Master's Degree in Informatics Engineering
Thesis
Final Report

Web application for interoperability testing of web services

Bruno Miguel Filipe Martins
bmfm@student.dei.uc.pt

Supervisor:
Prof. Nuno Laranjeiro

Date: January 29th, 2016

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Master's Degree in Informatics Engineering
Thesis
Final Report

Web application for interoperability testing of web services

Author:
Bruno Miguel Filipe Martins
bmfm@student.dei.uc.pt

Jury:
Prof. Marco Vieira
Prof. Carlos Bento

Supervisor:
Prof. Nuno Laranjeiro
Date: January 29th, 2016

Acknowledgments

I would like to start by thanking my supervisor, Professor Carlos Nuno Laranjeiro. Without his immeasurable help and guidance this thesis would have been a shadow of itself. To this day I still cannot comprehend how can someone not be completely fed up with me after three meetings a day. His resilience is admirable.

To Ivano Elia, for putting up with my endless questions about his previous work. His input throughout the thesis, especially in the testing phase, was invaluable.

To Paulo Martins. After a day's work, after taking care of his own company and after looking for his own family, still having the time to provide me with some tips is nothing short of incredible.

To Catarina Lopes, because a long-distance relationship is hard enough without one of the members always complaining about his day. For always having the patience to hear me out even when going through some tough times herself. For understanding that this thesis took a lot more of my time than I expected.

And last but definitely not least, to my parents. For believing me until the end. For never letting me down. For being the best parents someone could ever ask for. For always being a safe haven, no matter how grim things were. This long journey wasn't easy. A lot of ups and downs. But here we are. None of this would have been possible without them. Words cannot express how crucial they were. Nonetheless,

Obrigado Irene.

Obrigado Tomé.

You have my deepest and sincere gratitude.

Abstract

The web services technology has been created to support communication between heterogeneous platforms. Despite its maturity, built upon more than a decade of experience, research and practice show that the technology still fails to connect the two sides of an interaction, even when the platforms involved are the same. This is especially concerning for providers, as a failure in the inter-operation of web services can bring in disastrous consequences for the services involved, which frequently support businesses. In this paper we present WitWS, an on-line web application designed to test the interoperability of a web service against specific client-side platforms. The tool is able to test not only the pre-runtime steps involving code generation, but also the end-to-end runtime communication present in a web service interaction with a client. WitWS has been used to test a set of web services deployed on Glassifsh and WildFly against the well-known Metro JAX-WS, JBossWS, and Axis2 client platforms and was able to disclose severe interoperability issues.

Keywords

"Web Services"; Interoperability; Certification; Testing.

Table of Contents

Figure's List

Table List

Acronyms

Acronym	*Meaning*
AJAX	**A**synchronous **J**avaScript **A**nd **X**ML
CISUC	**C**entre for **I**nformatics and **S**ystems of the **U**niversity of **Coimbra**
CSS	**C**ascading **S**tyle **S**heets
DB	**D**ata**b**ase
FTP	**F**ile **T**ransfer **P**rotocol
FURPS	**F**unctionality, **U**sability, **R**eliability, **P**erformance, **S**upportability
GUI	**G**raphical **U**ser **I**nterface
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
IDE	**I**ntegrated **D**evelopment **E**nvironment
INTENSE	**In**teroperability **Te**sti**n**g **S**ervic**e**
MoSCoW	**M**ust-have, **o**, **S**hould-have, **C**ould-have, **o** and **W**on't-have.
MVC	**M**odel **V**iew **C**ontroller
PODAM	**Po**jo **Da**ta **M**ocker
REST	**Re**presentational **S**tate **T**ransfer
RMI	**R**emote **M**ethod **I**nvocation
SMART	**S**pecific, **M**easurable, **A**ttainable, **R**ealistic and **T**ime-bound.
SMTP	**S**imple **M**ail **T**ransfer **P**rotocol
URL	**U**niform **R**esource **L**ocator
UTF	**U**nicode **T**ransformation **F**ormat
WAR	**W**eb application **Ar**chive
WSDL	**W**eb **S**ervices **D**escription **L**anguage
XML	E**x**tensible **M**arkup **L**anguage
XMPP	E**x**tensible **M**essaging and **P**resence **P**rotocol

Chapter 1
Introduction

1.1. Scope

This thesis was developed within the discipline of Thesis/Internship inserted into the Master in Informatics Engineering of the Department of Informatics Engineering – which is part of the Faculty of Sciences and Technology, University of Coimbra. This project started on February 11th, 2015 and ended on January 15th, 2016.

The thesis, in its entirety, was developed in a laboratory located in CISUC's facilities, within the Software and Systems Engineering group, with the close guidance of Professor Nuno Laranjeiro, PhD.

The project team consisted solely by the author and the invaluable feedback and advice of Professor Nuno.

1.2. Motivation

In a web services environment, a provider publishes its service interface in a WSDL document, which describes the available operations, including required input and output parameters. A client can use the published service description to understand which operations are available and how they can be invoked [1] In both cases, i.e., to publish a service interface or to invoke a remote operation, developers do not need to implement all supporting code. In fact, most web service frameworks (software platforms that allow the creation, deployment, and invocation of web services, e.g. Axis2, Wsimport) provide the necessary support for service description generation, client-side code generation, and also runtime support to allow the easy invocation of remote operations by taking care of all communication requirements, in particular, message serialization and deserialization [2].

Although web services have been designed to support interoperable application-to-application communication, experience shows that this is a quite difficult goal to achieve, raising concerns when implementing services with reliability requirements. For example, using a web service framework to generate C client code from a web service written in Java might pose some problems, as some data types do not exist in both languages. If the tool is able to generate the code, more problems might arise when trying to compile it. Even by having the same programming language on both ends, the process does not always end well, as shown later on.

The interoperability of web services is, in fact, an issue in which the Web Services Interoperability organization (WS-I) [3] has been working for several years now, with limited success so far. Thus, practice shows that developers are deploying WS-I compliant services, which however are many times unable to inter-operate with client applications, even when the client applications are also supported by WS-I compliant frameworks.

Several testing tools have been created to test web services, but they are, in general, quite limited in what concerns interoperability testing. Among all tools, the ones provided by the WS-I Testing Tools Working Group [4] are widely accepted by the industry. Even so, services that pass the WS-I tests are still failing to achieve interoperability, which ultimately shows the limitations of the assessments produced by the WS-I tools.

The interoperability problem was also analyzed by the authors in [5], where they tested a large number of web services frameworks. However, these tests were carried out in a manual way at some points. They state the difficulty of difficulty of testing web services for interoperability in an automatic-way because of its technical challenges.

1.3. Objectives

Based on the issues explained in the previous section, the objectives of this thesis are as follows:

- Build a web-based tool that implements the interoperability testing procedure discussed in [5];
- Automate the interoperability tests, not only for pre-runtime steps (e.g. client-side code generation) but also for the runtime steps (e.g. communication between client and server);
- Modularity: provide the means to easily extend the core functionality (e.g. add more test phases);
- Validate the tool against a subset of services used by the authors in [5]–[7], obtaining 100% of correctness in the results.

This thesis is partially based on the following work submitted to an international tier A [8] conference: INTENSE - INteroperability TEstiNg ServicE [9].

1.4. Report structure

The second section provides context about the work field and analyzes other tools and approaches similar to what we're trying to accomplish. The third clarifies the necessary requirements the application must follow and implement, followed by the fourth section, "Methodology and planning", which explains the software development cycle, along with the task plan and risk management.

After explaining all the background, process and necessary features, a more technical discussion can be had, starting with the fifth section, "Support technologies", which analyzes the several tool and frameworks available in order to build the application. After that analysis, the sixth section, "Architecture and operation", details the architecture of the system.

The seventh, "Implementation details", goes into detail about several aspects of the application implementation and the eighth section presents the experimental evaluation designed to verify the application's functionality.

The ninth section contains formation about the current limitations of the developed system, along with instructions on how to extend its functionality.

The final user also has a dedicated chapter with the tenth section, "User manual". It showcases and explains every view and component of the web user interface.

Finally, the last section, "Conclusions", talks about the tasks completed throughout the year, indicating the experience acquired and the aspects to work on in the future.
The "Annexes" section at the end include class diagrams, sequence diagrams, mockups which are not introduced in the main body of the thesis, the developer manual (which does through all the necessary steps to run and deploy the application) and the aforementioned research paper.

Chapter 2 Background and Related Work

So what is, exactly, a web service? There are many different definitions throughout the World Wide Web and in books. For simplicity and objectivity sake, since it is commonly accepted, we are sticking with the one provided by [10]:

"A Web service is any piece of software that makes itself available over the Internet and uses a standardized XML messaging system"

Given the incredible amount of programming languages and web services middleware available, an extremely important question arises: how can they all communicate between one another?

In a typical web services environment, the provider (e.g. the server) offers a well-defined interface to consumers (e.g. clients), which includes a set of operations and typed input/output parameters. Clients and servers interact by using a **web service framework** that provides a set of mechanisms to support the execution of the following steps: first the web service is deployed at the server, along with a service interface description (e.g. a WSDL is published); second, a client developer generates (and compiles, when required) client side artifacts to easily invoke the service operations; and finally both client and server applications communicate by exchanging SOAP messages that are generated by the underlying frameworks, on behalf of the client and server applications.

Before moving on to the next chapter, a brief introduction is needed on the aforementioned basic concepts.

2.1. SOAP

The SOAP acronym originally meant Simple Object Access Protocol, but later it was dropped and now SOAP is an official name by itself. It is a protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format, and relies on other application layer protocols for message negotiation and transmission, such as the popular Hypertext Transfer Protocol (HTTP), but also Simple Mail Transfer Protocol (SMTP), or XMPP, just to name a few. The framework has been designed to be independent of any particular programming model and other implementation specific semantics [11]. Figure 1 and Figure 2 represent a simple example of a SOAP request and a SOAP reply, respectively.

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Figure 1. Example of a SOAP request

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

Figure 2. Example of a SOAP reply

2.2. WSDL

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate. The next figure represents an example of a WSDL file structure.

```
<!-- WSDL definition structure -->
<definitions
name="MathService"
targetNamespace="http://example.org/math/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
>
  <!-- abstract definitions -->
   <types> ...
   <message> ...
  <portType> ...

  <!-- concrete definitions -->
  <binding> ...
  <service> ...
</definition>
```

Figure 3. WSDL Structure Example

2.3. The five steps of the inter-operation process

After explaining some of the basic concepts composing the web services technology, we can now explore, in more detail the inter-operation between web services clients and providers. Figure 4 represents a typical web services inter-operation scenario using two different **frameworks** at the client (on the left-hand side of Figure 4) and server sides (at the right-hand side of the figure), which are placed at each endpoint of the interaction.

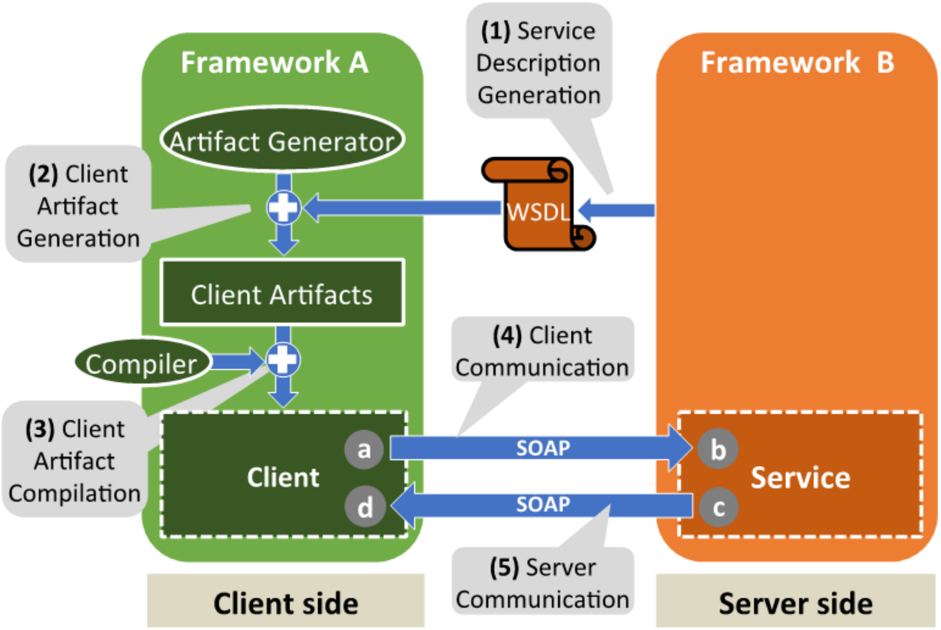


Figure 4. Typical web services environment

In the scenario represented, the client uses the client-side subsystem of framework A, while the server uses the server-side subsystem of framework B (although these could also be two subsystems of the same framework). As shown, five key steps have to be performed for a client to invoke a remote service operation. These steps represent the critical points where platform-level interoperability issues may arise.

2.3.1. Service Description Generation – Step 1

This step typically occurs automatically when the service is deployed in the server. Two approaches can be used to reach this step: WSDL-first and Code-first, also called Contract-first and Contract-last, respectively.

In the WSDL-first web service, the WSDL file is created first, without actually writing any service code. Clients are decoupled from any logic on the server. The logic can be revised on the server without affecting the clients. More granular control exists over what is in the request and response messages and WSDL structure.

In the Code-first web-service, existing logic is "exposed" as a web service and the WSDL is created at the very end. Developers do not need to learn SOAP or any XML-related technologies, and services are created quickly by "exposing" internal APIs with automated tools. [12].

In both cases the result of this step is a WSDL document that provides the information needed by a client that uses framework A to invoke the service provided by framework B.

2.3.2. Client Artifact Generation – Step 2

The client's artifacts are produced using the artifact generation tool provided by Framework A (refer to Figure 4). These artifacts are pieces of code that, at runtime, will translate application-level calls at the client to SOAP messages that will be delivered to the server. Some frameworks do not produce source files; instead they generate these artifacts dynamically, at runtime.

2.3.3. Client Artifact Compilation – Step 3

The Client Artifact Compilation step is necessary only for platforms that are implemented in programming languages that require compilation of the code before execution (e.g., Java, C#), and in some languages/frameworks (e.g., Python) it may not be required (the client-side artifacts are dynamically generated at runtime). Obviously, the artifacts (that after this step are in a compiled form) only provide the methods for a client application to invoke the remote service, but do not actually invoke it. It is up to the developer to create the client code to invoke the artifact methods that will allow communicating with the server.

2.3.4. Client communication – Step 4

This step starts precisely when the client code calls the remote operation using, as arguments, the object (or objects) required by the remote operation. This delegates the control to the framework that serializes the outgoing object (represented by *a* in Figure 4) to SOAP and sends the message. The step ends when the application-level object is delivered to the service application (i.e., after being deserialized from SOAP to the exact object that the server-side platform expects), which is represented by *b* in Figure 4 .The object is then processed at the server (i.e., business logic is executed).

2.3.5. Service Processing – Step 5

Finally, the **Server Communication** step begins immediately before the server sends a response object back to the client (represented by *c* in Figure 4). As in the previous step, control is delegated to the framework that translates the response object to SOAP. This step ends when the response object is delivered to the client application, after being translated from SOAP by the client-side framework – represented by d in Figure 4.

2.4. The Interoperability Problem

Experience and research clearly show that despite the effort put in bringing in interoperability between heterogeneous systems it is still very frequent to find cases where the different systems (i.e., client and server) are unable to inter-operate [5]–[7]. Thus, previous research reports cases where message contents are lost between client and server; support code cannot be generated or sometimes cannot be compiled in a specific platform, among many others [5]–[7]. These cases are due to many factors, ranging from poor web service platforms implementations, to issues that can be traced back to the specification documents, which are written in natural language.

Recognizing the interoperability problem, the WS-I organization [3] has been working, for over a decade, on creating standards to promote interoperability, and this includes refining and restricting the web services specifications and producing tools that can be used to test web services [4], [13]. The problem is that, despite the effort, still a WS-I compliant service shows problems to inter-operate with specific client platforms. Also, testing tools nowadays are usually limited to execute the WS-I recommended tests, which are also quite limited and are unable to detect severe inter-operation issues.

2.5. Approaches and Tools for Interoperability Assessment

Authors in [5]–[7] studied the inter-operation problem by testing a large set of web services frameworks in which we were able to disclose numerous issues. Despite of the usefulness of the results, the authors carried out the tests in a semi-automatic way (at some points with manual intervention). The authors indicate the difficulty of building a tool that could be used by developers to test services for interoperability in a completely automatic way. In fact, this holds huge difficulties, which include having to generate code for multiple platforms, having to generate a test workload that fits any service, being ready to test a new platform with minimum effort, but especially producing client code that is able to communicate with any service, on any platform.

Since our focus is inter-operation, the WS-I Basic Profile and respective tools are closely related to our work [14], [15] . As previously referred, although the WS-I effort represents a first step, experience shows that even WS-I compliant services may have interoperability problems, and obviously a more practical and realistic insight of a given platform's interoperability is many times preferred to conceptual results. This is especially true in systems where interoperability is critical.

When analyzing tools and frameworks, there are only a couple that perform compliance testing. Compliance testing, also known as conformance testing, is a type of non-functional software testing. It's a methodology used in engineering to ensure that a product, process, computer program or system meets a defined set of standards [16], the WS-I standards in our case. Since the focus of this thesis is on interoperability between web services, it is the type of testing that best fits in this thesis. To have a wider set of results, we have extended our analysis to tools that do not actually perform this type of testing but also test web services in a more generic way.

2.5.1. SoapUI

SoapUI [17] is an open-source web service testing application for service-oriented architectures (SOA) and representational state transfers (REST). Its functionality covers web service inspection, invoking, development, simulation and mocking, plus functional, load and compliance testing. The Pro (paid) version also includes WSDL coverage and refactoring, API discovery, XML inspector, among others.

SoapUI's GUI makes it easy to test all of its supported technologies. The tool received a number of awards, including [18], [19] and [20].

SoapUI supports compliance testing by including the execution of the WS-I Basic Profile 1.1 test suites [21], for which we already highlighted the limitations. SoapUI also supports the generation of client side artifacts with a limited set of frameworks. However, developers do not have a way of adding a new framework (or in some cases, not even a new version of an existing one) without changing SoapUI's code, which obviously is not practical.

2.5.2. SOAPSonar

SOAPSonar [22] is a software testing and diagnostics tool for SOAP, XML and REST based Web Services. The core focus is on functional, performance, interoperability, and security testing of service endpoints by performing client simulation and automated generation of client messages.

In what concerns interoperability, the tool (in its paid versions) is able to check for WS-I Basic Profile 1.1 compliance. It also supports WSDL Region WS-I Assertion Violation Highlighting and Dynamic XSD-Mutation Active WS-I BP 1.1 compliance. SOAPSonar's website claims: "Adhering to WS-I Basic Profile 1.1 ensures that your components are interoperable and that your WSDL can be consumed by any .NET or Java client".

As of time of writing, it was not possible to test the application since the installer fails to conclude its execution on Windows – and it is the only operating system supported. This emphasizes the need of a platform-agnostic web tool without any kind of installation.

2.5.3. WebInject

WebInject [23] is a free tool for automated testing of web applications and web services. It can be used to test individual system components that have HTTP interfaces (JSP, ASP, CGI, PHP, AJAX, Servlets, HTML Forms, XML/SOAP Web Services, REST, etc), and can be used as a test harness to create a suite of (HTTP level) automated functional, acceptance, and regression tests.

HTTP response times can be collected and monitored in real-time during test execution. Timer statistics are calculated and displayed in a monitor window during runtime. This is used to verify responses from the web application or web service under test are within an acceptable range (to meet a SLA or quality of service criteria). This also enables WebInject to be run as a performance probe for application/service monitoring.

Regarding interoperability, WebInject does not support any kind of compliance testing and does not check any set of best practices (e.g WS-I). With the right modifications, this tool could start checking for interoperability but it would be too time-consuming and it is not suited for our needs.

2.5.4. JMeter

JMeter [24] is an open-source testing tool developed by Apache Software Foundation (ASF). It is distributed under Apache License. It was originally designed to test Web applications but has been extended to other test functions. The core function of JMeter is to load test client/server application but it can also be used for performance measurement. Furthermore, JMeter is also helpful in regression testing by facilitating in the creation of test scripts with assertions. Most of its architecture is based on plugins, so developers can easily extend its functionalities. However, when it comes to interoperability, the tool is limited to custom tests the user might create and even so, it would only partially cover the process we are aiming for.

2.5.5. Storm

Storm [25] is a free and open-source tool for testing web services. Storm is developed in F# language and is available free to use, distributed under New BSD license. Storm allows testing web services written using any technology. Storm supports dynamic invocation of web service methods even those that have input parameters of complex data types and facilitates editing/manipulation of raw SOAP requests. Multiple web services can be tested simultaneously which saves time, speeding up the testing schedule. Compliance testing is not readily available – the user would have to create time-consuming custom tests to even remotely achieve it.

2.5.6. WizDL

WizDL [26] is a utility that allows you to quickly import and test web services. It supports calling complex web services that take arrays and deeply nested objects as parameters. The tool allows saving the data as an Extensible Markup Language (XML) file, which can be loaded later for regression testing. Just like the majority of the tools in this section, no compliance testing of any kind is available out of the box.

2.5.7. Coyote

Coyote [27] is an XML-based object-oriented testing framework to test web services that consists of two parts: test master and test engine. The test master allows testers to specify test scenarios and cases as well as performing various analyses such as dependency analysis, completeness and consistency, and converts WSDL specifications into test scenarios. The test engine interacts with the web services under test, and provides tracing information. The test framework incorporates concepts from object-oriented application frameworks so that it is relatively easy to change test scenarios/cases.

Coyote does have an interesting concept behind it, but the software doesn't seem to be publicly available. The authors talk about Coyote [28] from a theoretical point of view and point out its hypothetical drawbacks, but without a real product it is not possible to actually test or comment on it.

2.5.8. Tool summary table

Table 1 represents a summary of the aforementioned tools. The "Development Activity" column represents the state of the tool development as of January 2016. The column can take two values:

- No development – if there haven't been new software releases in the past year.
- Slow development – if the last version dates from one year back to 6 months ago.
- Active development – if there has been a version released between 6 to 3 months ago.
- Very active development – if the last version dates from less than 3 months ago.

Tool	Technology Support	Compliance Testing Support	Development activity	OS Support	License
SoapUI	HTTP, HTTPS; REST; SOAP; Databases via JDBC; JMS (through HermesJMS); REST; AMF;	WS-I Basic Profile 1.1	Active development	Cross-platform	Standard Version Open Source Pro Version – Paid
SOAPSonar	HTTP, HTTPS, IBM MQ, Tibco EMS, Weblogic JMS, Active MQ, FTP, SFTP	Design Time WS-I Basic Profile 1.1; WSDL Region WS-I Assertion Violation Highligthing; Dynamic XSD-Mutation Active WS-I BP 1.1.	Slow development	Microsoft Windows	Personal Version – Free Pro Version and Server Version - Paid
WizDL	SOAP	No.	No development.	Microsoft Windows	Open Source
WebInject	HTTP,SOAP,REST	No.	No development.	Microsoft Windows; All other platforms after compiling source code.	Open Source
JMeter	Web-HTTP, HTTPS; SOAP; Databases via JDBC; LDAP; JMS; SMTP, POP3 IMAP; Native processes;	No.	Slow development	Cross-platform	Open Source – Apache License 2.0
Storm	SOAP	No.	No development	Microsoft Windows	Open source
Coyote	N/A	N/A	-	N/A	N/A

Table 1. Tool summary table

Chapter 3
Requirements

The requirements where written and analyzed utilizing the **MoSCoW** technique. Since it is almost impossible to deliver a product with every single feature desirable on-time, on-quality and on-budget, this technique helps define the importance of each requirement, forcing the stakeholders to prioritize them. [29]

The **FURPS** and **FURPS+** methods were considered but our focus was on prioritization and well-formed requirements, not so much on their qualification (e.g. reliability or performance requirement). Furthermore, having a relative low number of requirements did not demand such granularity.

