not performed too often, therefore the timing should be tolerable.

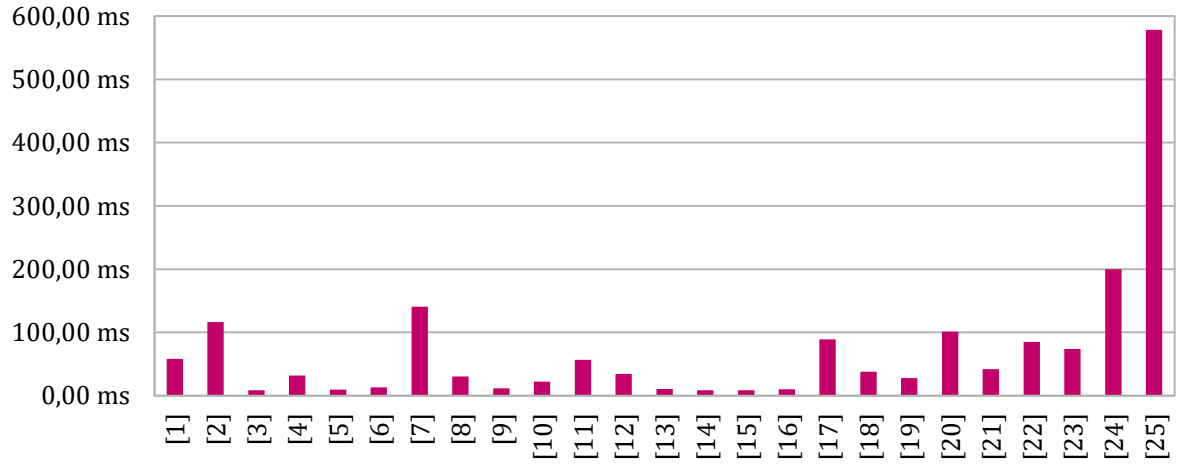

Figure 21: Waiting time test results (1).

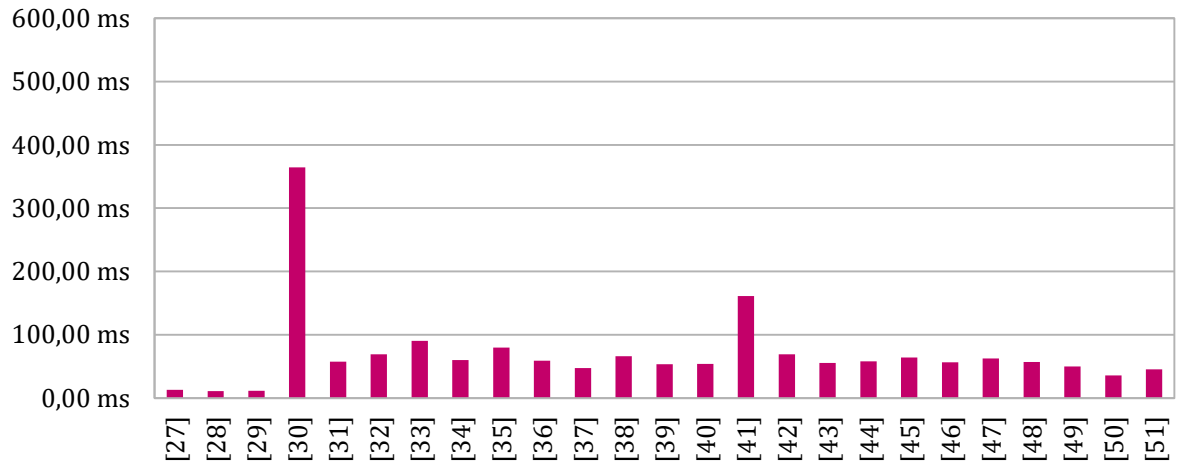

Figure 22: Waiting time test results (2).

Finally, this data can be used to calculate the average response time for each sub system (see Table 7). The results are as expected. The Amiona system has a higher average due to the geographical distance to the client application. The Profiler also has a high distance and a higher time of calculation, which is consumed by graph-based algorithms.

Sub system	Average waiting time
Core functionality ([1]-[23], [26]-[29])	40.69 ms
Amiona ([30]-[51])	77.99 ms
Profiler ([24]-[25])	388.99 ms

Table 7: Average waiting time of sub systems.