The requirements came to be after several meetings with the thesis' supervisor. The mockups presented in section 3.2 also helped to provide a better insight about the necessary features.

3.1. Functional

A function is a defined objective or characteristic action of a system or component and a functional requirement specifies a function that a system or system component must be able to perform.

WITWS-REQ-F-0001	Sign In
The user must be able to log in with a previously created account. Required fields: • Email • Password	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0002	Sign up
The user must be able to register a new account for himself/herself. Required fields: • Username (25 characters max.) • Email • Password (16 characters max.)	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0003	Account confirmation email
After the user signs up, an account confirmation email must be sent to his/her address. The user won't be able to log in unless he/she accesses the link provided in the confirmation email.	
MoSCoW: Could-have	
Status: Not completed	

WITWS-REQ-F-0004	Dashboard
The user could be able to see a dashboard as soon as he logins. Some relevant information to display: • Charts with number of tests failed/passed per Level of analysis • Current number of pending WSDL analysis	
MoSCoW: Could-have	
Status: Not completed	

WITWS-REQ-F-0005	Submit WSDL
The user must have an option to submit a WSDL by providing an URL.	
MoSCoW: Must-have	
Statis: Completed.	

WITWS-REQ-F-0006	Select client code generation tool – Submit WSDL
The user must be able to select a code generation tool when submitting a WSDL.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0007	WSDL Validation
The system must validate the URL before proceeding to any certification level.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0008	Upload web service source files
The user must have an option to upload the web service source files – along with all its dependencies.	
MoSCoW: Must-have	
Status: Completed.	

WITWS-REQ-F-0009	Select client code generation tool - Upload web service source files
The user must be able to select a code generation tool when uploading web service source files.	
MoSCoW: Must-have	
Status: Completed.	

WITWS-REQ-F-00010	Select deployment server - Upload web service source files
The user must be able to select the server where the uploaded files are going to be deployed.	
MoSCoW: Must-have	
Status: Completed.	

WITWS-REQ-F-0011	Deploy web service code
The system must be able to deploy the submitted web service source code.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0012	Service Description Generation
The system must be able to generate a service interface description document.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0013	WS-I Standards Compliance Check
The system must be able to check the WSDL (the one submitted or the one created by deploying the source code) against the WS-I standards.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0014	Client Artifacts Generation
The system must be able to generate client-side code from the service interface description document.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0015	Client Artifacts Compilation
The system must be able to compile generated client-side code.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0016	Client Execution
The system must be able to communicate correctly with the server, i.e., message is sent from the client and delivered correctly at the service application.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0017	Client Execution – Random values
When communicating with the server, the client must call the service with randomly generated values.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0018	Client Execution – Send to comparison system
Before actually calling the service with the randomly generated values, the client must send the values to a comparison system for further inspection.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0019	Service Processing Execution
The system must be able to communicate correctly with the client, i.e., a reply is sent from the service and delivered correctly at the client application.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0020	Service Processing Execution – Random Values
When communicating with the client, the service must send a randomly generated reply.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0021	Service Processing Execution – Send to comparison system
Before actually sending a randomly generated reply, the service must send the values to a comparison system for further inspection.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0022	Comparison system
The system must be able to compare the values received from the client and the service.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0023	Results
The system must provide the user the result of each analysis according to the testing levels explained in **WITWS-REQ-F-0008** through **WITWS-REQ-F-00012.**	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0024	Submit notification
The system must display a notification to the user after the WSDL submitted.	
MoSCoW: Should-have	
Status: Completed	

WITWS-REQ-F-0025	Upload notification
The system must display a notification to the user after the files are uploaded.	
MoSCoW: Should-have	
Status: Completed	

WITWS-REQ-F-0026	WSDL Operation Selection
The user must be able to select the operations of the WSDL he/she wants to see analyzed after uploading the WSDL.	
MoSCoW: Could-have	
Status: Not completed	

WITWS-REQ-F-0027	WSDL Operation Custom Range
The user could be able to set a custom range of values to feed the operations of the WSDL being analyzed.	
MoSCoW: Won't-have	
Status: Not completed	

WITWS-REQ-F-0028	Analysis History
The system must store a history of all test runs and display it to the user.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0029	Results exporter
The user should be able to select which reports he wants to export. The reports should be exported to an Excel file.	
MoSCoW: Should-have	
Status: Not completed	

WITWS-REQ-F-0030	User profile
The user should be able to set some configurations from his/her profile, which will be applied to all WSDL analysis. • WSDL List Maximum Size • Request Timeout	
MoSCoW: Won't-have	
Status: Not completed	

WITWS-REQ-F-0031	Testing Modularity
The entire testing pipeline must be modular enough in order to support additional test steps.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0032	Support for additional frameworks and tools
The system must support the addition of new frameworks and tools into each testing level (e.g. adding a new tool to generate the client code). If applicable, the user must be able to choose them in the WSDL submission screen.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0033	Support for different operating systems
All of the tools and frameworks available might be running in different operating system and machines. The system must have a platform-agnostic way to contact them.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0034	Fault tolerance
When any of the machines or tools pf the system cannot be reached, the user must be informed. At the very least, a message saying to "try again later" must be displayed.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0035	Support for compiled programming languages
The system must be prepared to support other compiled languages besides the main one, Java.	
MoSCoW: Must-have	
Status: Completed	

WITWS-REQ-F-0036	Support for interpreted programming languages
The system could support interpreted languages (e.g. Python, Ruby).	
MoSCoW: Could-have	
Status: Not completed	

3.2. Mockups

Some mockups were initially designed to better understand the application, its requirements and its layout. Figure 5 represents one of the most important mockups designed, the "Detailed results mockup", where the user could see a list of all his test runs (pending, executing and completed).

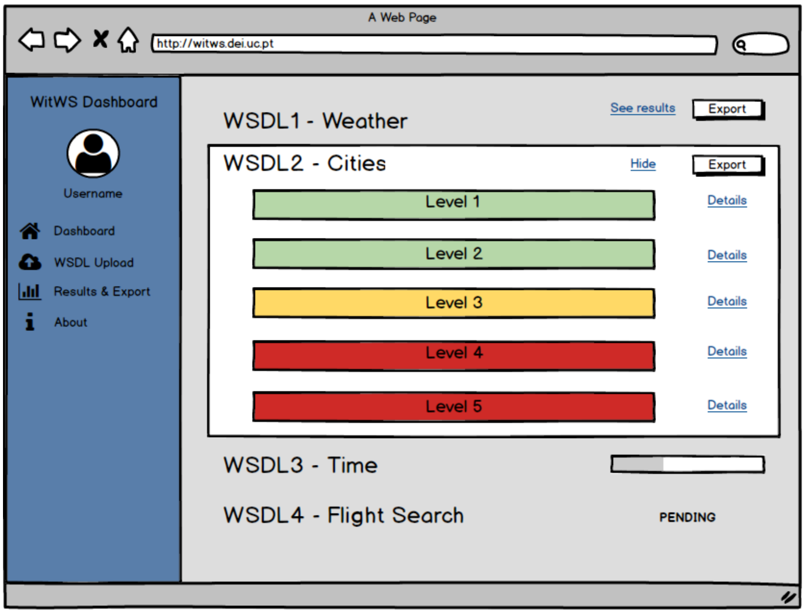
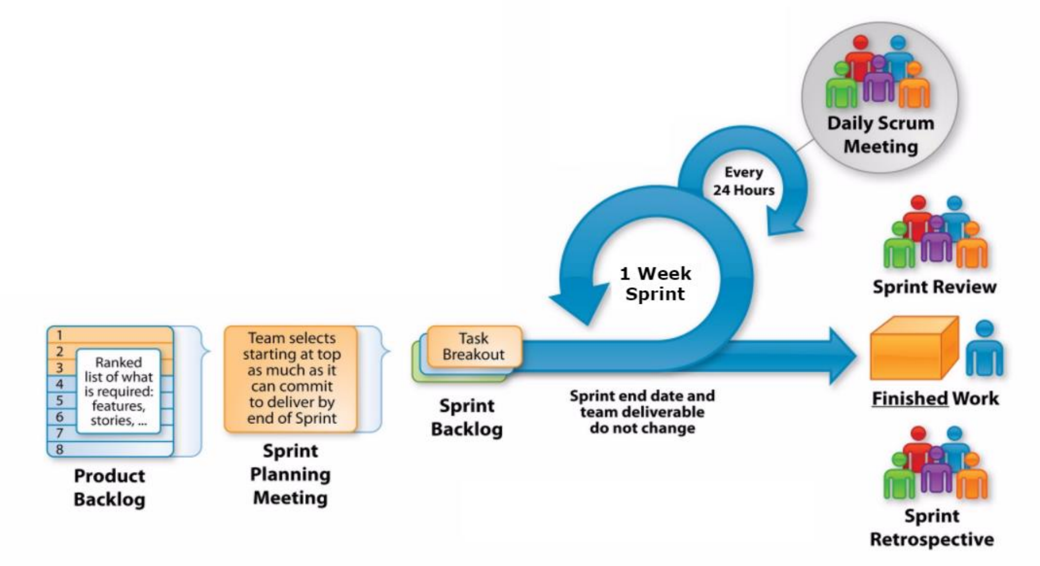


Figure 5. Detailed results mockup

The rest of the mockups originally design can be consulted in the Annexes, section E1.

Chapter 4
Methodology and planning

This thesis followed an agile software development methodology, which means, among other things, that it promoted an evolutionary development cycle and continuous improvement. [30]. SCRUM is one agile software development methodology and is the one that best reassembles the process used in this thesis. [31]. While not being a carbon copy of SCRUM, it's an adaptation of said methodology. The adaptations consist in not having a Product Owner and a Scrum Master because the team only consisted in one element, changing the default Spring duration from two weeks to one week to have a faster development cycle and not having a Spring Retrospective at the end of every Sprint because it was not always necessary. Figure 6 represents the methodology used. Adapted from [32].



Daily Scrum Meeting
Every 24 Hours
1 Week Sprint
Sprint Review
1
2
3
4
5
6
7
8
Ranked list of what is required: features, stories, ...
Product Backlog
Team selects starting at top as much as it can commit to deliver by end of Sprint
Sprint Planning Meeting
Task Breakout
Sprint Backlog
Sprint end date and team deliverable do not change
Finished Work
Sprint Retrospective

Figure 6. Thesis development process

The product backlog consists in a ranked list of what is required. This corresponded to all the system's requirements (which some of them might have been broken down into smaller chunks). At the beginning of each spring, usually on a Monday morning, several tasks from the product backlog were chosen and pledged to be finished at the end of the Sprint. The Spring Backlog is made of those selected tasks. Each Sprint had the duration of one week and, every morning, the author and his supervisor met to talk about the work completed the day before and what is planned to be accomplished until the end of that same day. Any blockers or difficulties were also discussed during this period. At the end of the Sprint, a new, running version of the application, was ready to be delivered, and so, the author and the supervisor met again to discuss it and to review the sprint. – this normally occurred on Friday's afternoons. This process was then repeated every single week until the end of the project.

In some occasions, on Fridays, after the Sprint Review, a Sprint Retrospective was held for the sake of determining what could be changed that might make the next sprint more productive.

Using this approach, the system never deviated from its scope and was much easier to prevent the occurrence of problematic issues.

4.1. Planning

The workflow of the thesis had two distinct parts: an implementation phase where the requirements and the desired functionality were somewhat straightforward, and a research phase, where a part of the process required a lot of research because we wanted to test a service to which we do not have access.

4.1.1. First Semester

During the first semester, the focus was to build the skeleton of the application and to get familiar with the several different frameworks used. In this phase, the first three testing levels were implemented, along with a beta version of the web frontend for the user to submit his/her own WSDL's. The Gantt Chart of the first semester is represented in Figure 7, and a higher resolution version of it can be found at https://goo.gl/0k03yx :

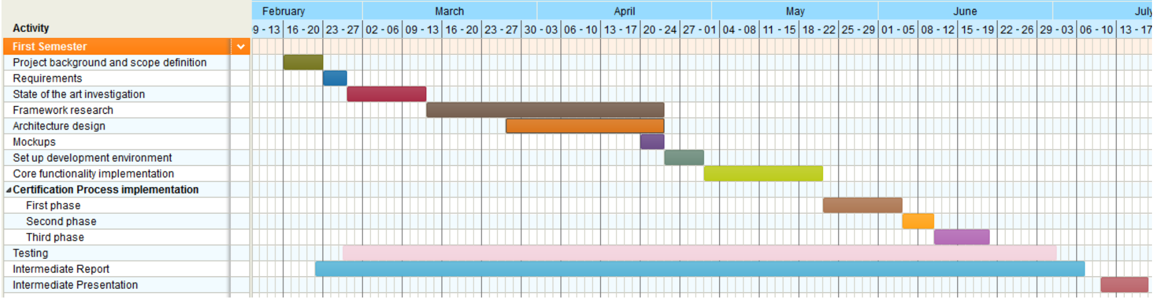


Figure 7. First Semester Gantt chart

- **Requirements:** since the foundations of this thesis were clear and well-defined, the requirements were quick to identify and easy to understand.
- **State of the art investigation:** As previously stated, there aren't many tools or frameworks in this work field so we also had to extend our search to more generic web service testing tools.
- **Framework research:** The overwhelming amount of web frameworks demanded an equally overwhelming amount of time to process.
- **Testing:** It was started at a very early stage, even when we had no code. Static software testing was constantly used in order to check the validity of the documentation.
- **Mockups:** Web frontend wise, this thesis was extremely simple, so there wasn't the need to mock many screens.
- **Core functionality implementation:** This consisted in the implementation of the registration and login process, the modularity of the testing pipeline, the database and its persistence layer, a remote RMI server to process the calls to several tools available (e.g. wsconsume) and a very alpha version of the WSDL submission user-interface.

4.1.2. Second Semester

The planned Gantt chart for the second semester is represented in Figure 8 and a higher resolution of it can be found at https://goo.gl/Z04zf2 .

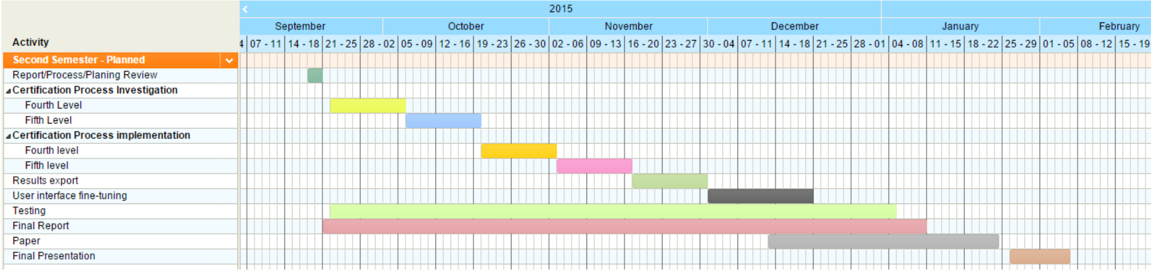


Figure 8. Second Semester Gantt Chart - Planned

The tasks worth of note in this semester are:

- **Report/Process/Planning review:** During the intermediate presentation, the jury may pose some criticism, questions and suggestions. This item in the planning represents the assessment of those topics.
- **Certification Process Investigation:** Research about the fourth and fifth phases of the testing pipeline.
- **Certification Process Implementation:** Development of the fourth and fifth phases of the testing pipeline.
- **Results export:** Implementation of the export functionality - specifically to an Excel file.
- **User interface fine tuning:** Web page layout and design improvement.
- **Testing:** Testing carried on the application and its subsystems.
- **Paper:** Given the lack of research in this field, one of the goals of this thesis is to also write a scientific paper on the subject so we are saving some time to do it before the final delivery and presentation.
- **Final Presentation:** Time dedicated to the thesis' final public presentation.

To guarantee the quality of the final system, the Gantt chart suffered some changes. Figure 9 shows the real one. A higher resolution version can be found at https://goo.gl/2CBUhz at the two can be seen side by side for an easier comparison at http://goo.gl/DZ6RvB .

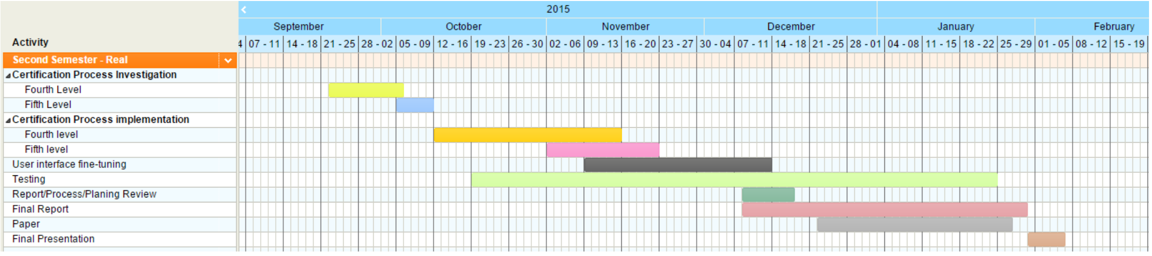


Figure 9. Second Semester Gantt Chart - Real

The research for the fourth and fifth phases took less time than expected. However, their implementation required much more effort than anticipated. The improvement of the GUI required more work. Testing remained about the same, but instead of starting right at the beginning of the semester, it began some days after the implementation of the fourth phase started. The juries' notes were taken into consideration when the final report started to be written, which took place towards the end of year, instead of right at the beginning.

The biggest difference from the planned Gantt chart to the real one is the absence of the "Results export" feature. Unfortunately, there was no remaining resources to implement it. Be as it may, said functionality was not considered crucial.

4.2. Risk Management

Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks. A look into them is taken using the same approach and terminology as in [33]:

1. The probability of the risk might be assessed as:
    a. Very low (< 10 %);
    b. Low (10-25%);
    c. Moderate (25-50%);
    d. High (50%-75%);
    e. Very high (>75%).
2. The impact of the risk might be assessed as:
    a. Catastrophic (threatens the survival of the project);
    b. Serious (would cause major delays);
    c. Tolerable (delays are within allowed contingency);
    d. Insignificant.

4.2.1. RK001 – Changing requirements

All the requirements were discussed and written down during the first semester, but, as with all things, we must embrace change and be prepared for it.

Probability: Moderate.

Impact: Tolerable.

Mitigation Plan: Introduce more meetings with the supervisor to control and prioritize requirements and / or drop some requirements altogether.

4.2.2. RK002 – Report written in a non-native language

Since this report is entirely written in English and the native language of the author is Portuguese, translation and grammar issues may arise and slow down the writing process.

Probability: High.

Impact: Tolerable.

Mitigation Plan: Ask the supervisor for some guidelines, since he has a vast experience with English papers/thesis. Allocate more time to the thesis writing task.

4.2.3. RK003 – Wrong estimation of the tasks

Only one person is responsible to develop all tasks and requirements. If they aren't delivered on time, this can cause a snowball effect and can jeopardize the entire project.

Probability: Moderate.

Impact: Tolerable.

Mitigation plan: Requirements and their specific tasks are classified using the MoSCoW technique in order to understand and establish priorities, dropping some requirements if needed to.

4.2.4. RK004 – Higher learning curve than anticipated

The system uses many different tools, frameworks and libraries with very different learning curves. The ramp-up period each one demands cannot be overlooked.

Probability: High.

Impact: Serious.

Mitigation Plan: If SpringMVC and Vaadin turn out to be too complex, the author can always lean on Strut2 plus Bootstrap and jQuery, all in which he already has a considerable amount of experience. MySQL could also be an alternative if PostgreSQL revealed itself to be too challenging.

4.2.5. RK005 – Over planning

This thesis consists in mainly two phases: an implementation phase where the development of the certification levels is somewhat straightforward and an investigation phase, where a part of the process requires a lot of research. Trying to establish a work plan in the beginning of the thesis to cover these two distinct phases might be problematic.

Probability: Very low.

Impact: Insignificant.

Mitigation Plan: Divide the thesis in two parts: the implementation part, which consists in the first three certification levels, goes to the first half of the thesis (first semester) and the research phase, which consists in the last two testing levels is taken care of in the second half (second semester).

Probability	Very high	🟧	🟥	🟥	🟥
	High	🟧	🟧 RK002	🟥 RK004	🟥
	Moderate	🟧	🟧 RK001, RK003	🟧	🟥
	Low	🟩	🟧	🟧	🟥
	Very low	🟩 RK005	🟩	🟧	🟥
		Insignificant	Tolerable	Serious	Catastrophic
		Impact			

Table 2. Risk matrix

Chapter 5
Support Technologies

All tools discussed in Chapter 2 are lacking in different ways: may it be for the lack of simplicity, forcing the user to use a specific operating system or simply because they do not test interoperability properly - or they do not test it at all out of the box. With that in mind, the tools and frameworks in this section will be used to create a web-based system that, in its core, is platform agnostic and will be able to assess the issue of interoperability in a proper manner.

This section starts by clarifying the selection criteria, and then the development environment is explained. Afterwards, a lengthy analysis of the web frameworks currently available on the market (including some popularity surveys and market adoption rate) is done and the section finishes by crosschecking all the information against the system's needs.

5.1. Selection criteria

Before dwelling into the support technologies, it is important to take note of our selection criteria. We want a framework that takes care of some aspects so we can focus on building the system itself. We are talking specifically about three major areas identified during the first meetings with the supervisor:

- Session management
- Database connection and querying
- Security

In the future, this project may be integrated with another application which was created using Spring, is currently using Maven as its dependency and build manager and is storing its data into a PostgreSQL database so we're also taking that into consideration. The programming language of choice will be Java, mainly because of the author's experience with it.

5.2. Development environment

The system will be developed under OSX. IntelliJ's IDEA 15 is the IDE chosen because of its amazing HTML5, CSS3 and Javascript capabilities. It also keeps winning Dr. Dobb's Jolt Productivity Award. [34]. Undertow [35] was chosen as the Web Server because of its flexible and lightweight nature (e.g. it uses less than 4Mb heap space). Having our selection criteria in mind, to build and manage the application, Maven was chosen [36] and to save all the information processed, a PostgreSQL database will be used. Last but not least, the department's deployment of GIT [37] will be used to serve as the code repository.

5.3. Web frameworks

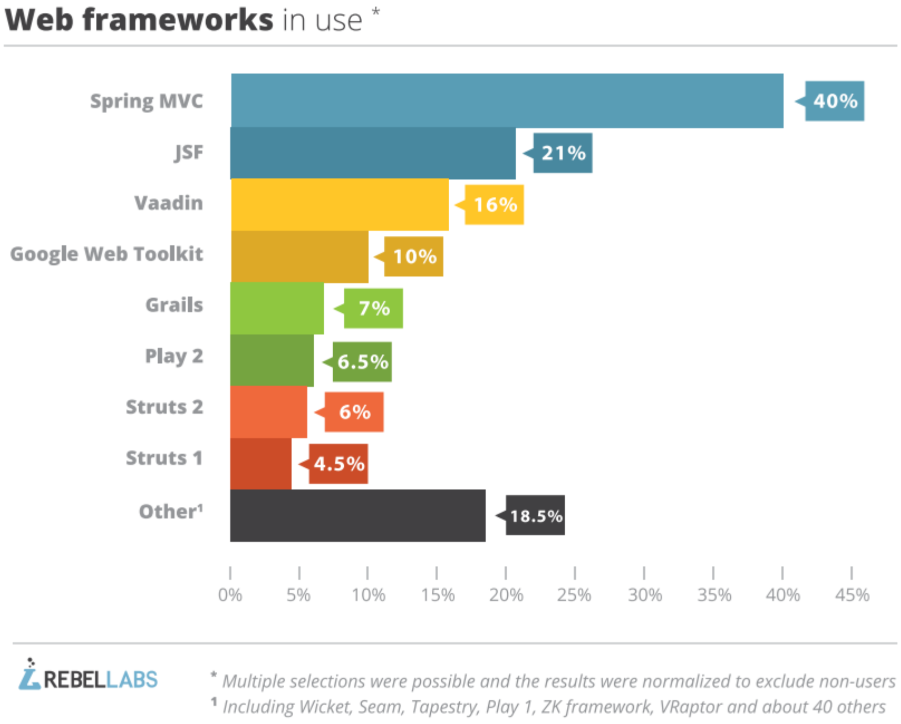
There are many options when it comes to the development framework. They divide into 3 major groups: full stack web frameworks, pure web frameworks and SOFEA.

Full-stack web frameworks gather multiple libraries useful for web development into a single cohesive software stack for web developers to use. They handle everything from web serving to database management right down to HTML generation. Examples include **Grails**, **Play** and **Lift**.

Pure web frameworks provide a more versatile development environment since many different plugins can be used and custom complex business rules can be more easily implemented. **Spring MVC**, **Google Web Kit**, **Vaadin**, **Wicket**, **Struts2** and **JSF** fall into this category.

SOFEA stands for Service-Oriented Front-End Architecture and consists in the idea that the server side should only provide data (e.g. via SOAP) and the whole MVC paradigm happens on the client. **AngularJS**, **BackboneJS** and **EmberJS** are clear examples of it.

For a quick overview on all these different frameworks, let us look at RebelLabs' analysis in Figure 10.



* Multiple selections were possible and the results were normalized to exclude non-users
¹ Including Wicket, Seam, Tapestry, Play 1, ZK framework, VRaptor and about 40 others

Figure 10. RebelLabs' Developer Productivity Report

The chart in Figure 10 was based on over 1800 developer responses. We can see that SpringMVC is easily the most popular framework available, followed by JSF and Vaadin. GWT comes in the fourth place and the last four places are reserved for Play 2, Grails and Struts, with a relatively low score. What might come as a surprise is that one in six developers (around 17%) do not use any real framework at all – just JSPs and servlets.

RebelLabs also compared these frameworks for several aspects. The results were [38] :

- **Rapid Prototyping**: Grails and Play
- **Framework Complexity**: Vaadin, GWT and Struts
- **Ease of use**: Grails and Vaadin
- **Documentation and Community**: Grails and Vaadin
- **Throughput/Scalability**: Play
- **UX, Look and Feel**: Vaadin and GWT

ThoughtWorks' Technology Radar also provides an interesting view on several technologies on the market. Figure 11 represents the chart currently on their website. [39].

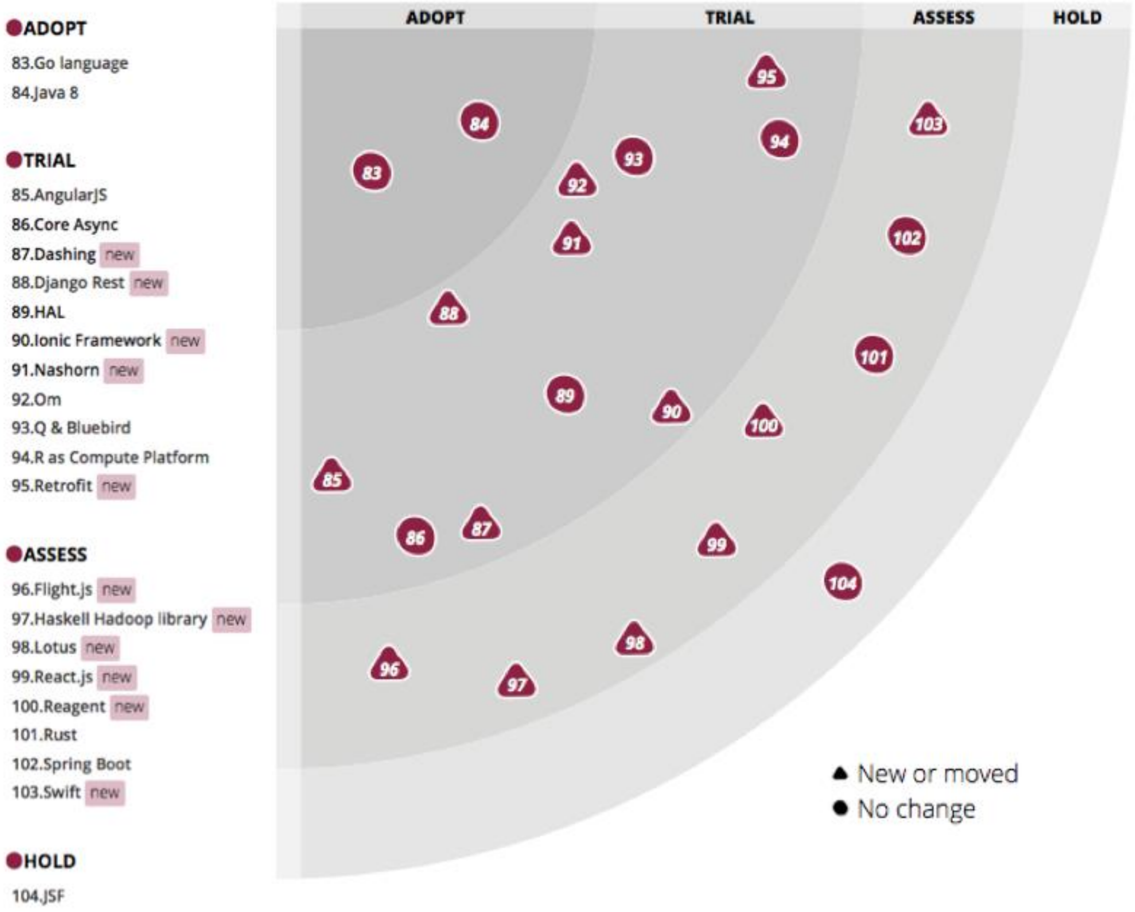


Figure 11. ThoughtWorks' Technology Radar

According to ThoughtWorks, the *Adopt Ring* represents blips that they think the user should be using now and is something where there is no doubt that it is proven and mature for use. The *Trial Ring* is for blips that they think are ready for use, but not as completely proven as those in the adopt ring, so for most organizations they think the user should use these on a trial basis, to decide whether they should be part of his/her toolkit. The *Assess Ring* are things that the user should look at closely, but not necessarily trial yet – unless he/she thinks they would be a particularly good fit. Finally, the *Hold Ring* is for things that are getting attention in the industry, but ThoughtWorks do not think are ready for use. Sometimes this is because they do not think they are mature enough yet, sometimes it means they think those things are irredeemably flawed. There's no "avoid" ring, but they throw things in the *hold ring* that they wish their clients would not use.

5.4. Web framework adoption

Technical specs only tell half the story and so a real look into the market's framework adoption is needed. In order to achieve it, Matt Riable, AppFuse's creator [40], took a different approach when analyzing the great amount of web frameworks available where he compared them with real-world market adoption numbers. The focus was on:

- Number of jobs on a job aggregator site
- Job trends on Indeed.com
- Interest over time on Google
- Number of questions on StackOverflow

In order to not analyze every single framework available and exclude some of them right at the start, we adapted his research and the graphs in this section were created using the following elimination criteria:

- At least 1 release in 2014
- At least 1 book on Amazon.com
- 10 jobs on itjobs.pt
- At least 300 questions on StackOverflow
- As we have stated at the beginning of this Chapter, programming language must be Java or based on Java at least - so SOFEA frameworks will not be analyzed.

In Figure 12 we can see number of job positions for each framework. SpringMVC and Struts2 dominate the market, followed by JSF and GWT. No other frameworks fulfilled the 10 minimum jobs criteria.

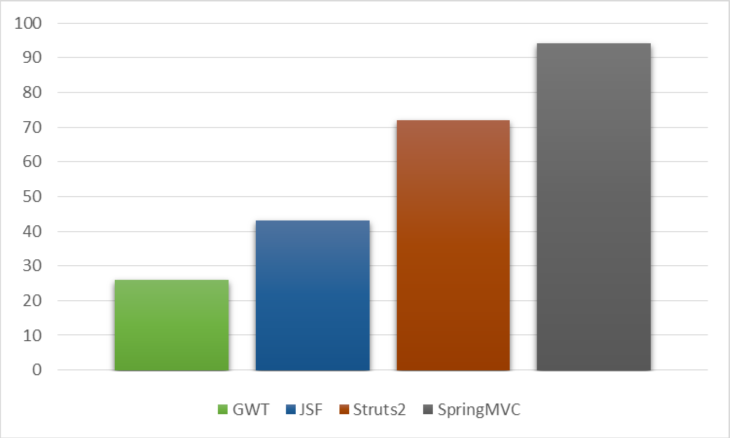


Figure 12. Jobs on itjobs.pt

Figure 13 and Figure 14 represent the job trends on Indeed.com. As we can see, on the full stack web framework side, Grails had a huge spike from 2009 until 2012 but kept declining since that time (becoming slightly more popular in 2015). Even with the strong decline, it remains the most relevant of all full stack frameworks.

Focusing on pure web frameworks, Spring dominates the business since 2006, with all the alternatives not even coming close.

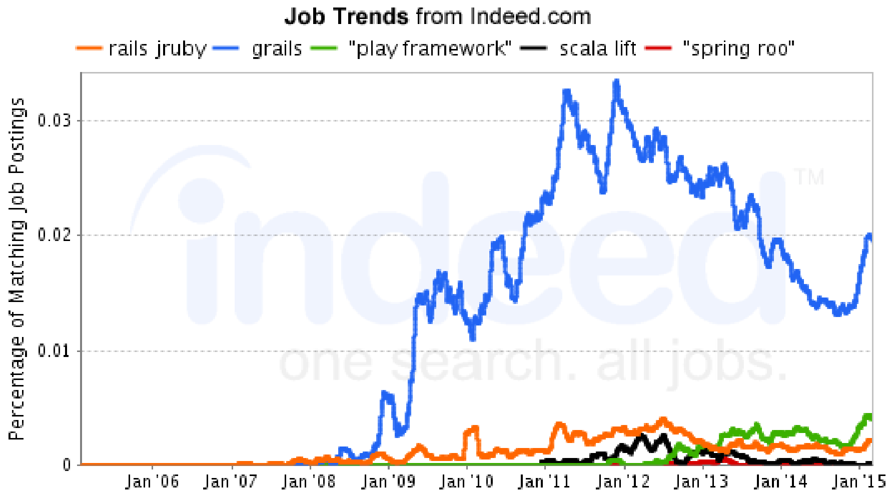


Figure 13. Job Trends – Full stack

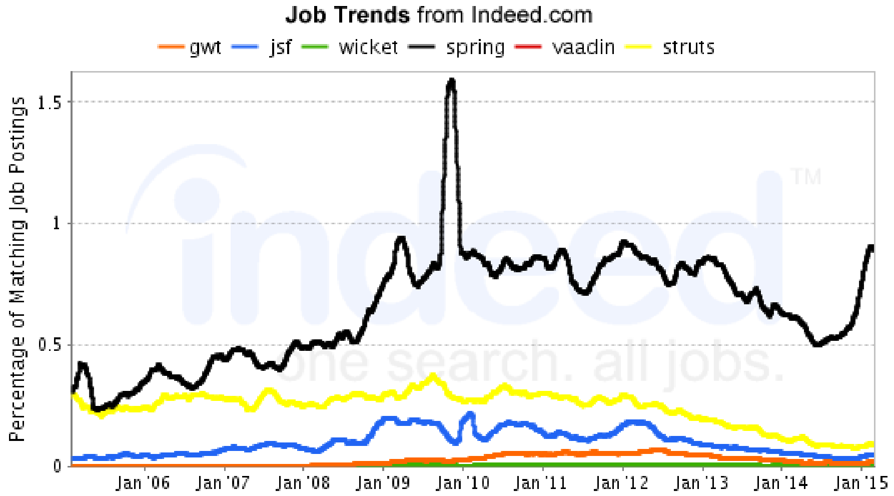


Figure 14. Job Trends – Pure Web

Figure 15 and Figure 16 represent the interest over time in Google searches. Grails is clearly a popular framework on the full stack side, with Play coming second and all the alternatives tied in third place with similar results.

On the pure web side of searches, JSF kept being the most popular until 2010 when it matched GWT. Nowadays, they are all similar.

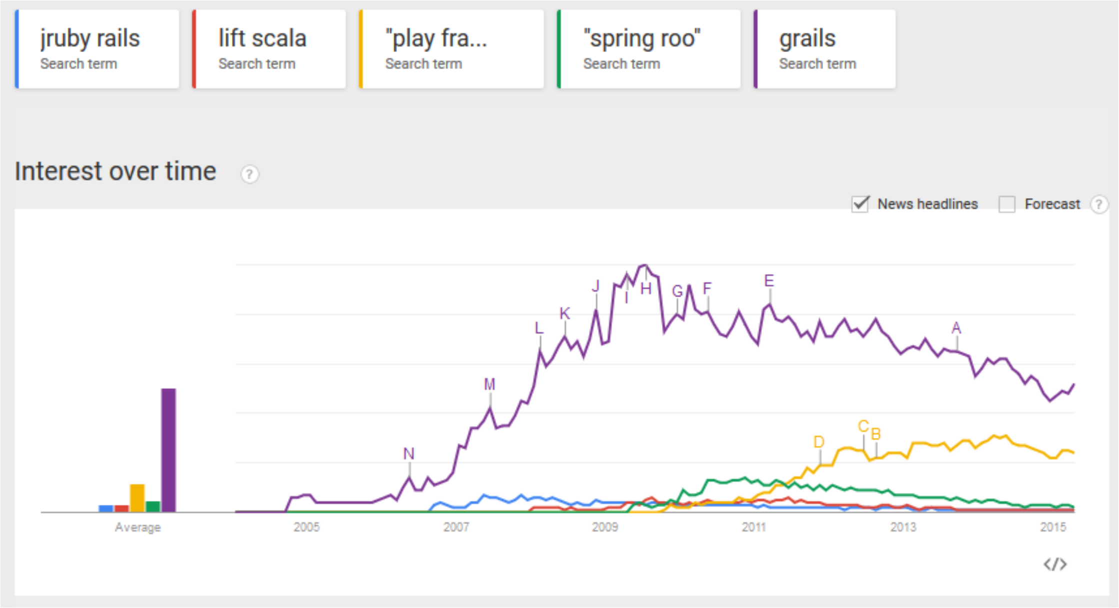


Figure 15. Google Trends – Full Stack

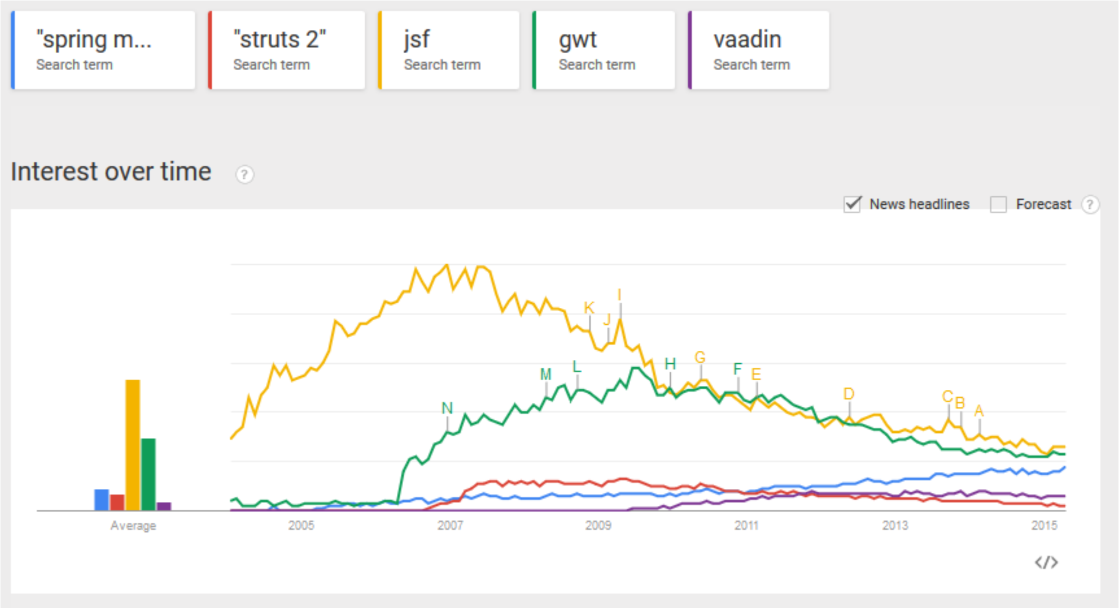


Figure 16. Google Trends – Pure Web

Looking at the StackOverflow question scenario, Figure 17 shows us that Grails completely obliterates the competition with more than twenty thousand questions raised in the platform. Play comes in a very distant second place and JRuby on Rails, Lift and Spring Roo tie for third.

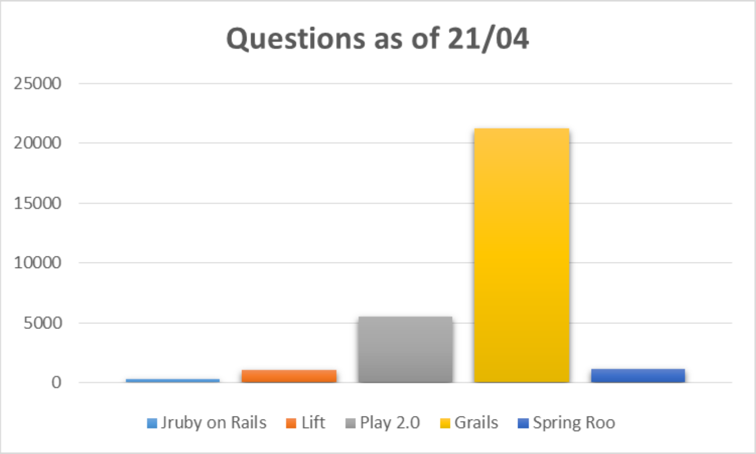


Figure 17. StackOverflow Tagged Questions – Full Stack

When we switch to pure web frameworks, SpringMVC and JSF tie for having the highest amount of questions asked – around twenty five thousand. GWT comes in second place with roughly nineteen thousand, Strust2 in third with barely ten thousand and Vaadin and Wicket tie for last place with two and a half thousand.

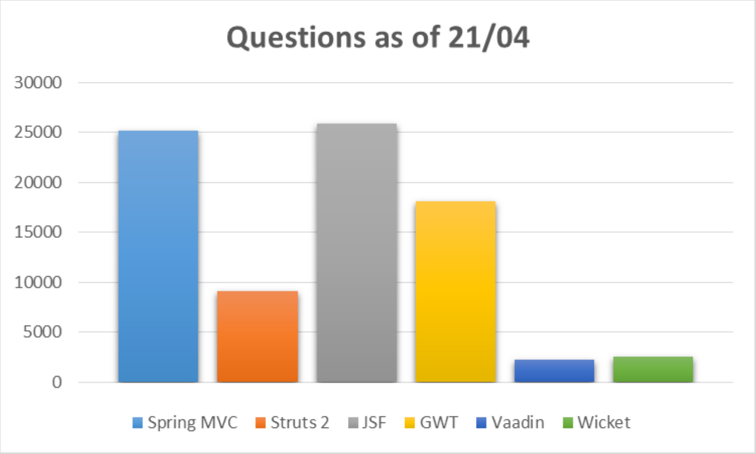


Figure 18. StackOverflow Tagged Questions – Pure Web

5.5. Web framework analysis

Picking up on the results of the last two subsections (5.3 and 5.4), we reach the conclusion that the most important and relevant ones are **Grails, Play, Lift Wicket, SpringMVC, GWT, Vaadin, Struts** and **JSF.** We will now analyze each one of them into more detail while also applying the selection criteria mentioned in section 5.1.

It is important to say that, even with our own selection criteria, all of these final nine frameworks could fit our needs, one way or another. Mix and match is a viable option and, in some cases, they even use the same underlying technologies (e.g. Spring Security is or can be used in most of them). There is no single silver bullet or one framework to rule them all. It all comes down to our own preference and sometimes we need to go an extra-mile to rule one out (e.g. Wicket, GWT) – that is why the reader will notice that the reasons and the set of rules to exclude a certain framework will not always be the same.

5.5.1. Grails

Grails is one of the most popular frameworks nowadays. It is ranked first in DevRates [41] and got highly praised in the RebelLabs' The Curious Coder's Java Web Frameworks Comparison [38]. Unfortunately, one needs to know Groovy to use this framework and even though it is based on Java, it brings an extra overhead to the development process.

5.5.2. Play

This framework also managed to get excellent scores all across the board and, unlike Grails, uses pure Java as the main programming language. The problem lies within the market adoption rate. This thesis also serves as a showcase for the author's skills and if the market is not looking for those skills, it feels like a wasted effort to a certain extent. It also uses Scala in some parts of the framework, so we are ruling it out.

5.5.3. Lift

Lift does not have a relevant presence in any of the areas analyzed, so there is very little reason to use it.

5.5.4. Wicket

On 2011, a study called The World Wide Wait [42] took place in Devoxx, a European Java, Android and HTML5 [43]. They did a comparison between Wicket, JSF, Spring, MyFaces and GWT. They found out that Wicket, JSF and MyFaces had enormous costs of scale. Figure 19 illustrates the situation:

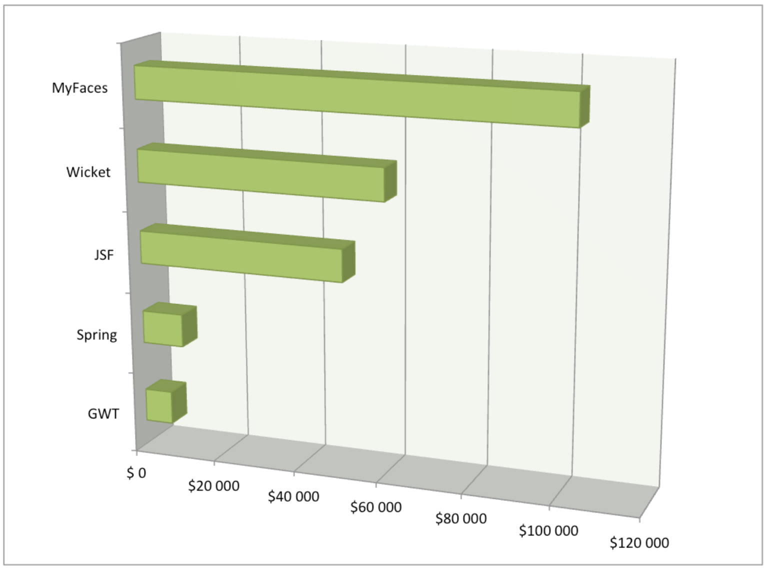


Figure 19. Framework cost of scale

For this reason, Wicket won't be used.

5.5.5. SpringMVC

Of all the Java frameworks currently available, SpringMVC is probably the one that has been relevant the longest and still continues to innovate. As we have stated previously, our project will eventually be integrated into a system that was built using this framework. It must also use Maven and using it with SpringMVC is quite straightforward. Concerning authentication, Springs' own Spring Security is used by a number of other frameworks and is one of the best solutions in its field. All of this makes SpringMVC a very strong candidate.