ID	Package	Time
[1]	https://demo.getvivid.eu/core/auth/auth.v10	58.25 ms
[2]	https://demo.getvivid.eu/core/auth/token.v10	116.63 ms
[3]	https://demo.getvivid.eu/core/badges/confirm.v10	8.51 ms
[4]	https://demo.getvivid.eu/core/badges/definitions/list.v10	32.17 ms
[5]	https://demo.getvivid.eu/core/badges/list.v10	9.81 ms
[6]	https://demo.getvivid.eu/core/badges/update.v10	13.13 ms
[7]	https://demo.getvivid.eu/core/calendar/events/create.v10	140.89 ms
[8]	https://demo.getvivid.eu/core/calendar/events/delete.v10	30.54 ms
[9]	https://demo.getvivid.eu/core/calendar/events/get.v10/PRIVATE-14	11.47 ms

Table 8: Waiting time test endpoints (1).

ID	Package	Time
[10]	https://demo.getvivid.eu/core/calendar/events/list.v10	22.32 ms
[11]	https://demo.getvivid.eu/core/calendar/events/update.v10	56.72 ms
[12]	https://demo.getvivid.eu/core/calendar/reminders/confirm.v10	34.34 ms
[13]	https://demo.getvivid.eu/core/calendar/reminders/list.v10	10.75 ms
[14]	https://demo.getvivid.eu/core/chat/entity/list.v10	8.70 ms
[15]	https://demo.getvivid.eu/core/chat/entity/markRead.v10	8.80 ms
[16]	https://demo.getvivid.eu/core/chat/latestConversations.v10	10.36 ms
[17]	https://demo.getvivid.eu/core/chat/listen.v10	88.91 ms
[18]	https://demo.getvivid.eu/core/chat/markRead.v10	37.82 ms
[19]	https://demo.getvivid.eu/core/chat/message/list.v11	28.12 ms
[20]	https://demo.getvivid.eu/core/chat/send.v10	101.38 ms
[21]	https://demo.getvivid.eu/core/log/authError.v10	42.00 ms
[22]	https://demo.getvivid.eu/core/log/create.v10	85.14 ms
[23]	https://demo.getvivid.eu/core/profiler/fetchHelpExchange.v10	73.75 ms
[24]	https://demo.getvivid.eu/core/profiler/searchHelpExchange.v15	199.71 ms
[25]	https://demo.getvivid.eu/core/profiler/suggestCategory.v11	578.27 ms
[26]	https://demo.getvivid.eu/core/profiler/suggestTime.v11	32.29 ms
[27]	https://demo.getvivid.eu/core/users/details.v11	13.16 ms

[28]	https://demo.getvivid.eu/core/users/search.v10	11.14 ms
[29]	https://demo.getvivid.eu/static/redirect.html	11.50 ms
[30]	https://demo.getvivid.eu/usg/1/appointment/request	364.47 ms
[31]	https://demo.getvivid.eu/usg/1/bookable/	57.24 ms
[32]	https://demo.getvivid.eu/usg/1/bookable/144	69.21 ms
[33]	https://demo.getvivid.eu/usg/1/create_new_service	90.56 ms
[34]	https://demo.getvivid.eu/usg/1/demand/	59.80 ms
[35]	https://demo.getvivid.eu/usg/1/demand/67	79.80 ms
[36]	https://demo.getvivid.eu/usg/1/demand/edit/	59.16 ms
[37]	https://demo.getvivid.eu/usg/1/demand/is_active/	47.38 ms
[38]	https://demo.getvivid.eu/usg/1/edit_service	65.81 ms
[39]	https://demo.getvivid.eu/usg/1/frontend_profiling_trigger	53.67 ms
[40]	https://demo.getvivid.eu/usg/1/get_service_bookable_dates/431	53.76 ms
[41]	https://demo.getvivid.eu/usg/1/service_bookable_date/	161.15 ms
[42]	https://demo.getvivid.eu/usg/1/service_bookable_date/90	69.10 ms
[43]	https://demo.getvivid.eu/usg/1/service/is_active/	55.60 ms
[44]	https://demo.getvivid.eu/usg/1/user_group/create	57.76 ms
[45]	https://demo.getvivid.eu/usg/1/user_group/get_my	64.27 ms
[46]	https://demo.getvivid.eu/usg/1/user_group/get/167	56.69 ms
[47]	https://demo.getvivid.eu/usg/1/user/session_data	62.35 ms
[48]	https://demo.getvivid.eu/usg/user_group/167/edit	57.12 ms
[49]	https://demo.getvivid.eu/usg/user_group/167/member/delete	49.70 ms
[50]	https://demo.getvivid.eu/usg/user_group/167/member/invite	35.96 ms
[51]	https://demo.getvivid.eu/usg/user_group/delete	45.18 ms

Table 9: Waiting time test endpoints (2).

4.2 Manual system tests

4.2.1 Testing approach

The final test stage is manual system testing. Manual testing is necessary, because (semi-)automated tests can only check assumptions made while writing the tests. Such assumptions usually only cover low-level functionality and lack a high-level understanding of the overall system. Especially the semantic content, the meaning of displayed messages is usually related to a complex state created by the interaction between user and system. Thus it is necessary to perform manual tests.