The biggest issue lies in its initial configuration. The framework is massive and is hard to grasp if one's just starting out.

5.5.6. Google Web Kit

On February 2013, Iván García Sainz-Aja posted to AppFuse's board an implementation of the application which included GWT functionality [44]. To see how the GWT flavor compared to the other implementations in AppFuse, Matt Raible (one of AppFuse's developers) created a cloc report – cloc stands for "count lines of code" and is a utility that counts blank lines, comment lines, and physical lines of source code in many programming languages - on the various web frameworks in AppFuse [45]. We have created two graphs based on this information to better illustrate the scenario, which are represented in Figure 20 and Figure 21.

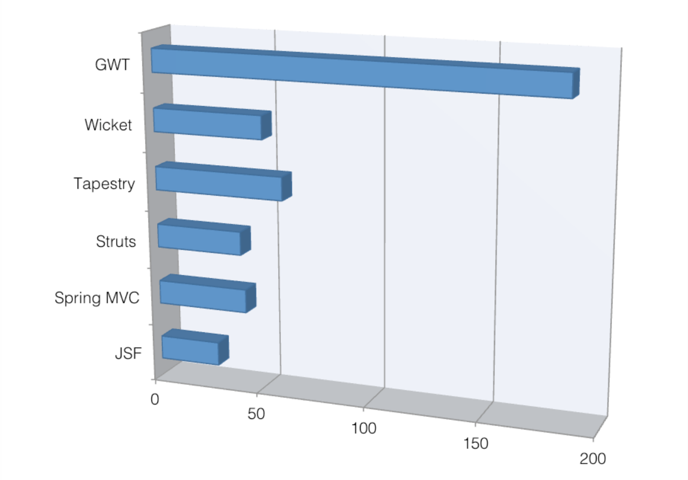


Figure 20. Number of files

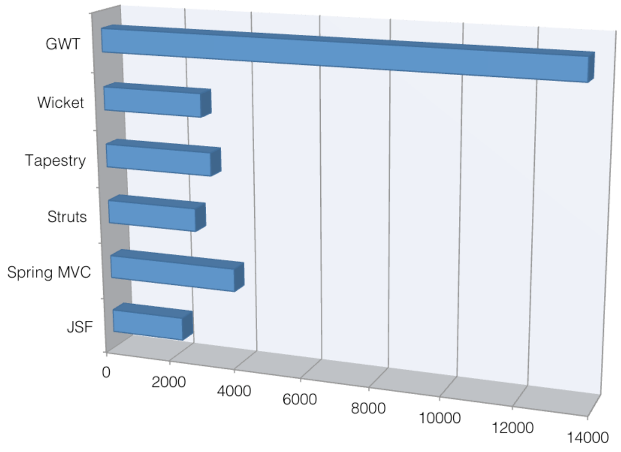


Figure 21. Lines of java

It's important to note that this implementation followed many of the GWT best pratices: MVP pattern, activities and places, EventBus, Gin and Guice [46]. Since we don't want to maintain so much code, GWT won't be used.

5.5.7. Vaadin

Vaadin also came on top on RebelLab's tests. It allows one to implement an entire UI layer using an Object Oriented model, so one can use Object Oriented knowledge to design UI components. It eliminates the task of designing and implementing the client-server communication that is usually required for AJAX applications.

5.5.8. Struts/Struts2

Struts was the first web framework to provide the MVC paradigm. All of the other frameworks today exist because of it. Struts should no longer be used because Struts2 superseded it.

Strust2 is pure MVC with a more or less straightforward architecture (e.g. no extra components that may add complexity), however it can be seen as a legacy technology as there is lots of boilerplate code [47], no built-in code generation and no external powerful tools.

5.5.9. Java Server Faces

ThoughtWorks' Technology Radar [39] is quite interesting as they put JSF "on hold". They say that teams are running into several problems with it and are recommending the user to avoid this technology. ThoughtWorks think that JSF is flawed because its programming model encourages use of its own abstractions rather than fully embracing the underlying web model. JSF attempts to create stateful component trees on top HTML markup and the stateless HTTP protocol. The improvements in JSF 2.0 and 2.2, such as the introduction of stateless views and the promotion of GET, are steps in the right direction, maybe even an acknowledgement that the original model was flawed, but they feel this is a too little too late – and so this framework won't be used.

According to the World Wide Wait study previously mentioned (Figure 19), it also has an enormous cost of scale, along with Wicket.

5.6. Web framework final verdict

Based on all previous criteria, a combination of Spring Boot plus Spring MVC and Vaadin will be used. The major issue with Spring MVC is the amount of time needed to configure it properly. This can be addressed with Spring Boot: it provides a radically faster and widely accessible 'getting started' experience for all Spring development; it is opinionated out of the box, but gets out of the way quickly as requirements start to diverge from the defaults. It also provides a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration). Vaadin will facilitate the presentation view since it eliminates the task of designing and implementing the client-server communication that is usually required for AJAX applications. Using IntelliJ's Spring Initializr, it is extremely easy to "glue together" these tools.

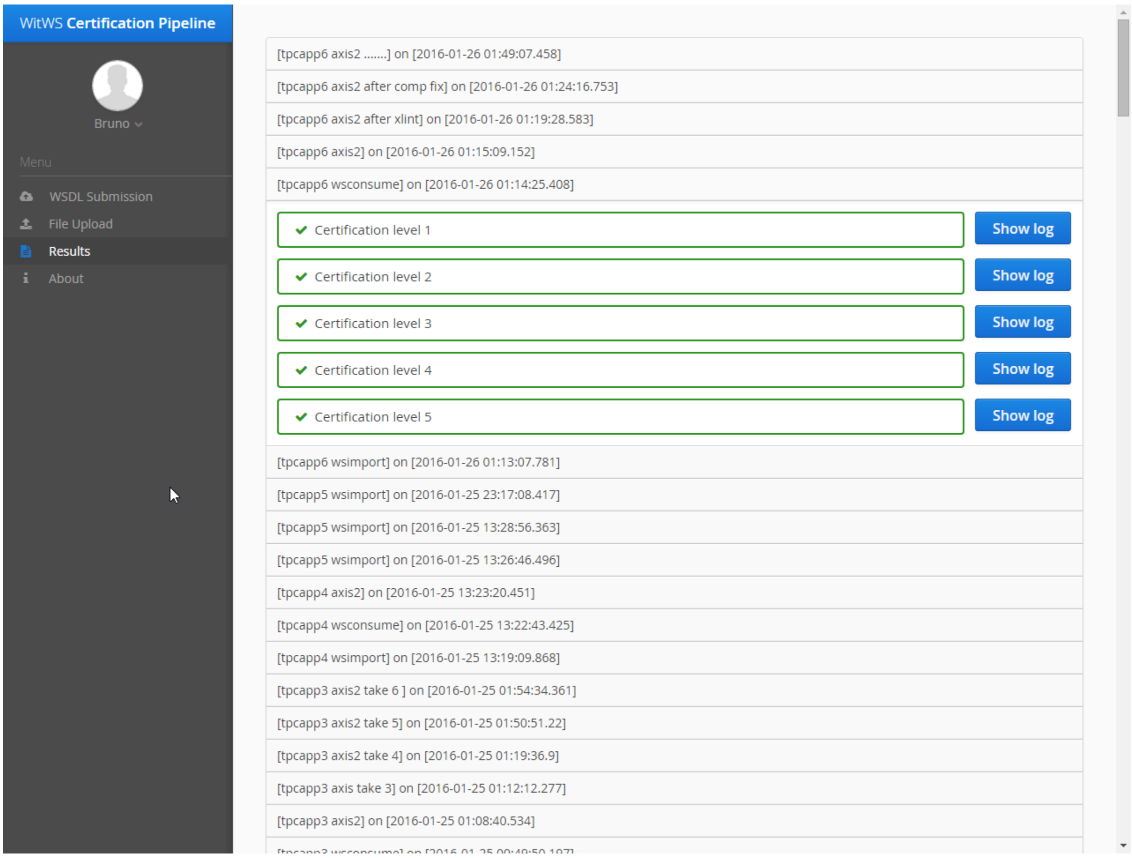
Chapter 6
Architecture and Operation

This section will examine the architecture built to achieve the objectives stated in 1.3 - Objectives by using a "top-down" approach, starting by showing an overview of the GUI to give a rough idea what can be done using the application, followed by use cases, passing to the overall architecture, the MVC pattern and finally the data model.

For a much more detailed analysis on the background of a typical process in this application, please refer to the **Annexes, section C1 – Web service source code upload background process.**

6.1. WitWS Overview

Figure 22 shows the layout of the application and one of its main views and features: the Results' view. It contains all the user's test runs (pending, executing and completed). This one and the rest of the user's interaction with the web page can be seen in the next section, Use Cases.



[tpcapp6 axis2 ....] on [2016-01-26 01:49:07.458]

[tpcapp6 axis2 after comp fix] on [2016-01-26 01:24:16.753]

[tpcapp6 axis2 after xlint] on [2016-01-26 01:19:28.583]

[tpcapp6 axis2] on [2016-01-26 01:15:09.152]

[tpcapp6 wsconsume] on [2016-01-26 01:14:25.408]

✓ Certification level 1

✓ Certification level 2

✓ Certification level 3

✓ Certification level 4

✓ Certification level 5

[tpcapp6 wsimport] on [2016-01-26 01:13:07.781]

[tpcapp5 wsimport] on [2016-01-25 23:17:08.417]

[tpcapp5 wsimport] on [2016-01-25 13:28:56.363]

[tpcapp5 wsimport] on [2016-01-25 13:26:46.496]

[tpcapp4 axis2] on [2016-01-25 13:23:20.451]

[tpcapp4 wsconsume] on [2016-01-25 13:22:43.425]

[tpcapp4 wsimport] on [2016-01-25 13:19:09.868]

[tpcapp3 axis2 take 6] on [2016-01-25 01:54:34.361]

[tpcapp3 axis2 take 5] on [2016-01-25 01:50:51.22]

[tpcapp3 axis2 take 4] on [2016-01-25 01:19:36.9]

[tpcapp3 axis take 3] on [2016-01-25 01:12:12.277]

[tpcapp3 axis2] on [2016-01-25 01:08:40.534]

[tpcapp3 wsconsume] on [2016-01-25 00:49:50.197]

Figure 22. Results view

The complete set of views and features can be seen in chapter Chapter 10 User Manual in more detail.

6.2. Use cases

The use case relevant to the scenario is represented in Figure 23. It contains the relevant actions the user can perform on the application. For the sequence diagrams of "Submit online WSDL" and "Upload web service source code", please go to the Annexes, Sequence Diagrams section.

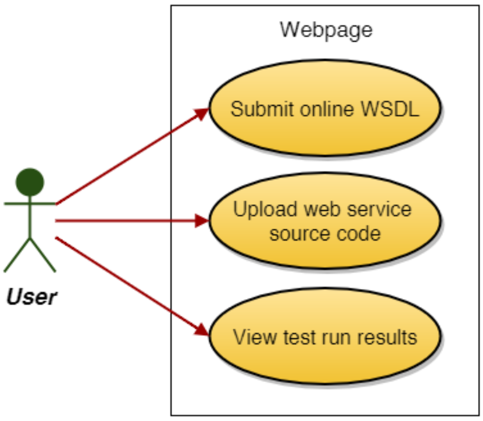


Figure 23. Webpage use case

The description of each use case can be seen in the following tables, using the format specified in [33].

WitWS: Submit online WSDL	
Actors	Web page users
Description	A user may submit any WSDL which is already available online for the system to test. It's possible to select the client code generation tool when submitting the WSDL.
Data	WSDL endpoint of the web service.
Stimulus	User command initiated by pressing the "Start testing" button.
Response	Web page notification indicating that the test process has started.
Comments	The user needs to be registered in the system and must have already logged in to access this functionality.

WitWS: Upload web service source code	
Actors	Web page users
Description	A user may submit any web service (along with its dependencies) source code to the application for the system to analyze. It's possible to select the client code generation tool and the deployment server when submitting the code.
Data	Web service source code and any class dependencies it might have.
Stimulus	User command initiated by pressing the "Start testing" button.
Response	Web page notification indicating that the test process has started.
Comments	The user needs to be registered in the system and must have already logged in to access this functionality.

WitWS: View test run results	
Actors	Web page users
Description	A user may consult all his submitted test runs (already completed, pending and still executing).
Data	N/A.
Stimulus	User command initiated by clicking on each test run and "Show log".
Response	Test run's testing phases and corresponding logs.
Comments	The user needs to be registered in the system and must have already logged in to access this functionality.

6.3. Architecture overview

To implement the use cases in the previous section, a system represented by Figure 24 was built.

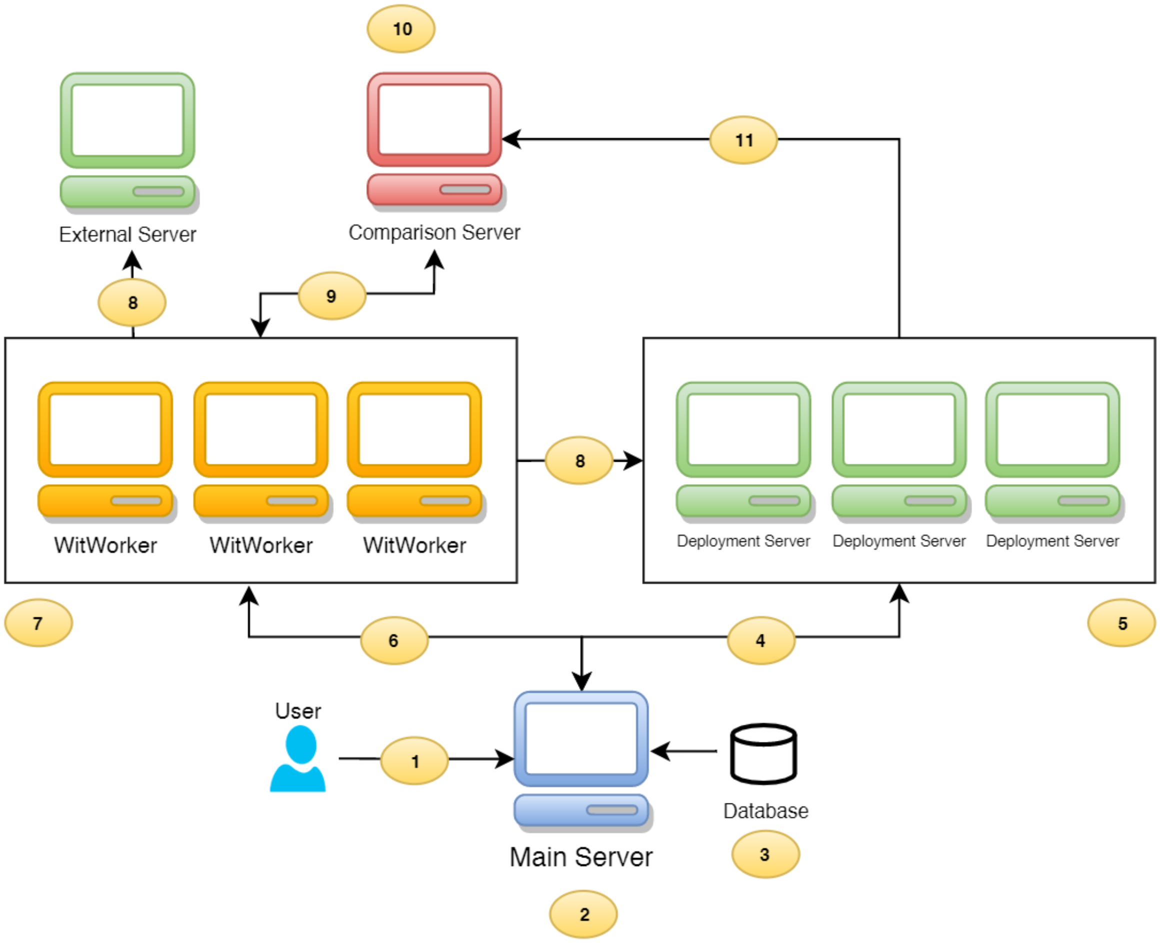


Figure 24. Architecture overview

The architecture represented in Figure 24 contains several annotations but it's important to mention that those number annotations do not represent a particular communication order.

(1) Vaadin: The main entrance point of the application. It represents a user connecting to the web page built in Vaadin – which is located in the main server. Vaadin is responsible for abstracting the HTML code and to facilitate the access to the beans created in the main server. Using this framework, it was possible to design the entire application without writing a single line of HTML or JavaScript.

(2) Main server: The main server/machine. It contains the webserver and the majority of the application's logic. It is built following the MVC paradigm around Spring MVC. Fetches all the information contained in the database, sends it to the Vaadin layer, creates new test runs and forwards them to the appropriate WitWorker/Deployment server, among other things.

(3) Database: The PostgreSQL database, which is located in the same machine as the main server, contains all the info about users, logs, deployment servers, tools and test runs. The main server fetches the information of what to do and where to do it from here. In order to facilitate all the database related operations, Hibernate is used as the JPA provider and Spring Data JPA hides Hibernate behind its repository abstraction. The data model and an overview of the database tables can be found in section 6.4.

(4) Communication between main server and deployment servers: Represents the communication between the main server and the deployment servers available. Main server and deployment servers communicate exclusively through FTP. In fact, it's just a one-way communication channel – from the main server to the deployment one. The other way around is not necessary, so it is not used.

(5) Deployment servers: As previously mentioned, this application allows the users to submit their own web service code (and its dependencies) in order to go through a more comprehensive set of interoperability tests. One of the steps of one of those tests consists in deploying the code to a deployment server.

(6) Communication between main server and WitWorker servers: Represents the communication between the main server and the WitWorker servers.

(7) WitWorker servers: Regardless of the comprehensiveness of the tests being run, they always go through one WitWorker server. They are responsible for generating code, compiling code, build the run log, etc.

(8) Communication between the WitWorkers and the External Server/Deployment servers: The tests run on the platform can either be started from an online WSDL - represented by the "External Server" - or by submitting the source code and deploying it to our own servers – represented by the "Deployment Servers" in square **5.** This communication is initiated by the WitWorkers to the wsdl endpoints of those services.

(9) Communication between the WitWorker servers and the comparison server: This arrow represents the communication between the WitWorker servers and the comparison server.

(10) Comparison server: Two of the testing phases in the pipeline require the system to generate random values for one component and send them to another component. The comparison server, as the name suggests, is a server which compares these randomly generated values.

(11) Communication between the deployment servers and the comparison server Represents the communication between the deployment servers and the comparison server.

The system follows the MVC architectural pattern, which is represented in Figure 25. MVC Pattern. Vaadin represents the presentation layer, the controller is built around Spring and the model translates to the combination of JPA and Hibernate. The data model is explained in the next section.

.

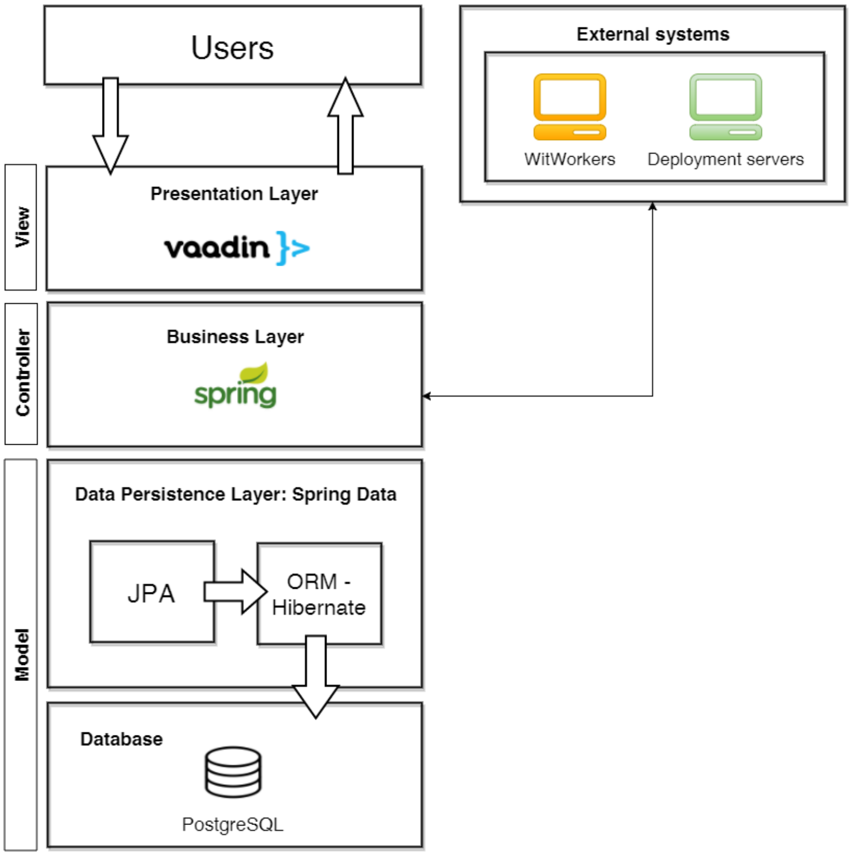
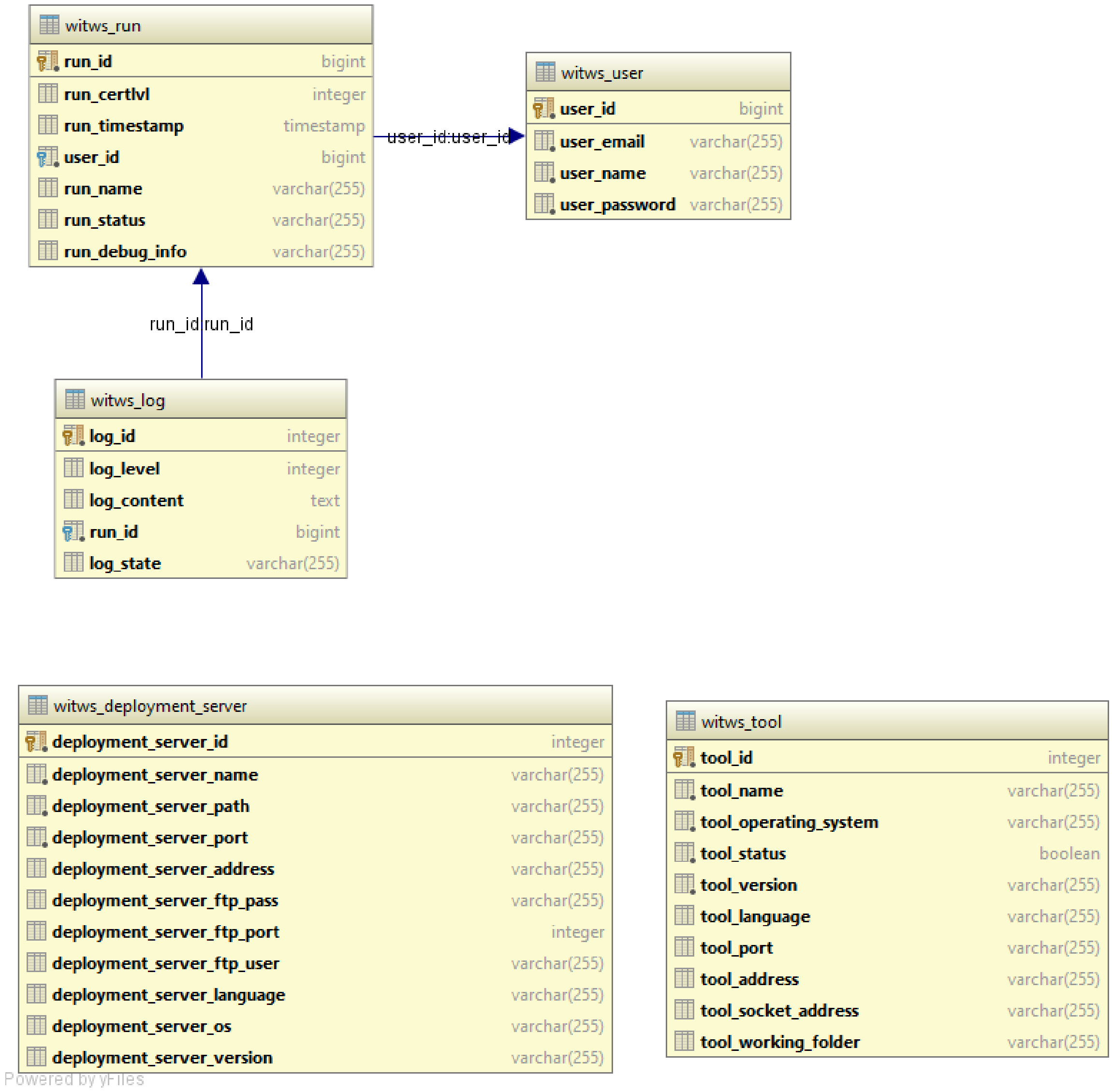


Figure 25. MVC Pattern

Notice that in the "External Systems" box, there is no Comparison Server. This is because the Comparison server does not communicate directly the controller, so it was left out of the figure.

6.4. Data Model

Figure 26 represents the physical data model of the application.



witws_run	
run_id	bigint
run_certlvl	integer
run_timestamp	timestamp
user_id	bigint
run_name	varchar(255)
run_status	varchar(255)
run_debug_info	varchar(255)

witws_user	
user_id	bigint
user_email	varchar(255)
user_name	varchar(255)
user_password	varchar(255)

witws_log	
log_id	integer
log_level	integer
log_content	text
run_id	bigint
log_state	varchar(255)

witws_deployment_server	
deployment_server_id	integer
deployment_server_name	varchar(255)
deployment_server_path	varchar(255)
deployment_server_port	varchar(255)
deployment_server_address	varchar(255)
deployment_server_ftp_pass	varchar(255)
deployment_server_ftp_port	integer
deployment_server_ftp_user	varchar(255)
deployment_server_language	varchar(255)
deployment_server_os	varchar(255)
deployment_server_version	varchar(255)

witws_tool	
tool_id	integer
tool_name	varchar(255)
tool_operating_system	varchar(255)
tool_status	boolean
tool_version	varchar(255)
tool_language	varchar(255)
tool_port	varchar(255)
tool_address	varchar(255)
tool_socket_address	varchar(255)
tool_working_folder	varchar(255)

Figure 26. WitWS Data Model

6.4.1. Table witws_user

Info about the user. Straightforward without any special attribute. Maintains a OneToMany relationship with the witws_run table.

6.4.2. Table witws_run

The witws_run table stores the information about each user run and maintains a ManyToOne relationship with witws_user and a OneToMany with witws_log. Some columns worth of note:

- run_certlvl – The highest certification level attained in a specific run. Basically, consists in the log_level associated in the witws_log table.
- run_name – The name given by the user in the webpage, before starting the run.
- run_status – The current status of the run: "pending", "executing" or "completed".
- run_debug_info – Used for debugging purposes when the run fails to reach the first level of certification. The application stores the error in this column and reads it later on on the Results page.

6.4.3. Table witws_log

Data about the log of each run. One per each Certification Level. Maintains a ManyToOne relationship with the witws_run table.

- log_level – The level of the certification level being analysed.
- log_content – The output of said certification level.
- log_state – The outcome of the certification phase. "success", "warning" or "error".

6.4.4. Table witws_deployment_server

Used to keep track of the available deployment servers to be displayed in the webpage. The info here is added mannually by the developer.

- deployment_server_name – The name of the deployment server (Wildfly, Glassfish, etc)
- deployment_server_path – The deployment path of the server. In other words, the path where the server looks for WAR files to automatically deploy. Always relative to the configuration of its FTP server.
- deployment_server_address – Address of the server in the form of
- deployment_server_ftp_ ... – Columns which store the corresponding FTP server configuration.
- deployment_server_language – Web service programming language supported by the server.

6.4.5. Table witws_tool

Tools and WitWorker servers do not exist without one another, so in order to facilitate the extensibility of the app, the database table "witws_tool" tracks the available client code generation tools and their corresponding RMI address because that's how the system communicates with them.

- tool_name: The name of the tool. E.g.: wsconsume, wsimport.
- tool_operating_system: Programming language supported the tool.
- tool_status: Available/Unavaible;
- tool_address: RMI address of the RMI server running in the tool's machine.
- tool_socket_address: Address where the comparison server must send its answer to.
- tool_working_folder: The absolute path of the folder the main server should send the WitSerialization helper class to.

Chapter 7
Implementation details

Implementing the architecture and the system exposed in the previous chapter demanded many technical decisions. Some of them are explained in this section. For more details about the implementation, refer to the Annexes - sections A1, B1, C1 and D1.

7.1. Vaadin

The presentation layer of WitWS is built around Vaadin. Comparing this application to a system using a traditional combination of Bootstrap and jQuery, the entry point (the index.html) is the DashboardUI class and there are no other UI classes in project. This means that whatever page the user requests (e.g. http://witws.dei.uc.pt/#!submit), it will always go through the DashboardUI first and will update the page content's based on the current user status – if the user if logged in, main view is shown (or the page request in the URL. "submit" in this case), otherwise, login view is presented.

The GUI is built by Java classes which represent each view of the system – AboutView, LoginView, ResultsView, etc

7.2. Protocol used by the Comparison server

As soon as the Comparison server starts, it waits for a string containing the following format:

testID | origin | destination | number_of_following_write_operations | pair | language

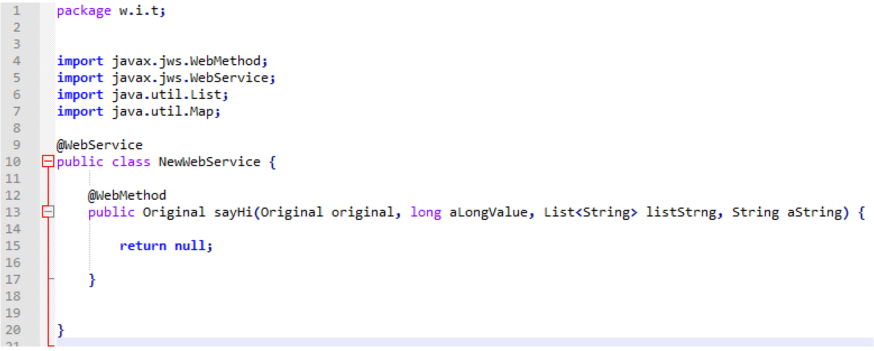
- testID – the ID of the test run
- origin – Where the test is coming from. 'cc' for client code and "ws" for webservice.
- destination – It is the server which the Comparison Server should send its results to. It was previously injected in both the Invoker and in the web service code.
- number_of_following_write_operations – Size of the map that is going to be received after this message.
- pair – Number corresponding to the pair of the operation. 1 if it is the invoker-web service pair or 2 if it is web service-invoker response pair.
- language – Programming language of the web service / invoker.

This string (and all the communication to and from the Comparison Server) actually initiates with 2 bytes. These 2 bytes express the number of 8-bit characters, which will follow up, so that the other side is able to read the exact quantity or fail. This is the same contract used in the Java language in the methods *writeUTF* and *readUTF* of the data stream classes and can be implemented in any language that supports the sockets abstraction. Also, we are not interested at the moment in a more complex protocol, which would allow us to recover from failures (i.e., logging and re-sending lost messages, using negative acknowledgments, etc.). We are simply interested in understanding if the information is being received as a whole or not. If there is a problem in the communication, WitWS interrupts the interoperability test.

After the first reading operation, the comparison server reads N more times in the following structure:

variable_structure | variable_type | variable_value

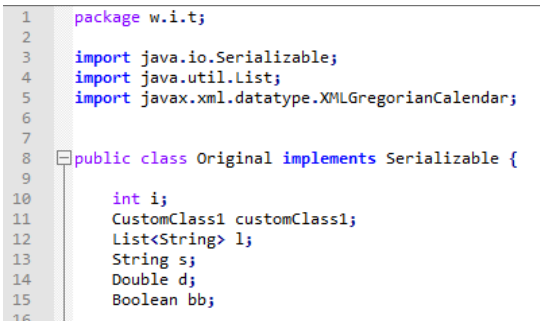
For a better understanding of this structure, let us take a look at Figure 27, Figure 28 and Figure 29. Notice that some code is intentionally omitted because it is not important in this example. Figure 27 represents a web service with a web method which takes several parameters: an Original type value (a class created by the developer), a long type value , a list of strings and a string.



```
package w.i.t;


import javax.jws.WebMethod;
import javax.jws.WebService;
import java.util.List;
import java.util.Map;

@WebService
public class NewWebService {

    @WebMethod
    public Original sayHi(Original original, long aLongValue, List<String> listStrng, String aString) {

        return null;

    }


}
```

Figure 27. NewWebService.java - Webservice example

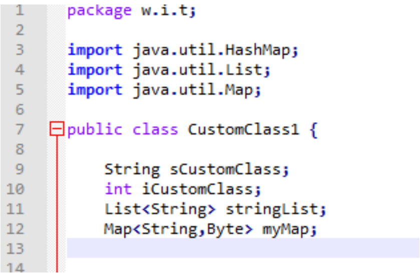
Figure 28 is a screenshot of the Original class code. It contains a an integer, a CustomClass1 (another class created by the developer), a list of strings, a string, a double and a Boolean as fields.



```
package w.i.t;

import java.io.Serializable;
import java.util.List;
import javax.xml.datatype.XMLGregorianCalendar;


public class Original implements Serializable {

    int i;
    CustomClass1 customClass1;
    List<String> l;
    String s;
    Double d;
    Boolean bb;
```

Figure 28. Original.java - class used as parameter and return value

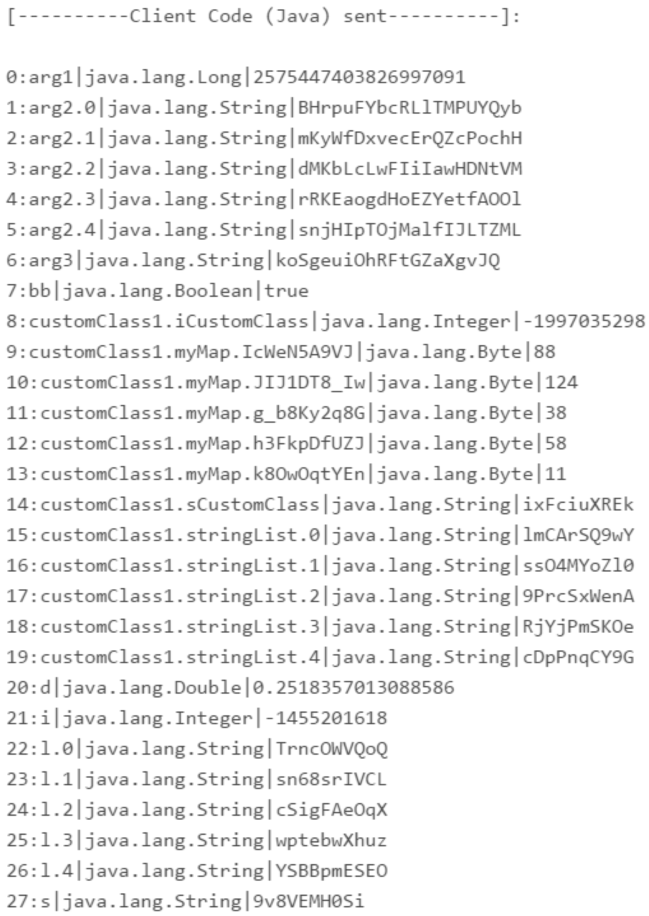
Figure 29 is another code screenshot, but this time it shows the CustomClass1 class. As fields, we can see a string, an integer, a list of strings and a Map – with strings as keys and bytes as values.



```
package w.i.t;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class CustomClass1 {

    String sCustomClass;
    int iCustomClass;
    List<String> stringList;
    Map<String,Byte> myMap;
```

Figure 29. CustomClass1.java - class used by Original.java

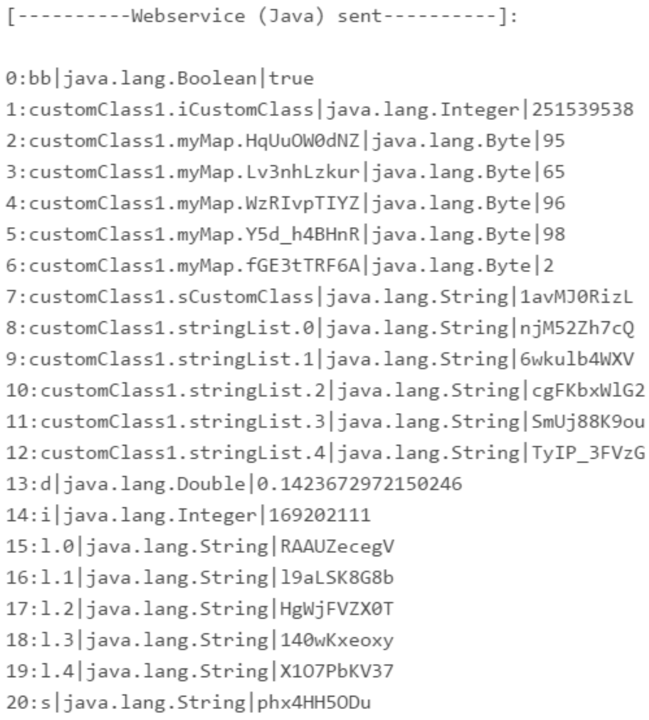
After being submitted to the Certification Pipeline, these classes produced the output in Figure 30 during phase 4 (some output omitted).



```
[----------Client Code (Java) sent----------]:

0:arg1|java.lang.Long|2575447403826997091
1:arg2.0|java.lang.String|BHrpuFYbcRLlTMPUYQyb
2:arg2.1|java.lang.String|mKyWfDxvecErQZcPochH
3:arg2.2|java.lang.String|dMKbLcLwFIiIawHDNtVM
4:arg2.3|java.lang.String|rRKEaogdHoEZYetfAOOl
5:arg2.4|java.lang.String|snjHIpTOjMalfIJLTZML
6:arg3|java.lang.String|koSgeuiOhRFtGZaXgvJQ
7:bb|java.lang.Boolean|true
8:customClass1.iCustomClass|java.lang.Integer|-1997035298
9:customClass1.myMap.IcWeN5A9VJ|java.lang.Byte|88
10:customClass1.myMap.JIJ1DT8_Iw|java.lang.Byte|124
11:customClass1.myMap.g_b8Ky2q8G|java.lang.Byte|38
12:customClass1.myMap.h3FkpDfUZJ|java.lang.Byte|58
13:customClass1.myMap.k8OwOqtYEn|java.lang.Byte|11
14:customClass1.sCustomClass|java.lang.String|ixFciuXREk
15:customClass1.stringList.0|java.lang.String|lmCArSQ9wY
16:customClass1.stringList.1|java.lang.String|ssO4MYoZl0
17:customClass1.stringList.2|java.lang.String|9PrcSxWenA
18:customClass1.stringList.3|java.lang.String|RjYjPmSKOe
19:customClass1.stringList.4|java.lang.String|cDpPnqCY9G
20:d|java.lang.Double|0.2518357013088586
21:i|java.lang.Integer|-1455201618
22:l.0|java.lang.String|TrncOWVQoQ
23:l.1|java.lang.String|sn68srIVCL
24:l.2|java.lang.String|cSigFAeOqX
25:l.3|java.lang.String|wptebwXhuz
26:l.4|java.lang.String|YSBBpmESEO
27:s|java.lang.String|9v8VEMH0Si
```

Figure 30. Example classes phase 4 output

The output produced in phase 5 is displayed in Figure 31 (some code omitted).



```
[----------Webservice (Java) sent----------]:

0:bb|java.lang.Boolean|true
1:customClass1.iCustomClass|java.lang.Integer|251539538
2:customClass1.myMap.HqUuOW0dNZ|java.lang.Byte|95
3:customClass1.myMap.Lv3nhLzkur|java.lang.Byte|65
4:customClass1.myMap.WzRIvpTIYZ|java.lang.Byte|96
5:customClass1.myMap.Y5d_h4BHnR|java.lang.Byte|98
6:customClass1.myMap.fGE3tTRF6A|java.lang.Byte|2
7:customClass1.sCustomClass|java.lang.String|1avMJ0RizL
8:customClass1.stringList.0|java.lang.String|njM52Zh7cQ
9:customClass1.stringList.1|java.lang.String|6wkulb4WXV
10:customClass1.stringList.2|java.lang.String|cgFKbxWlG2
11:customClass1.stringList.3|java.lang.String|SmUj88K9ou
12:customClass1.stringList.4|java.lang.String|TyIP_3FVzG
13:d|java.lang.Double|0.1423672972150246
14:i|java.lang.Integer|169202111
15:l.0|java.lang.String|RAAUZecegV
16:l.1|java.lang.String|l9aLSK8G8b
17:l.2|java.lang.String|HgWjFVZX0T
18:l.3|java.lang.String|140wKxeoxy
19:l.4|java.lang.String|X1O7PbKV37
20:s|java.lang.String|phx4HH5ODu
```

Figure 31. Example classes phase 5 output

It now becomes clear what **variable_structure, variable_type** and **variable_value** are. Looking at the code above, particularly at Figure 30, one can see that the "primitive" parameters of the web method (the ones recognized by the Java programming language) are represented as "arg0" , "arg1", "arg2". The rest are fields inside other classes and are "nested" as the system, recursively, inspects all fields, for all classes passed as parameters. E.g.: The field "bb" is "inside" the first level of nesting, as we can see by the code above – it belongs to the Original class passed as parameter to the web method. Using the same logic, it's easy to understand that the customClass1.iCustomClass variable represents an integer field, which belongs to a "custom class" named customClass1 which, in turn, is a field in the Original class. When it comes to lists and maps, some more detail might be needed:

- Since lists can have any kind of generic type, the index position is used as the name of each value of the list. E.g. Figure 31, line 8 through 12.
- Maps also need some extra work. The key of each entry of the map is used as its variable name (and the value of that entry is the **variable_value**). E.g.: Figure 31, line 2 through 6.

After the comparison process, it then creates several answers for the WitWorker. The order is:

1. **testID | "cc" | pair1ClientListSize | lvlDebugInfo | language**
    - **a.** testID – ID of the test run;
    - **b.** "cc" – Means "client code" which is used by the WitWorker to control the order in which it is receiving the information;
    - **c.** pair1ClientListSize – Size of the list it going to send in the next message, so the server knows how many reading operations it needs to do;
    - **d.** lvlDebugInfo – Number used for debug purposes. Analyzed at the WitWorker. HTTP like codes:
        - **i.** 200 – OK
        - **ii.** 500 – Internal server error
        - **iii.** 404 -- Couldn't contact the server
    - **e.** language – Programming language of the client code.
2. **pair1ClientList**
    - **a.** The same list it received earlier, one element at a time, following the structure explained above (variable_structure|variable_type|variable_value).
3. **testID | "ws" | pair1WebserviceListSize | lvlReachedStructure | lvlReachedType | lvlReachedValue | language**
    - **a.** testID – ID of the test run
    - **b.** "ws" – Means "web service" which is used by the WitWorker to control the order in which it is receiving the information;
    - **c.** pair1WebserviceListSize -- Size of the list it going to send in the next message, so the server knows how many reading operations it needs to do;
    - **d.** lvlReachedStructure – Similarity level of the structures analyzed.
        - i. 0 – Not even similar;
        - ii. 1 – Similar enough or equivalent;
        - iii. 2 – Equal;
    - **e.** lvlReachedType – Similarity level of the types analyzed:
        - i. 0 – Not even similar;
        - ii. 1 – Similar enough or equivalent;
        - iii. 2 – Equal;
    - **f.** lvlReachedValue – Similarity level of the values analyzed:
        - i. 0 – Not even similar;
        - ii. 1 – Similar enough or equivalent;

        iii. 2 – Equal;
   - **g.** language – Programming language of the web service;

4. **pair1WebserviceList**
   - **a.** Same logic as above;
5. **testID | "ws" | pair2WebserviceListSize | lvlDebugInfo | language**
   - **a.** Same logic as above;
6. **pair2WebserviceList**
   - **a.** Same logic as above;
7. **testID | "cc" | pair2ClientListSize | lvlReachedStructure | lvlReachedType | lvlReachedValue | language**
   - **a.** Same logic as above;
8. **pair2ClientList**
   - **a.** Same logic as above;

7.3. Custom Class Loader

WitWS utilizes reflection in many parts of the testing pipeline in order to get the information it needs from the classes submitted. To do it, it must load the web service class to a Class Loader. E.g.:

- Class c = Class.forName("com.package.MyClass");

Once the class loader has imported the class, it won’t change anymore. There is no way of unloading such a class. As a result, the system can't change the class being loaded. This becomes a problem when, for example, the user submits two classes with different contents but with the same name.

After numerous tries, it was decided to adapt the method used by Lê Anh Quân [48].

Chapter 8
Experimental Evaluation

In this section the experimental evaluation is described, which was designed essentially to verify and validate the tool's capabilities in detecting interoperability issues. The next sections describe the scenarios used and main results obtained during the experiments.

The test bed consisted in:

- Client side code generation tools:
  - JAX-WS 2.3.1
  - JBossWS 5.0.0
  - Axis2 1.6.4
- Server side application servers:
  - WildFly 9.0.1 Final
  - Glassfish 4.1

Every possible combination between the client-side platforms and server-side was used during the interoperability tests, thus resulting in a total of 6 combinations. Regarding the services, we considered the following scenarios (the whole set of services and results is available at https://goo.gl/8iGdXf).

- **Synthetic services:** A set of 10 custom services, created to provide initial different test cases to exercise the tool. The services range from simple cases (a single and simple argument for service operation) to more complex cases (service operations accepting complex parameters involving lists, maps, and custom complex objects with nested complex objects).
- **Realistic services:** A set of 10 web services specified by the TPC-App benchmark [49]. The goal here is simply to demonstrate the application of WitWS to a realistic scenario and further exercise the tool and disclose any possible bug in the tool.
- **Real services:** Selected cases of 80 web services publicly available on the Internet. In this case the goal is to, based on the authors results in [6], show that WitWS is based to detect known web service framework bugs and at the same time does not signal inexistent problems.
- **Faulty services:** The most complex service of the synthetic services was selected to be used with a fault injector. This fault injector applies three different types of faults into the data being exchanged between client and server and vice-versa. Thus, it emulates 10 different cases (5 problems detectable at the server-side plus 5 at the client-side). The fault types are:
  - i) structure changing faults;
  - ii) data type changing faults;
  - iii) value mutation faults (which can be value replacement, value addition, or value removal).

8.1. Results

Table 3 summarizes the results of the experimental evaluation, showing the results per service set and per framework used. The results for each of the inter-operation levels (regarding each combination service set / framework) are further detailed. Note that we do not show the results per server-side framework, as they were found to be the same. In the table, *w* means one or more *warnings* were found; *e* refers to one or more *errors* being found; and a dash indicates that the tests were not run for that level.