The manual testing follows the functional requirements as they were agreed up on during sprint planning. Both Mobile and HbbTV client are used to verify that the given requirement is implemented and provides the agreed functionality. The importance of manual testing will continue to grow as the project continues. Interactions between multiple clients and profiling features create a level of complexity that is hard to test with simple assertions. While most requirements can be tested by the use of the UI only, it is sometimes necessary to verify the time and content of certain requests to the server-side API. For this purpose most modern browsers (e.g.

Chrome or Firefox) provide developer tools that allow to monitor network traffic. These tools also provide means to perform an error diagnosis in case of failure.

Finally note that the manual system tests are executed in the demo environment of the project, which includes real end-user hardware.

4.2.2 Intermediate results

The results below are based on the development status by the mid of December (Sprint 2014-11). Furthermore, the status by the end of the previous sprints is also visualized. The list of features is grouped by the corresponding user stories (see Table 10). The columns mark the past sprints/deployed versions. Yellow means, that a backend is available. Green means, that one of the clients implements the UI for the corresponding backend. Anyways, a colour highlighted block next to a feature means, that it passed the manual system test (as well as the Unit Tests described in section 3 above). See Table 6 for a detailed description of the colour highlighting.

System Components	
USG	The feature is implemented by the ACS system (backend and web client).
PLUS	The feature is implemented by PLUS' backend.
ISOIN	The feature is implemented by the profiling component.
IRT	The feature is implemented by the HbbTV client
EVISION	The feature is implemented by the mobile client.

Table 10: List of system components

Feature/User Story	2014-07	2014-08	2014-09	2014-10	2014-11
03 - Login and Logout					
Use credentials to create a session (OAuth2.0). (HbbTV client uses hard coded credentials.)					
Use session to access resources. (Other system components are unable to validate token.)	USG only	USG only	USG only	USG only	
Show basic session information.					
05 - Offer-driven approach					
Create a Professional General Offer (Web)					
Respond to a Professional General Offer (by id) / Request an Appointment					
Get Appointment details by id					

Show available categories
Get list of offers by category (HbbTV client fetches offers, but no UI is available.)
Get Offer details (by id)
Accept / Cancel an Appointment by id
Search for Offers (free-text)
Get first name, last name and avatar of users by id
08 - Message Service
Messaging: Set status to online (on log-in) (The message view uses a hard coded user at the moment.)
Messaging: Send a message to a single user (by id)
Messaging: Send a message to a group in the contact list
Messaging: Show a message from another user (Currently only in the message view.)
Contact Management: Show groups and contact list
Contact Management: Add a user to contact list by JID
Contact Management: Create a group in the contact list
Contact Management: Add a user in the contact list to a group
Contact Management: Remove a user from a group in the contact list
Contact Management: Remove a user from the contact list
Contact Management: Search users by first, last or user name.
Contact Management: Retrieve user details by id list.
11 - User Profiling
Store a profiling event
Wizard: Get category suggestion based on title.
Wizard: Get time suggestion based on category id.
Wizard: Get location suggestion based on category id.

Table 11: List of implemented features

It might seem odd, that some functionality that was available in Sprint 2014-10 is marked unavailable in Sprint 2014-11. This is due to major changes in the UI design, which were introduced in the same sprint. The integration of all existing features was not finished by the end of the sprint and is therefore marked yellow. The availability of the backend functionality was not affected, though.

4.2.3 Final results

The final results (sprint 2016-02) are discussed separately from the intermediate results to give a better overview. Also the items have been reorganized in order to give a better idea of the GeTVivid platform's features. See Table 12-15 for the final manual test report. A green cell marks a passed test.

Functionality	HbbTV	Mobile
Log-in/-out		
Log-in with a valid account		
Quick switch do another account		
Log-out of session		
Show help information		
Start page		
List recent Offers		
List recent Demands		
Show help information		
Search/browse		
Search/browse Offers and Demands		
Show help information		
Offer wizard		
Match-making with Demands		
Page 1: enter title, description, category		
Page 2: enter date, time, location		
Page 3: enter group, distance restriction		
View summary and save Offer		
Show help information		

Offer wizard		
Match-making with Offers		
Page 1: enter title, description, category		
Page 2: enter date, time, location		
Page 3: enter group, professional/informal		
View summary and save Demand		
Show help information		

Table 12Manual test results (1).

Functionality	HbbTV	Mobile
My profile		
Display my details		
Display my most used categories		
Display my earned badges		
List my Offers and Demands		
Change one of my Offers/Demands		
Hide one of my Offers/Demands		
Recreate one of my Offers/Demands		
Show my Offer/Demand in organizer		
Show help information		
My settings		
Couple with tablet / HbbTV application		
Change keyboard layout		
Request a change of my personal details		
Modify display settings of badges		
Receive notification about earned badges		
Show help information		

Other's profile	
Display other's details	
Display other's most used categories	
Display my earned badges	
List other's Offers and Demands	
Add other user to group	
Send other user a message	
Show help information	

Table 13: Manual test results (2).