Sets	**Metro**					**JBossWS**					**Axis2**				
	I	**II**	**III**	**IV**	**V**	**I**	**II**	**III**	**IV**	**V**	**I**	**II**	**III**	**IV**	**V**
Synthetic													*w*	–	–
Realistic													*w*	–	–
Real	*w*	*e*		–	–	*w*	*e*		–	–	*w*	*e*	*e*	–	–
Faulty				*e*	*e*				*e*	*e*			*w*	–	–

Table 3. Results of the interoperability tests

As we can see in Table 1, WitWS was able to detect different problems in all steps of the inter-operation process, with exception of the deployment step where we only uncovered warnings (resulting from the execution of the WS-I tests). In fact, authors in [7] emphasize that the deployment in current major platforms is quite robust, failing whenever the rules are not fulfilled (e.g., a class not being provided, or a wrong structure in the deployment file), but if the deployment complies with the basic rules it's very difficult to find a fatal problem. Obviously we also tried to deploy undeployable services, which served to verify the correctness of WitWS. Anyway, we did not manage to get an error in step 1, using a service that complies with the deployment rules (e.g., having a java class annotated with @*WebService*).

Regarding the **synthetic set**, no error was found, in any of the five inter-operation levels, probably due to the small size of the experiments, but anyway the set was useful for verifying WitWS basic functionalities. However, we did observe *warnings* using Axis2, related with the absence of typed collections during the compilation process, for all services in this set.

The results regarding the **realistic set** were similar to the above. Clearly, we were expecting detecting more problems in the Real services, as they include services built on many different platforms, and also obviously in the Faulty services. In the case of the **real services**, we selected cases from the set used by the authors in [6], for validating our tools results. 12.5% of the services in this set were selected for being known to not generate any kind of problem [6], which was confirmed by our tool. Of the remaining, we highlight one case resulting in an *error* in Level II with JBossWS (which however did not fail with Metro or Axis2). We also highlight 10 services failing with Metro and JBossWS but not failing with Axis2 in Level II. In 9 of these services a compilation problem was detected when using Axis2 in Level III. Also present in this more heterogeneous set, were some WS-I warnings, which we used to fine-tune WitWS warning detection capabilities. Obviously, when using the real services set we are unable to reach steps 4 or 5 (marked with a dash in Table 1), as we do not have access to the service code or infrastructure.

Finally, we used the faulty service jointly with a fault injector that emulates a framework holding bugs. As we were not using a weakly statically typed programming language in these experiments, we were not able to change the structure or object types, although we simulated these two situations by changing the information travelling to the comparison server, which handled these cases properly. However, we managed to detect three kinds of problems in the communication, even when the communication is slightly different due to the use of the different frameworks. Thus, we were able disclose errors due to missing values, extra values and modified values.

Chapter 9
Application Limitations and Extensibility

As the development of the system progressed, its complexity also rose. In some cases, the external libraries being used to facilitate the implementation of some tasks (e.g. populate POJOs) started to trigger some bugs. In other cases, there wasn't time to implement every single desirable feature and compromises had to be made. In this section, a deeper look into the limitations and extensibility of the system is provided, followed by a small guide on how to lift some of those limitations and how to extend the current functionality of the system.

9.1. Limitations

The PODAM library [50] is used to populate the "non-primitive" classes (classes created by the developer) with random data. However, it contains some limitations when going through that process.

First of all, the classes submitted by the user must contain setters and getters in the "conventional" format – we're not calling it standard because there isn't one. The naming convention it looks for is:

- private String qwertyField;

Should have its getters and setters in the following format:

- getQwertyField() / setQwertyField()

But:

- private String sQwerty;

Translates to:

- getsQwerty() / setsQwerty()

There is also the case with field names with only 1 character:

- private integer i;

Which, by following Java naming conventions, should have getters and setters like:

- getI() / setI();

If any of the files submitted contains field names in any other format, PODAM won't be able to populate the pojo, and so the system must verify each file against this compliance. The class src\main\java\com\dei\witws\misc\SettersAndGettersVerifier.java does just that.

PODAM is also unable to populate pojos if they contain a BigDecimal field type or a LinkedHashMap. In the case of the former, it causes the JVM to run out of memory regardless of how high the –Xmx parameter value was set when running the server. The latter triggers a generic error. The application searches for these field types right after looking for the setters and getters compliance by calling the searchForTypes() method in the \src\main\java\com\dei\witws\misc\NoSupport.java class.

When the system compiles the classes, right before the serialization process, even though we're using Java 8 and compiling with the flag "-parameters" it was not possible to retain the parameter names at run time, and so, when serializing, the method parameters are passed as "arg0", "arg1", "arg2", etc. Doing it inside the IDE worked, but using the javax.tools.JavaCompiler and calling the java compiler from inside the Java code, somehow, didn't.

The tools in the deployment servers also have a limitation. Since the system is communicating through FTP with the deployment servers (and FTP is the only server available on those machines), the cleanup method at the end of the testing process isn't as efficient as it is in the WitWorkers. When a test run finishes, the testing manager in the main server connects to the deployment server used and deletes the contents from the remote application server's auto-deployment directory. This, effectively, undeploys the web service from the server, however, after several deployments and undeployments, the application server might not behave as expected and might start causing memory leaks. The solution would have been to drop the FTP connection and logic all together and opt for RMI server in the deployment machines to restart the application server manually.

Lastly, the "File upload" feature supports web services with any number of web methods but only processes the first one it finds.

9.2. Extensibility

The extensibility of WitWS exists in the form of:

- Analyzing any number of web methods inside a web service;
- Adding more tests to the Testing Pipeline;
- Adding more WitWorkers servers that can support more client code generation tools for the user to choose from;
- Adding more deployment servers;
- Adding more Velocity templates to deal with those new tools and deployment servers.
- Usage of the Comparison Server by other systems.

The following subsections explain how to do it.

9.2.1. Analyze any number of web methods

The system can only analyze the first web method it finds inside a web service. To lift this limitation, the developer needs to follow the following steps:

- **com/dei/witws/views/UploadView.java**
    - In a for loop, pass all methods to the noSupport.searchForTypes() method.
- **com.dei.witws.controllers.SerializationManager#createModifiedWebservice**
    - Search for non-standard getters and setters in all methods;
- **com.dei.witws.controllers.SerializationManager#saveToTemplate**
    - Pass all info necessary from all fields to the method;
- **templates/template_invoker_java.vm**
    - Modify the template accordingly;
- **templates/template_webservice_java.vm**
    - Modify the template accordingly;

9.3. Add more phases to the Testing Pipeline

Right now there are five different test phases in the Testing Pipeline, but it's trivial to add more.

Navigate to com.dei.witws.controllers.CertificationLevelInterface and create a new class extending the CertificationLevelInterface which represents your new test phase and implement the "process" method. Implement your own code and return a Messase type variable with the following fields properly populated:

- com.dei.witws.communication.Message#log
- com.dei.witws.communication.Message#failedCondition
- com.dei.witws.communication.Message#warningCondition
- com.dei.witws.communication.Message#status
- (If status != 200) com.dei.witws.communication.Message#debugInfo

Afterwards, the developer just needs to go to the enum com.dei.witws.controllers.PipelineManager.CertificationLevels and insert its new phase anywhere in the process. E.g:

- COMPLIANCE_CHECK(new Compliance()),
- GENERATE_CODE(new Generate()),
- COMPILE_CODE(new Compile()),
- NEW_PHASE(new Qwerty());
- CLIENT_TO_SERVER(new ClientToServer()),
- SERVER_TO_CLIENT(new ServerToClient());

9.3.1. Add client code generation tools

WitWorkers and tools are connected in the database, as in they share the same table. For the system to recognize a new WitWorker server, the developer needs to add all the relevant data to the witws_tool table in the database located in the main server. However, for the system to actually work with the server is a different issue.

The server must have a RMI server up and running. Keep in mind that adding a new tool to a WitWorker server can demand a lot of work, even if it's in Java. For example, Axis2's wsdl2java generates artifacts in a completely different way than wsconsume or wsimport. For a new tool to work with the system, some details must be taken into consideration:

- Fail and warning conditions of the tool must be added to the client code generation phase – if they aren't already present.
- If the tool generates code with different dependencies, those must be added to the classpath – otherwise the compilation step will fail.

If the developer is adding a new physical server to generate client code in a language other than Java, looking at that new server as a black box (in which the dev is responsible for the code, configuration and operations inside), there are only two additional points to keep in mind:

- The main server communicates with it through RMI. Both ways.
- It must communicate with the Comparison server through tcp/sockets. The comparison server also answers through tcp/sockets.

The first bullet point should be quite straightforward. Since the TestManager in the main server just calls the methods in the RMI server and expects some text logs in return. The second one must be implement in the same way as explained in 7.2 - Protocol used by the Comparison server .

9.3.2. Add deployment servers

By adding a new deployment server to the pool, the developer needs to verify how the service is actually deployed. Besides that, the system appends a string "Service" to the name of the WAR file before sending it to the appropriate server for deployment.

Example:

The user uploads a SimpleWsdl.java file. After all the processing, it creates a SimpleWsdlService.war and sends it to the deployment server. If the user chose Wildfly as the deployment server, the final URL to the webservice wsdl endpoint will be:

http://youraddress.com/SimpleWsdlService/SimpleWsdl?wsdl

However, if the user chooses Glassfish, the URL is slightly different:

http://youraddress.com/SimpleWsdlService/SimpleWsdlService?wsdl

If the developer wishes to add more servers to deploy the user's web service, besides installing the software and adding the appropriate info to the deployment_server database table, he must also add the format of the final URL to the *initiateCertProcessFromClassUploaded* method in *\src\main\java\com\dei\witws\controllers\PipelineManager.java.*

9.3.3. Add Velocity templates

Velocity templates are the most important aspect of the extensibility of this application. Right now, it only supports Java web services and Java client code generators, but new functionality can be added.

To support client code generation tools which are not Java-based, the dev must also create new templates and deal with the peculiarities of each implementation.
A new file under the /resources/templates folder must be created. Recommendation:

- template_invoker_xxxxx.vm – For the client code Invoker, where "xxxxxx" is the programming language of the artifacts generated by the new client code generation tool.

For web services in other languages:

- template_webservice_yyyy.vm .

In both cases, the developer must handle the peculiarities of each implementation.

9.3.4. Comparison server usage by other systems

As stated previously, the system does not have much flexibility when it comes to handling web services written in other languages besides Java. It does not mean, however, that the Comparison server can only accept information from Java code. The whole comparison process and the communication from and to the comparison server was built with extensibility being one of the major impacting factors. By following the protocol explained in section 7.2, WitWorker and deployment servers written in another language can be easily added.

Chapter 10
User Manual

In this section the features of the WitWS Certification Pipeline will be presented.

The first screen the user sees when access the application is the Login screen. Figure 32. In here there are just two options. Entering credentials and logging in or signing up for a new account.

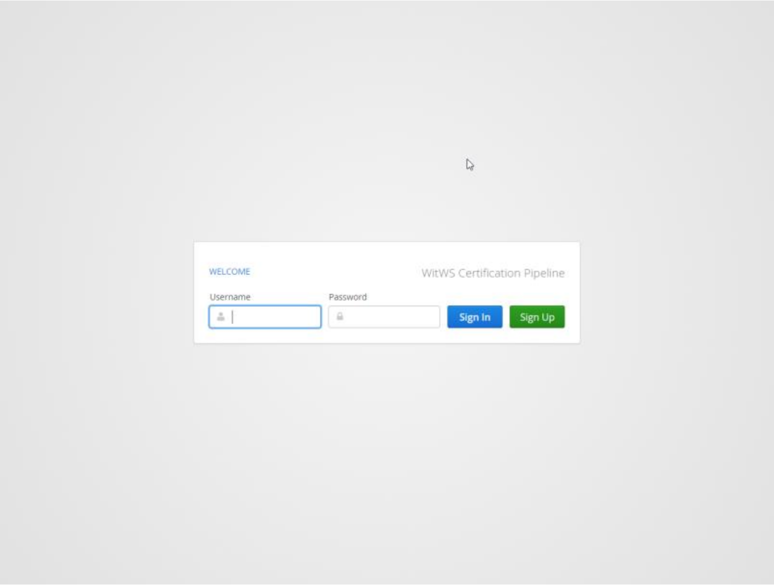


Figure 32. Login Screen

Figure 33 shows what the Registration screen looks like when the user presses "Sign Up" in the previous screen.

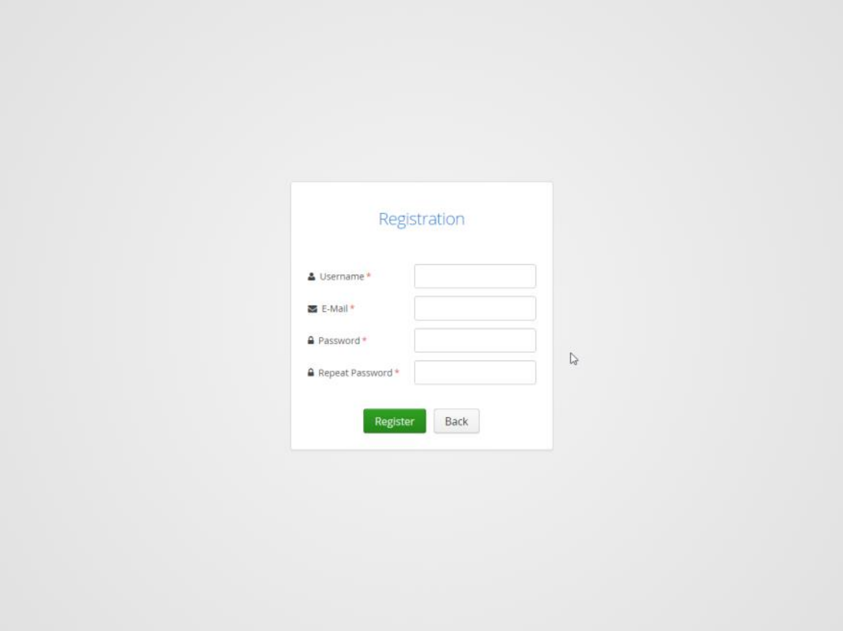
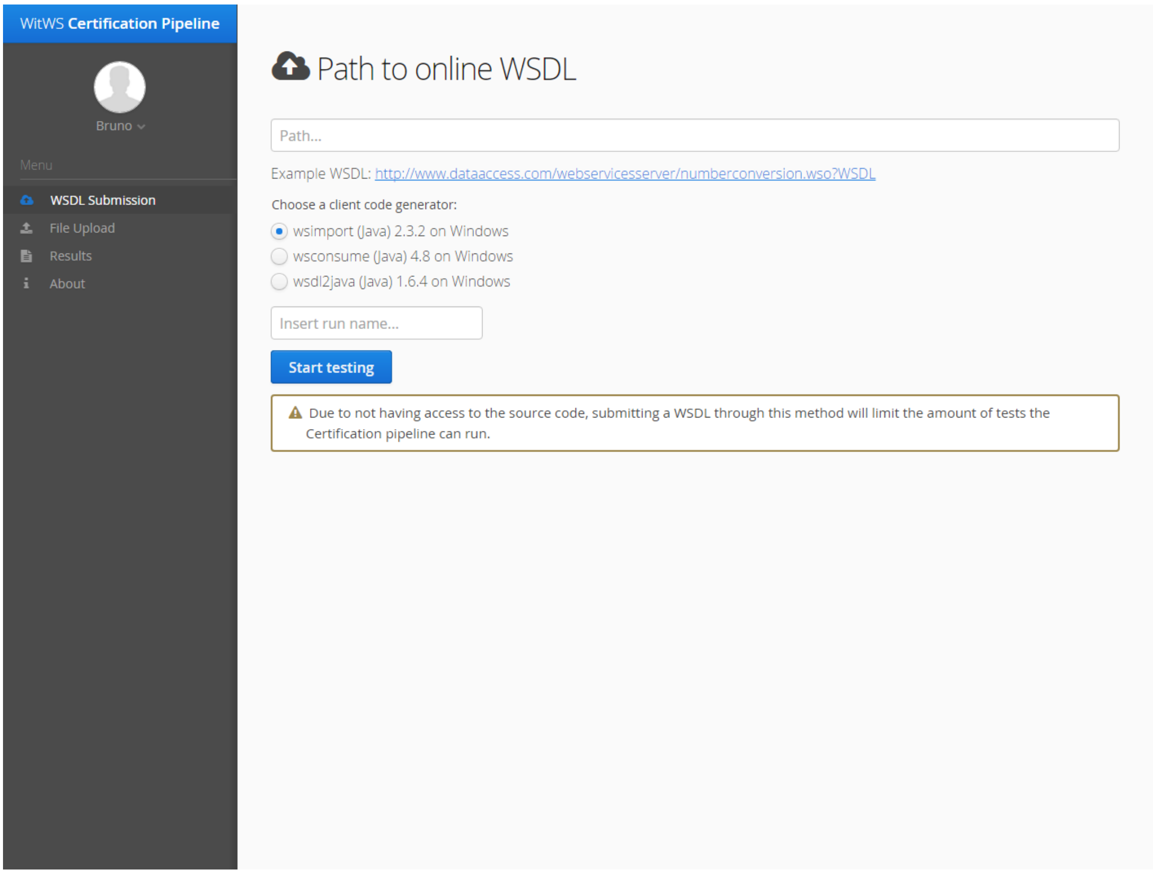


Figure 33. Registration Screen

When the user enters valid credentials, he is taken to the Dashboard screen – the main screen of the application. Regardless of where he is located, the left menu bar is always displayed. His name and profile icon is also displayed on the top left corner. The default menu option selected when the user enter the application is the "WSDL Submission", as we can see in Figure 34

In this screen the user can submit a WSDL URL for the system to process – there's an example of one right below the "path" field. Below it, the user can select a client code generator to process the WSDL. A run name must also be provided because it will make the task of tracking runs in the "Results" page much easier.

This feature does not support the full testing pipeline and there's a warning label in the page to indicate that.



Path to online WSDL

Path...

Example WSDL: http://www.dataaccess.com/webservicesserver/numberconversion.wso?WSDL

Choose a client code generator:

● wsimport (Java) 2.3.2 on Windows
○ wsconsume (Java) 4.8 on Windows
○ wsdl2java (Java) 1.6.4 on Windows

Insert run name...

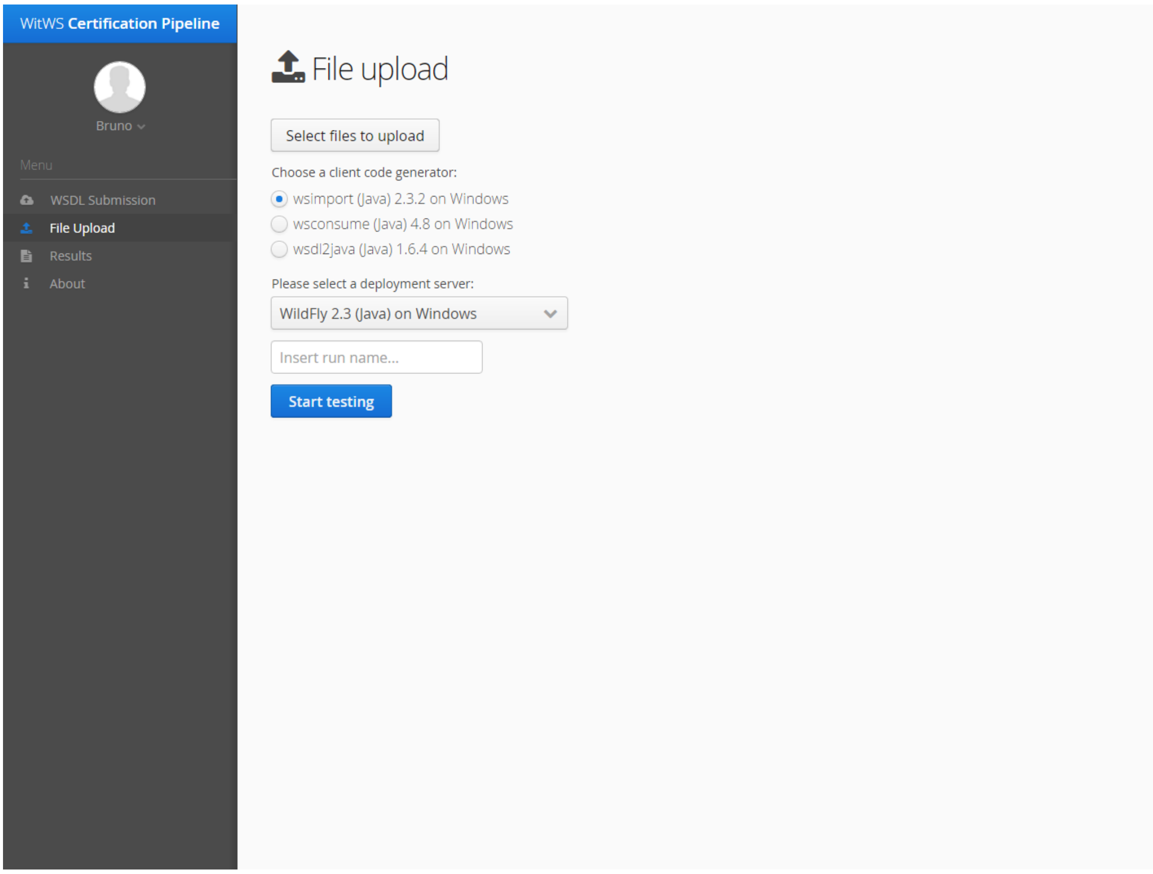
Start testing

⚠ Due to not having access to the source code, submitting a WSDL through this method will limit the amount of tests the Certification pipeline can run.

Figure 34. Submit WSDL

By clicking on the “File Upload” menu option, the user is taken to the view represented by Figure 35. When clicked, the “Select files to upload”, opens a system dialog for the user to select a web service source file along with its dependencies to upload to the application. A notification will be shown for each file uploaded. Similarly to the “WSDL Submission” feature, there’s also an option to select a client code generator and a run name must also be provided.

Besides being able to upload files, this feature is different from the one before because there’s a drop down menu for the user to select an application server to deploy the source code to.



Choose a client code generator:

● wsimport (Java) 2.3.2 on Windows
○ wsconsume (Java) 4.8 on Windows
○ wsdl2java (Java) 1.6.4 on Windows

Please select a deployment server:

WildFly 2.3 (Java) on Windows

Insert run name...

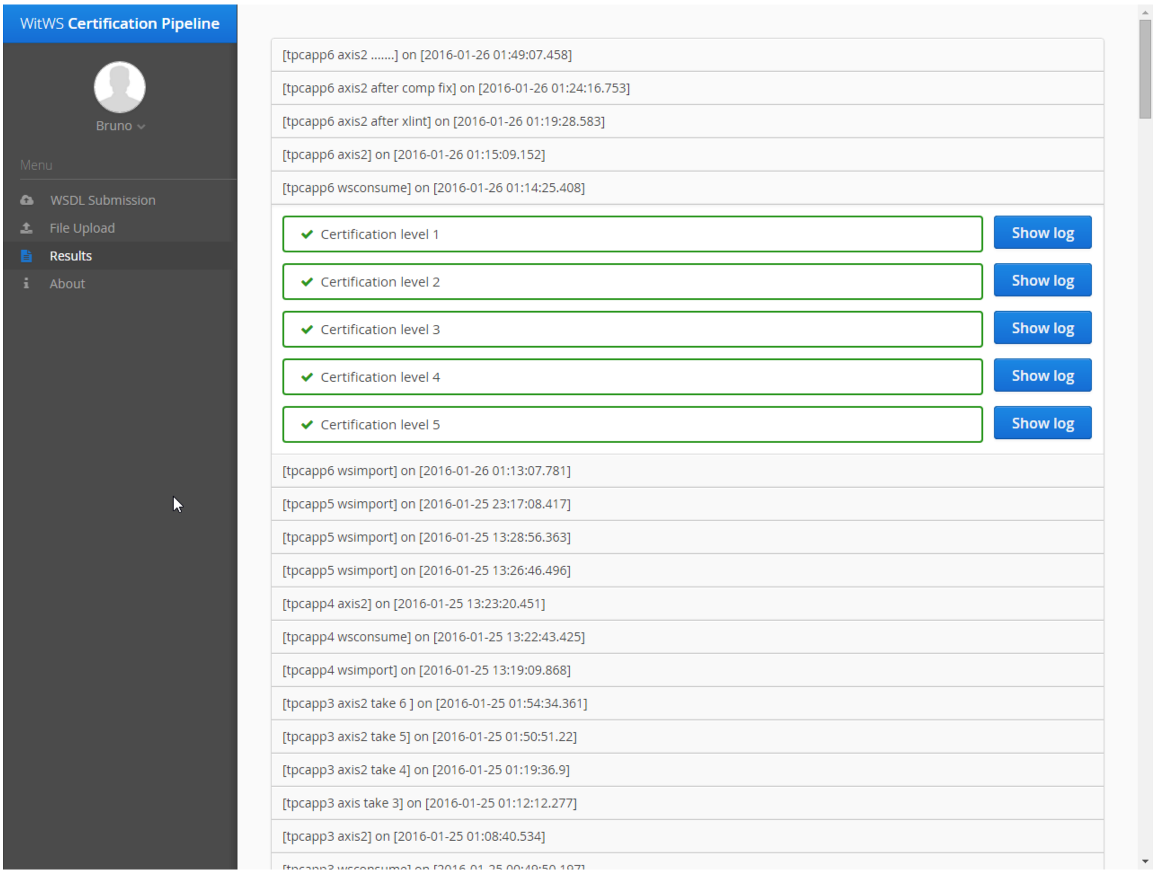
Figure 35. File upload view

Clicking the “Results” menu option on the left, takes the user to the Results page, where all his test runs are displayed - Figure 36.

Test runs in this page are displayed in an “accordion panel” style. Each panel, when collapsed, displays the name of the run followed by the timestamp of its initiation. It’s only when the user expands that that the result of each certification level is displayed.

The test runs are displayed in descending time order with the most recent run “accordion panel” opened. For a quick and easy overview, each Certification Level can be displayed in three different color (with matching icons): green (which means that the analyzed level completed without any kind of warning or errors), yellow (if the system identified a warning, but not critical, condition during the phase) or red (if an error – a condition which does not allow us to continue any further – was found).

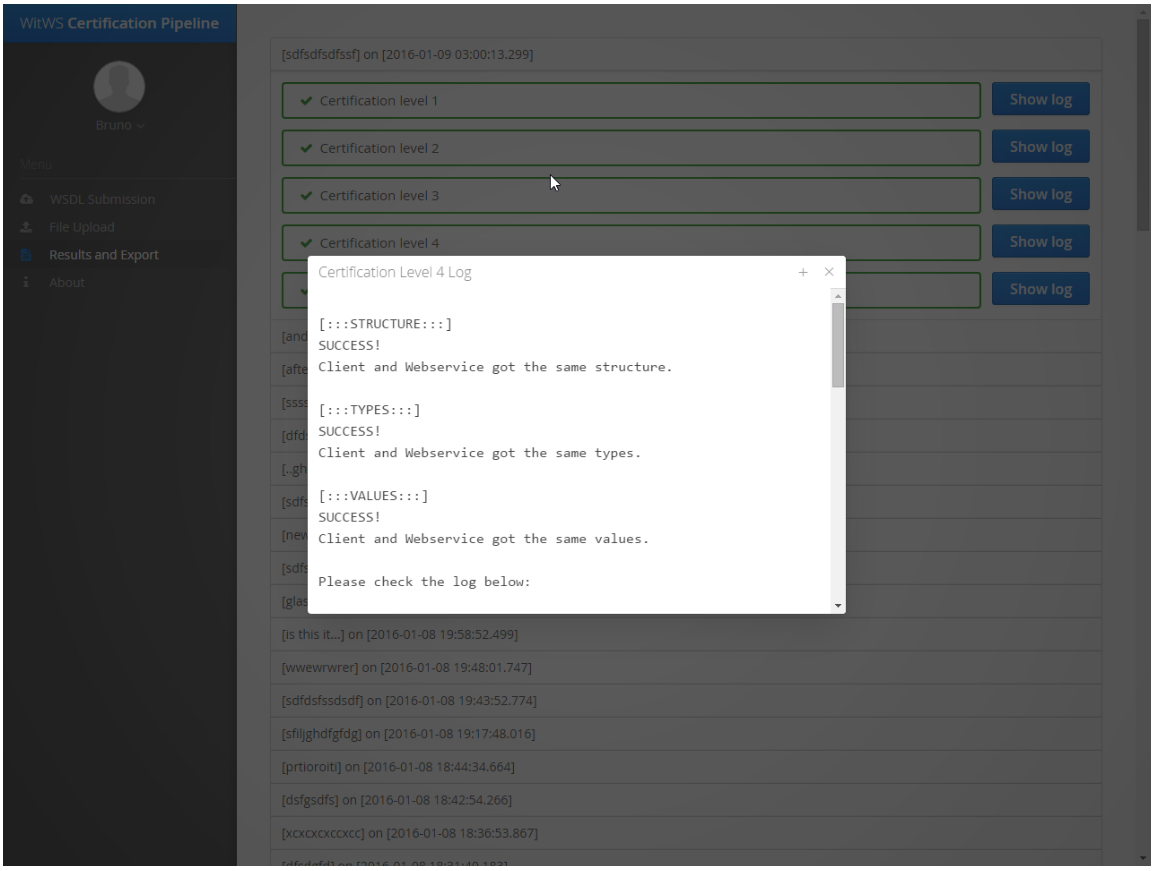
Alongside each certification level there is a “Show log” button.



[tpcapp6 axis2 ....] on [2016-01-26 01:49:07.458]
[tpcapp6 axis2 after comp fix] on [2016-01-26 01:24:16.753]
[tpcapp6 axis2 after xlint] on [2016-01-26 01:19:28.583]
[tpcapp6 axis2] on [2016-01-26 01:15:09.152]
[tpcapp6 wsconsume] on [2016-01-26 01:14:25.408]
✓ Certification level 1
✓ Certification level 2
✓ Certification level 3
✓ Certification level 4
✓ Certification level 5
[tpcapp6 wsimport] on [2016-01-26 01:13:07.781]
[tpcapp5 wsimport] on [2016-01-25 23:17:08.417]
[tpcapp5 wsimport] on [2016-01-25 13:28:56.363]
[tpcapp5 wsimport] on [2016-01-25 13:26:46.496]
[tpcapp4 axis2] on [2016-01-25 13:23:20.451]
[tpcapp4 wsconsume] on [2016-01-25 13:22:43.425]
[tpcapp4 wsimport] on [2016-01-25 13:19:09.868]
[tpcapp3 axis2 take 6] on [2016-01-25 01:54:34.361]
[tpcapp3 axis2 take 5] on [2016-01-25 01:50:51.22]
[tpcapp3 axis2 take 4] on [2016-01-25 01:19:36.9]
[tpcapp3 axis take 3] on [2016-01-25 01:12:12.277]
[tpcapp3 axis2] on [2016-01-25 01:08:40.534]
[tpcapp3 wsconsume] on [2016-01-25 00:49:50.197]

Figure 36. Results' view

When the user clicks the “Show log” button, a scrollable and expandable “modal” popup appear containing the corresponding resulting log of that phase. It can be closed by clicking on the “X” button on the top left corner or by pressing “ESC” on the keyboard. Figure 37 shows the modal.



[sdfsdfsdfssf] on [2016-01-09 03:00:13.299]

```
[:::STRUCTURE:::]
SUCCESS!
Client and Webservice got the same structure.

[:::TYPES:::]
SUCCESS!
Client and Webservice got the same types.

[:::VALUES:::]
SUCCESS!
Client and Webservice got the same values.

Please check the log below:
```

[is this it...] on [2016-01-08 19:58:52.499]

[wwewrwrer] on [2016-01-08 19:48:01.747]

[sdfdsfssdsdf] on [2016-01-08 19:43:52.774]

[sfiljghdfgfdg] on [2016-01-08 19:17:48.016]

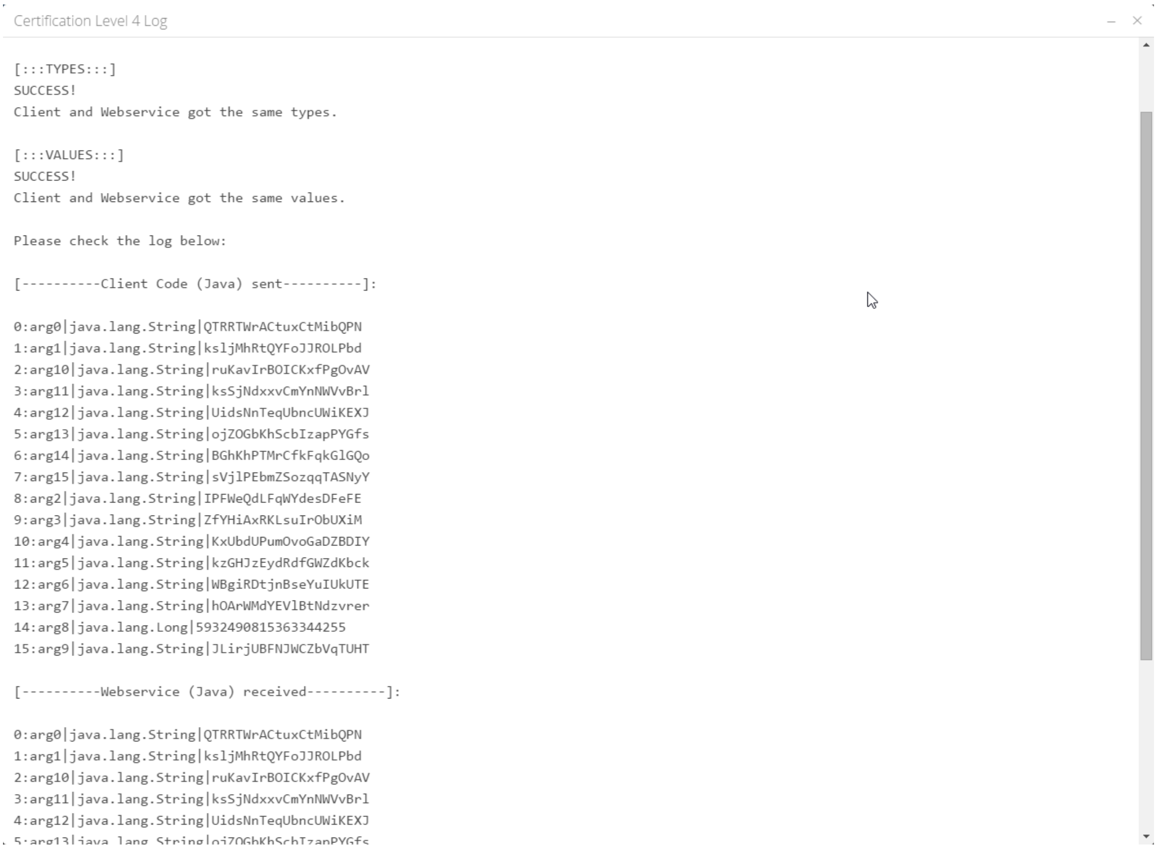
[prtioroiti] on [2016-01-08 18:44:34.664]

[dsfgsdfs] on [2016-01-08 18:42:54.266]

[xcxcxcxccxcc] on [2016-01-08 18:36:53.867]

Figure 37. Certification level's log detail

If the user clicks on the "+" button on the left of the "X" on the on the modal screen, the modal expands and takes up the whole screen state, allowing for a much easier log analysis. Figure 38 is an example of that feature.

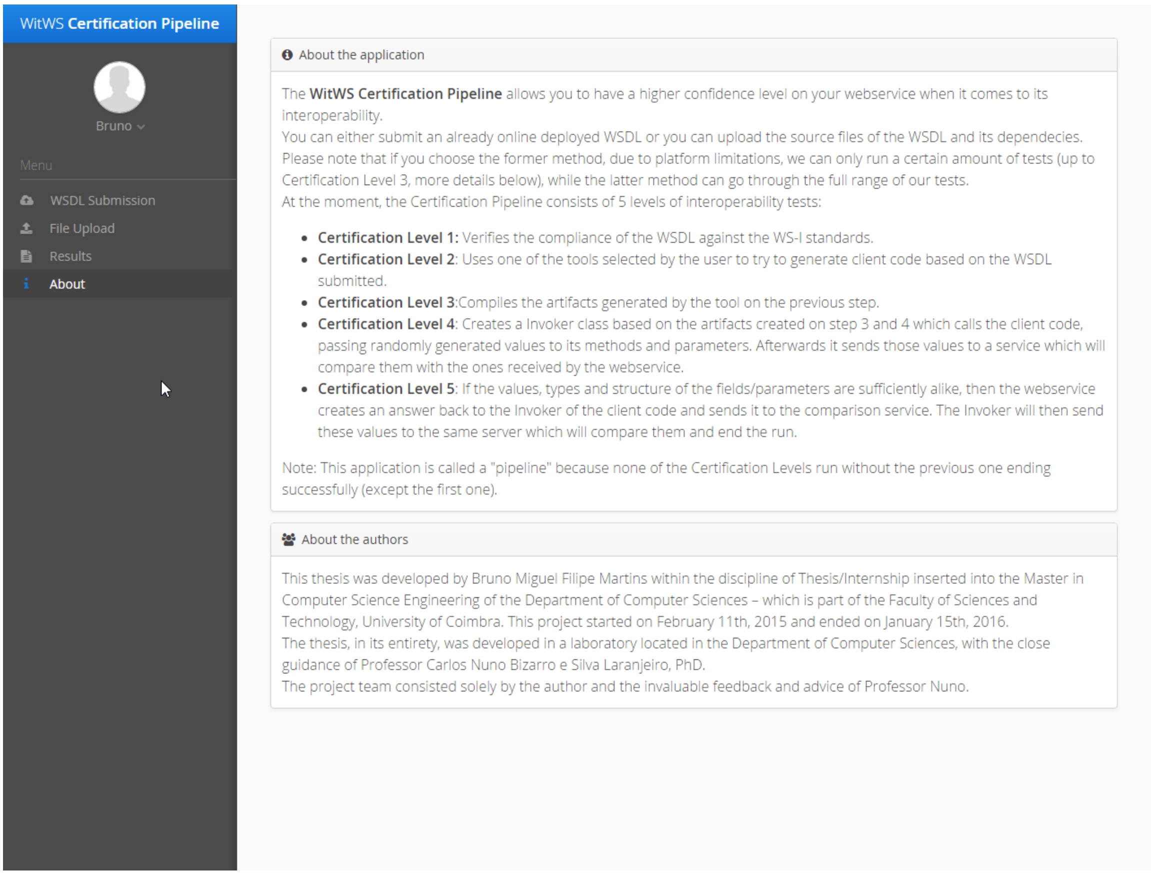


```
[:::TYPES:::]
SUCCESS!
Client and Webservice got the same types.

[:::VALUES:::]
SUCCESS!
Client and Webservice got the same values.

Please check the log below:

[----------Client Code (Java) sent----------]:

0:arg0|java.lang.String|QTRRTWrACtuxCtMibQPN
1:arg1|java.lang.String|ksljMhRtQYFoJJROLPbd
2:arg10|java.lang.String|ruKavIrBOICKxfPgOvAV
3:arg11|java.lang.String|ksSjNdxxvCmYnNWVvBrl
4:arg12|java.lang.String|UidsNnTeqUbncUWiKEXJ
5:arg13|java.lang.String|ojZOGbKhScbIzapPYGfs
6:arg14|java.lang.String|BGhKhPTMrCfkFqkGlGQo
7:arg15|java.lang.String|sVjlPEbmZSozqqTASNyY
8:arg2|java.lang.String|IPFWeQdLFqWYdesDFeFE
9:arg3|java.lang.String|ZfYHiAxRKLsuIrObUXiM
10:arg4|java.lang.String|KxUbdUPumOvoGaDZBDIY
11:arg5|java.lang.String|kzGHJzEydRdfGWZdKbck
12:arg6|java.lang.String|WBgiRDtjnBseYuIUkUTE
13:arg7|java.lang.String|hOArWMdYEVlBtNdzvrer
14:arg8|java.lang.Long|5932490815363344255
15:arg9|java.lang.String|JLirjUBFNJWCZbVqTUHT

[----------Webservice (Java) received----------]:

0:arg0|java.lang.String|QTRRTWrACtuxCtMibQPN
1:arg1|java.lang.String|ksljMhRtQYFoJJROLPbd
2:arg10|java.lang.String|ruKavIrBOICKxfPgOvAV
3:arg11|java.lang.String|ksSjNdxxvCmYnNWVvBrl
4:arg12|java.lang.String|UidsNnTeqUbncUWiKEXJ
5:arg13|java.lang.String|ojZOGbKhScbIzapPYGfs
```

Figure 38. Certification level's log detail maximized

Last, but not least, the last menu item is the "About" - Figure 39, which takes the user to the About page. It contains two sections:

- About the application: details about the application itself, how to use it and what the system is doing on the background.
- About the authors: information about the developers and contributors of the application.



About the application

The **WitWS Certification Pipeline** allows you to have a higher confidence level on your webservice when it comes to its interoperability.
You can either submit an already online deployed WSDL or you can upload the source files of the WSDL and its dependecies.
Please note that if you choose the former method, due to platform limitations, we can only run a certain amount of tests (up to Certification Level 3, more details below), while the latter method can go through the full range of our tests.
At the moment, the Certification Pipeline consists of 5 levels of interoperability tests:

- **Certification Level 1:** Verifies the compliance of the WSDL against the WS-I standards.
- **Certification Level 2**: Uses one of the tools selected by the user to try to generate client code based on the WSDL submitted.
- **Certification Level 3**:Compiles the artifacts generated by the tool on the previous step.
- **Certification Level 4**: Creates a Invoker class based on the artifacts created on step 3 and 4 which calls the client code, passing randomly generated values to its methods and parameters. Afterwards it sends those values to a service which will compare them with the ones received by the webservice.
- **Certification Level 5**: If the values, types and structure of the fields/parameters are sufficiently alike, then the webservice creates an answer back to the Invoker of the client code and sends it to the comparison service. The Invoker will then send these values to the same server which will compare them and end the run.

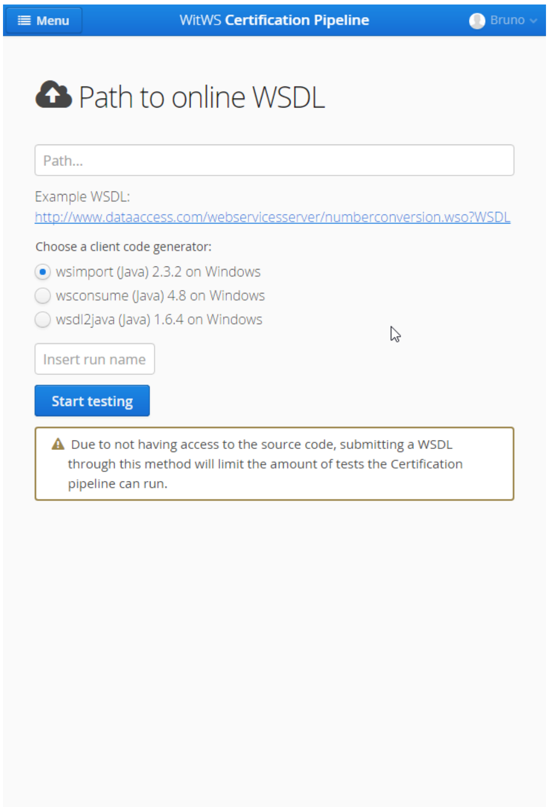
Note: This application is called a "pipeline" because none of the Certification Levels run without the previous one ending successfully (except the first one).

About the authors

This thesis was developed by Bruno Miguel Filipe Martins within the discipline of Thesis/Internship inserted into the Master in Computer Science Engineering of the Department of Computer Sciences – which is part of the Faculty of Sciences and Technology, University of Coimbra. This project started on February 11th, 2015 and ended on January 15th, 2016.
The thesis, in its entirety, was developed in a laboratory located in the Department of Computer Sciences, with the close guidance of Professor Carlos Nuno Bizarro e Silva Laranjeiro, PhD.
The project team consisted solely by the author and the invaluable feedback and advice of Professor Nuno.

Figure 39. About view

As a bonus, the application is responsive, so it adapts to several screen sizes, as we can see in Figure 40 and Figure 41.



Path to online WSDL

Path...

Example WSDL:
http://www.dataaccess.com/webservicesserver/numberconversion.wso?WSDL

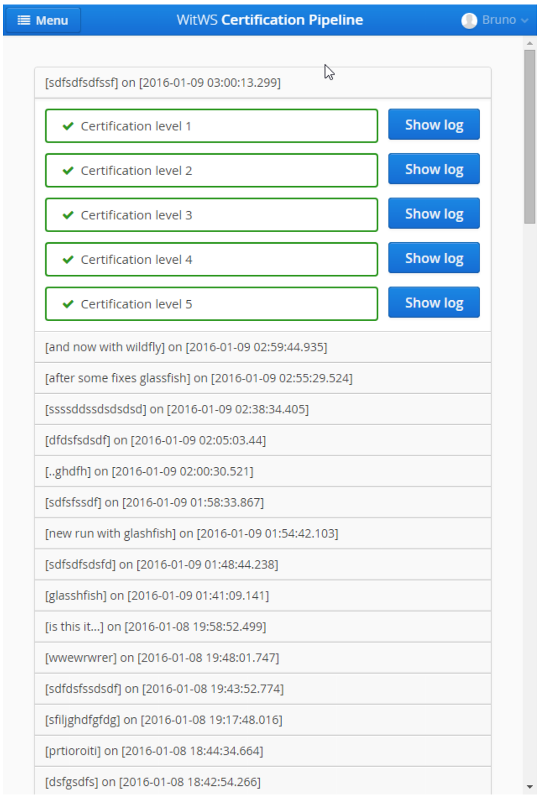
Choose a client code generator:

● wsimport (Java) 2.3.2 on Windows
○ wsconsume (Java) 4.8 on Windows
○ wsdl2java (Java) 1.6.4 on Windows

Insert run name

Due to not having access to the source code, submitting a WSDL through this method will limit the amount of tests the Certification pipeline can run.