Functionality	HbbTV	Mobile
Calendar		
List my appointments, Offers/Demands, private events		
Display my appointment, Offer/Demand, private event details		
Create/edit/remove a private event		
Create/edit/remove a reminder for an appointment		
Receive a reminder notification		
Confirm a reminder notification		
Show help information		
Messaging		
Display latest conversations		
Display conversation history		
Start a new conversation with one or many users		
Send the community manager a message		
Send a message		
Receive a message/a new message notification		
Show help information		

Organizer	
List appointments, reminders, Offers/Demands received from community manager, my Offers/Demands	
Display details for Offers/Demands received	
Request Offer/Demand with or without date	
Display details for my Offers/Demands	
Display/confirm a reminder	
Display/reschedule/accept/decline/comment an appointment	
Receive a notification about a new appointment coordination step	
Show help information	

Table 14: Manual test results (3).

Functionality	HbbTV	Mobile
Group management		
List my groups		
Create/edit/delete a group		
Add/remove users to/from a group		
Show help information		
Tutorial videos		
Browse/play tutorial videos		
Show help information		
On-screen overlay notifications		
Display notifications in an on-screen overlay (HbbTV) / Display notification as native Android notifications (mobile)		

Table 15: Manual test results (4).

5. OVERALL CONCLUSION

The deliverable gave an overview over the test techniques used during the development of the GeTVivid platform. Testing results were presented to show the quality of the developed software. The testing and integration can be summed up as follows:

Integration has taken more effort than expected due to the different nature of used sub-systems and the initial individual understanding of each partner of the platform. One successfully method to cut down this effort used during the project was to reduce the length of the development and integration cycle. One month turned out to be reasonable time span for five technical partners working together.

Most problematic were features that were not properly specified, i.e., first implementations had often to be reworked completely or features were implemented and used only to discover later that something different was needed. This circumstance got better during the projects' progression due to the insight of every partner that a thorough specification was needed for features. This also resulted in several constructive discussions, in which all partners laid out their ideas and they were discussed until all reached an agreement in a form of a clear specification. Features were written down in tasks for each partner and checked bi-weekly. This made development for all partners much easier and resulted in less reworks. For future projects, a thorough specification with tight checking of fulfilled tasks and demanding of them if need, should be adopted as a good and goal-oriented practice from the start of the project. It also turned out, that accurate, up-to-date technical documentation was the most crucial and helpful tool when partners with different technical background are collaborating. Therefore, a lot of time and effort was put into creating documentation.

One way to reduce invested time and increase quality was to use existing unit tests to automatically create and compile examples and descriptions for all tested REST endpoints. The testing of most components started before the actual development like in any other test-driven development environment. The mixture of automated unit tests and manual tests enabled the developer to develop the defined functionality without losing the overall view on the GeTVivid platform and without wasting too much time on rudimentary API tests. The user-centred design paradigm, which was an important part of the project philosophy, implied that each feature is controlled and tested by a human, before it is accessible to the end-user. All test steps together made sure that the platform behaves stable in the everyday usage even during peak hours.

The functionality implemented and available in the client applications by the end of sprint 2016-02 (last sprint before the field trials) works as specified. All involved system components are integrated as far as it concerns the defined endpoints and functionality. The platform is stable and operates within expected parameters (e.g., subjective tolerable overall time until response is visible to the user). In the current architecture there is a split within logical subsystems, which are now hosted on different servers. This makes synchronization problematic and results in delays that are bad for the user experience. To improve this user experience greatly the whole platform should be hosted on one physical server. From a technical perspective it would also make sense to merge all sub-systems into one system to reduce complexity and improve maintainability. This not possible at the moment, though, because the sub-systems use different programming languages. This is not a call to merge all possible sub-systems, but in the current architecture the split goes through a point where no splitting should happen (e.g., right through logical or functional units).

REFERENCES

Beck, K. 2003. *Test-Driven Development: By Example*. Addison-Wesley, Boston, MA.

Beck, K. 2004. *JUnit Pocket Guide*. O'Reilly, Sebastopol, CA.

Molyneaux, I. 2009. *The Art of Application Performance Testing* (2nd ed.). O'Reilly, Sebastopol, CA.

Nielsen, J. 1993. *Usability Engineering*. Academic Press, Boston, MA.

Schach, S.R. 2011. *Object-Oriented and Classical Software Engineering* (8th ed.). McGraw-Hill, New York, NY.