Figure 40. Submit WSDL - smaller screen size



[sdfsdfsdfssf] on [2016-01-09 03:00:13.299]

✓ Certification level 1	Show log
✓ Certification level 2	Show log
✓ Certification level 3	Show log
✓ Certification level 4	Show log
✓ Certification level 5	Show log

[and now with wildfly] on [2016-01-09 02:59:44.935]

[after some fixes glassfish] on [2016-01-09 02:55:29.524]

[ssssddssdsdsdsd] on [2016-01-09 02:38:34.405]

[dfdsfsdsdf] on [2016-01-09 02:05:03.44]

[..ghdfh] on [2016-01-09 02:00:30.521]

[sdfsfssdf] on [2016-01-09 01:58:33.867]

[new run with glashfish] on [2016-01-09 01:54:42.103]

[sdfsdfsdsfd] on [2016-01-09 01:48:44.238]

[glasshfish] on [2016-01-09 01:41:09.141]

[is this it...] on [2016-01-08 19:58:52.499]

[wwewrwrer] on [2016-01-08 19:48:01.747]

[sdfdsfssdsdf] on [2016-01-08 19:43:52.774]

[sfiljghdfgfdg] on [2016-01-08 19:17:48.016]

[prtioroiti] on [2016-01-08 18:44:34.664]

[dsfgsdfs] on [2016-01-08 18:42:54.266]

Figure 41. Results' view - smaller screen size

Chapter 11
Conclusion

In this thesis, WitWS, a web-based testing tool that allows testing a web service against multiple client platforms, was presented. The tool can be used without the presence of the code of the service being tested (i.e., it only requires a WSDL file to perform tests), although it can perform extended communication tests if the service source code (i.e., the service interface code) is provided. WitWS was used to test 4 sets of services deployed on very popular implementations of the web services stack – Metro, the JAX-WS reference implementation on Glassfish, and JBossWS on the WildFly server. Tests were run against the client-side implementation of Metro, JBossWS, and also against Axis2. The problems disclosed during the experiments, including problems introduced by our custom fault injector, served to illustrated the utility of our testing service and its problem detection capabilities. Without this type of testing, many of these problems usually pass unaware to developers, only to be found at runtime, when a particular client interacts with the service.

A research paper was also submitted to the International Symposium on Software Testing and Analysis conference, which takes place in Saarland University, Saarbrücken, Germany, in July 18–20, 2016

Experience acquired

The implementation required a solid knowledge about programming an application for the web and the ability to use, configure and manage several different tools, frameworks and libraries. The author never had contact with any of them until the start of the thesis.

Vaadin, Spring, PODAM and Velocity were, without a doubt, the most challenging ones, with Vaadin and Velocity having a really steep learning curve.

The author also had very little experience with web services until now, so this was an excellent opportunity to learn more about the field and its quirks.

Future Work

The modularity of the system allows to extend its functionality, so most of the tasks related to the future work were already discussed in section Chapter 9
Application Limitations and Extensibility.
Besides that, the following bullet points represent some of the possible future tasks:

- When uploading source code or submitting a WSDL, the user could have the option to select the method he wants to test;
- Have the system work with other protocols beyond SOAP.
- It would be very useful to add an "Export results" option to each test run. Unfortunately, there was no time to implement that feature;
- An option to delete runs could be added;
- A "Remember Me" option on the login screen would be interesting to have. That way the user wouldn't need to login every time his session times out.
- The "File upload" view could use a facelift. The interface looks a bit empty.

References

[1] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI," *IEEE Internet Comput.*, vol. 6, no. 2, pp. 86–93, Mar. 2002.

[2] E. Hewitt, *Java SOA Cookbook*. O'Reilly Media, Inc., 2009.

[3] "Web Services Interoperability Organization (WS-I)." [Online]. Available: http://www.ws-i.org/. [Accessed: 26-Mar-2015].

[4] Web Services Interoperabily Organization (WS-I), "Deliverables - Basic Profile Working Group," 2014. [Online]. Available: http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile. [Accessed: 13-Jan-2014].

[5] I. A. Elia, N. Laranjeiro, and M. Vieira, "Test-based Interoperability Certification for Web Services," in *The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*, Rio de Janeiro, Brazil, 2015.

[6] I. A. Elia, N. Laranjeiro, and M. Vieira, "A Field Perspective on the Interoperability of Web Services," in *11th IEEE International Conference on Services Computing (SCC 2014)*, Anchorage, Alaska, USA, 2014.

[7] I. A. Elia, N. Laranjeiro, and M. Vieira, "Understanding Interoperability Issues of Web Service Frameworks," in *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, Atlanta, Georgia, USA, 2014.

[8] "CORE Conference Portal." [Online]. Available: http://portal.core.edu.au/conf-ranks/. [Accessed: 29-Jan-2016].

[9] B. Martins, N. Laranjeiro, and M. Vieira, "INTENSE: INteroperability TEstiNg as a SErvice," in *The International Symposium on Software Testing and Analysis*, Saarbrücken, Germany, July 18–20, 2016.

[10] E. Cerami, *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc., 2002.

[11] "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)." [Online]. Available: http://www.w3.org/TR/soap12/. [Accessed: 27-Mar-2015].

[12] "Enterprise Java Contract-First vs. Contract-Last Web Services." [Online]. Available: http://www.developer.com/design/article.php/3745701/Enterprise-Java-Contract-First-vs-Contract-Last-Web-Services.htm. [Accessed: 27-Mar-2015].

[13] C. Ferris, A. Karmarkar, P. Yendluri, K. Ballinger, D. Ehnebuske, M. Gudgin, C. Liu, and M. Nottingham, "WS-I Basic Profile - Version 1.2," 24-Oct-2007. [Online]. Available: http://www.ws-i.org/Profiles/BasicProfile-1_2(WGAD).html. [Accessed: 14-Feb-2008].

[14] "Basic Profile - Version 1.1 (Final)." [Online]. Available: http://www.ws-i.org/profiles/basicprofile-1.1.html. [Accessed: 27-Mar-2015].

[15] "Deliverables - Basic Profile." [Online]. Available: http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile. [Accessed: 27-Mar-2015].

[16] S. Liu, T. Maibaum, and K. Araki, *Formal Methods and Software Engineering: 10th International Conference on Formal Engineering Methods ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008, Proceedings*. Springer Science & Business Media, 2008.

[17] "SoapUI - The Home of Functional Testing." [Online]. Available: http://www.soapui.org/. [Accessed: 26-Mar-2015].

[18] "Jolt Awards 2014: The Best Testing Tools," *Dr. Dobb's*. [Online]. Available: http://www.drdobbs.com/testing/jolt-awards-2014-the-best-testing-tools/240168372. [Accessed: 29-Mar-2015].

[19] "SYS-CON SOA World Reader's Choice Award." [Online]. Available: http://soa.sys-con.com/node/397933. [Accessed: 29-Mar-2015].

[20] "ATI Automation Awards: Best Open Source Performance Automated Test Tool." [Online]. Available: http://smartbear.com/news/news-releases/smartbear-collects-5-ati-automation-awards-at-test/. [Accessed: 29-Mar-2015].

[21] "Working with WSDLs | SOAP and WSDL." [Online]. Available: http://www.soapui.org/soap-and-wsdl/working-with-wsdls.html#2-Validating-the-WSDL-against-the-WS-I-Basic-Profile. [Accessed: 29-Mar-2015].

[22] "SOAPSonar - SOA Service Testing and Diagnostics." [Online]. Available: http://www.crosschecknet.com/products/soapsonar.php. [Accessed: 26-Mar-2015].

[23] "WebInject - (HTTP) Web Application and Web Services Test Tool." [Online]. Available: http://www.webinject.org/. [Accessed: 27-Mar-2015].

[24] "Apache JMeter - Apache JMeter^{TM}." [Online]. Available: http://jmeter.apache.org/. [Accessed: 27-Mar-2015].

[25] "Storm," *CodePlex*. [Online]. Available: https://storm.codeplex.com/Wikipage?ProjectName=storm. [Accessed: 27-Mar-2015].

[26] "wizdl - Web Service GUI Test Tool," *CodePlex*. [Online]. Available: http://wizdl.codeplex.com/Wikipage?ProjectName=wizdl. [Accessed: 27-Mar-2015].

[27] W. T. Tsai, R. Paul, W. Song, and Z. Cao, "Coyote: An XML-Based Framework for Web Services Testing," in *Ninth IEEE International Symposium on High-Assurance Systems Engineering (HASE'05)*, Los Alamitos, CA, USA, 2002, vol. 0, p. 173.

[28] Y. Bassil, "Distributed, Cross-Platform, and Regression Testing Architecture for Service-Oriented Architecture," *ArXiv12035403 Cs*, Mar. 2012.

[29] K. Wiegers and J. Beatty, *Software Requirements*. Pearson Education, 2013.

[30] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[31] K. Schwaber, "SCRUM Development Process," in *Business Object Design and Implementation*, D. J. Sutherland, C. Casanave, J. Miller, D. P. Patel, and G. Hollowell, Eds. Springer London, 1997, pp. 117–134.

[32] "Intro to Agile - Agile For All." [Online]. Available: http://www.agileforall.com/intro-to-agile/. [Accessed: 20-Jan-2016].

[33] I. Sommerville, *Software Engineering*, 9th ed. USA: Addison-Wesley Publishing Company, 2010.

[34] "Jolt Awards: Coding Tools," *Dr. Dobb's*. [Online]. Available: http://www.drdobbs.com/joltawards/jolt-awards-coding-tools/240165725. [Accessed: 12-Apr-2015].

[35] "Undertow · JBoss Community." [Online]. Available: http://undertow.io/. [Accessed: 24-May-2015].

[36] "Maven." [Online]. Available: https://maven.apache.org/. [Accessed: 05-Apr-2015].

[37] "GitLab." [Online]. Available: https://git.dei.uc.pt/users/sign_in. [Accessed: 05-Jul-2015].

[38] "Java Web Frameworks Comparison | zeroturnaround.com - Part 11," *ZeroTurnaround*. [Online]. Available: http://zeroturnaround.com/rebellabs/the-curious-coders-java-web-frameworks-comparison-spring-mvc-grails-vaadin-gwt-wicket-play-struts-and-jsf/11/. [Accessed: 15-Apr-2015].

[39] "Technology Radar Digital Edition | ThoughtWorks." [Online]. Available: http://www.thoughtworks.com/radar/languages-and-frameworks. [Accessed: 15-Apr-2015].

[40] "Raible Designs | Presentations." [Online]. Available: http://raibledesigns.com/rd/page/publications. [Accessed: 15-Apr-2015].

[41] "DevRates | Web Frameworks." [Online]. Available: http://devrates.com/project/list?query=%5Bweb+framework%5D. [Accessed: 15-Apr-2015].

[42] "WWW: World Wide Wait - Devoxx Edition," *prezi.com*. [Online]. Available: https://prezi.com/dr3on1qcajzw/www-world-wide-wait-devoxx-edition/. [Accessed: 20-Apr-2015].

[43] "Devoxx," *Wikipedia, the free encyclopedia*. 13-Apr-2015.

[44] "AppFuse - Dev - Creating a new archetype." [Online]. Available: http://appfuse.547863.n4.nabble.com/Creating-a-new-archetype-td4656359.html. [Accessed: 16-Apr-2015].

[45] mraible, "mraible/appfuse-loc.md," *GitHub Gists*. [Online]. Available: https://gist.github.com/mraible/5033218. [Accessed: 15-Apr-2015].

[46] "gwt-best-practices-soup." [Online]. Available: https://code.google.com/p/gwt-best-practices-soup/. [Accessed: 17-Apr-2015].

[47] R. Lämmel and S. P. Jones, "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming," in *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, New York, NY, USA, 2003, pp. 26–37.

[48] "Advanced Java Class Tutorial: A Guide to Class Reloading," *Toptal Engineering Blog*. [Online]. Available: http://www.toptal.com/java/java-wizardry-101-a-guide-to-java-class-reloading. [Accessed: 29-Jan-2016].

[49] "TPC-App - Homepage." [Online]. Available: http://www.tpc.org/tpc_app/. [Accessed: 26-Jan-2016].

[50] "PODAM (POjo DAta Mocker)." [Online]. Available: http://mtedone.github.io/podam/. [Accessed: 28-Jan-2016].

[51] "Utilities for streaming data over RMI." [Online]. Available: http://openhms.sourceforge.net/rmiio/. [Accessed: 17-Jan-2016].

[52] "WitWS Git Repository | GitLab." [Online]. Available: https://git.dei.uc.pt/bmfm/witws. [Accessed: 28-Jan-2016].

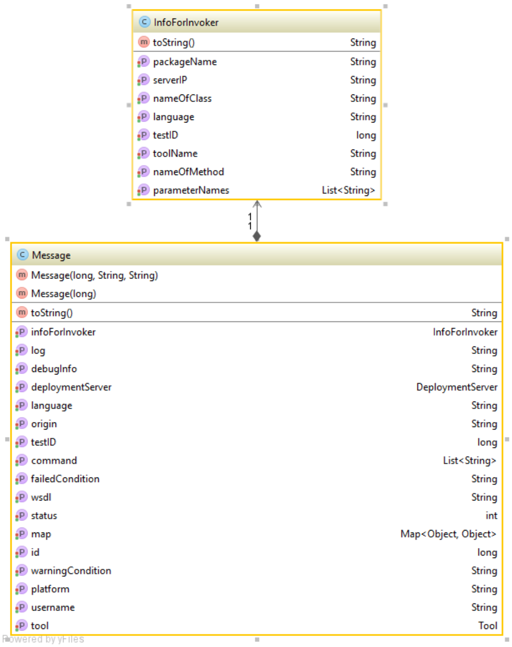
[53] E. Jendrock, R. Cervera-Navarro, I. Evan, K. Haase, and W. Markito, "Java EE Tutorial." Sep-2014.

Annexes

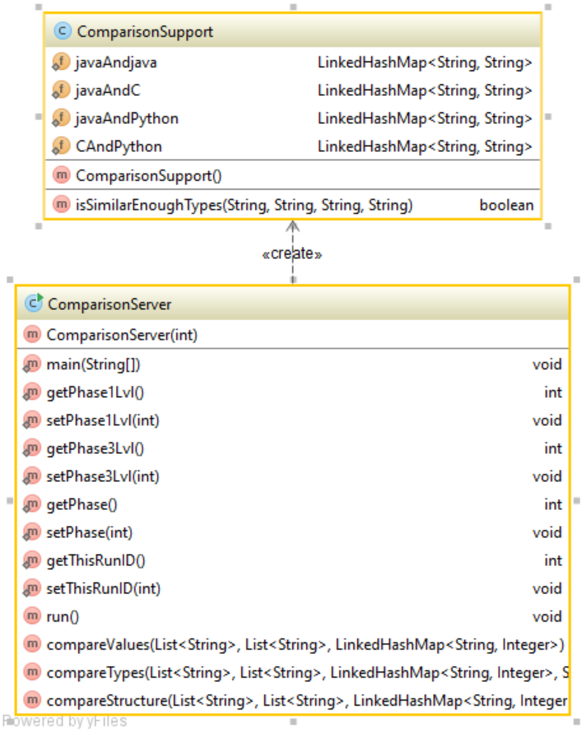
A1. Class diagrams

In this section the class diagrams of the application will be presented, divided by package.

A1.1. Communication package

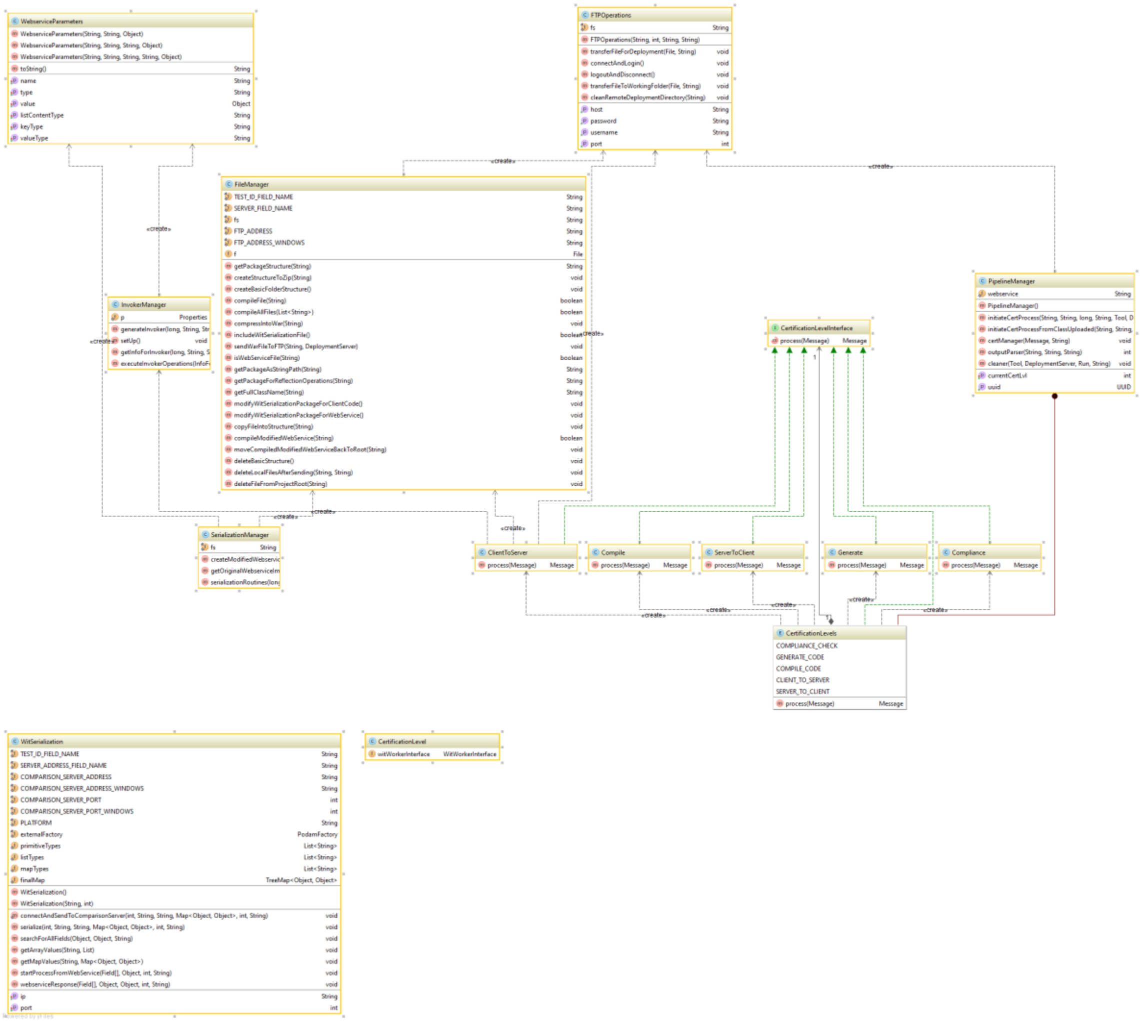


A1.2. Comparison package

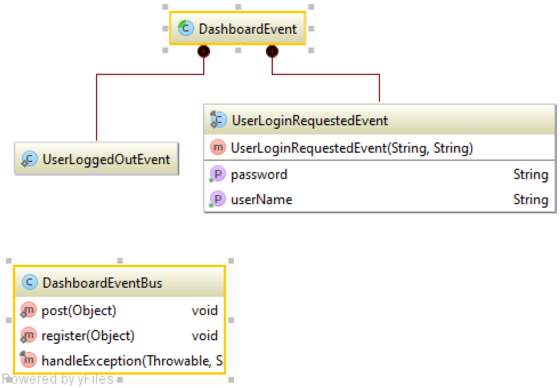

ComparisonSupport
javaAndjava LinkedHashMap<String, String>
javaAndC LinkedHashMap<String, String>
javaAndPython LinkedHashMap<String, String>
CAndPython LinkedHashMap<String, String>
ComparisonSupport()
isSimilarEnoughTypes(String, String, String, String) boolean
«create»
ComparisonServer
ComparisonServer(int)
main(String[]) void
getPhase1Lvl() int
setPhase1Lvl(int) void
getPhase3Lvl() int
setPhase3Lvl(int) void
getPhase() int
setPhase(int) void
getThisRunID() int
setThisRunID(int) void
run() void
compareValues(List<String>, List<String>, LinkedHashMap<String, Integer>)
compareTypes(List<String>, List<String>, LinkedHashMap<String, Integer>, S
compareStructure(List<String>, List<String>, LinkedHashMap<String, Integer
Powered by yFiles

A1.3. Controllers package

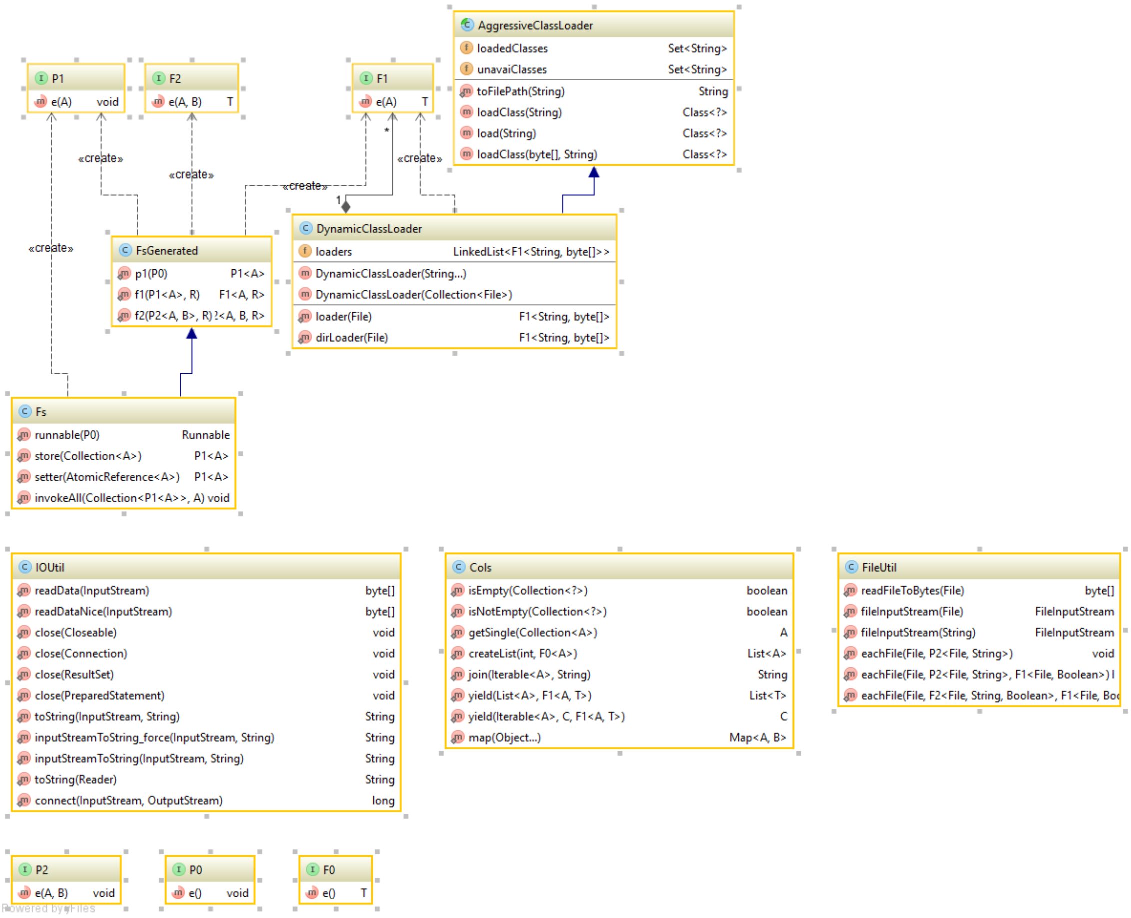
For a better look the this diagram, please go to: https://goo.gl/rVX0ss


WebserviceParameters
WebserviceParameters(String, String, Object)
WebserviceParameters(String, String, String, Object)
WebserviceParameters(String, String, String, String, Object)
toString() String
name String
type String
value Object
listContentType String
keyType String
valueType String
FTPOperations
fs String
FTPOperations(String, int, String, String)
transferFileForDeployment(File, String) void
connectAndLogin() void
logoutAndDisconnect() void
transferFileToWorkingFolder(File, String) void
cleanRemoteDeploymentDirectory(String) void
host String
password String
username String
port int
«create»
FileManager
TEST_ID_FIELD_NAME String
SERVER_FIELD_NAME String
fs String
FTP_ADDRESS String
FTP_ADDRESS_WINDOWS String
f File
getPackageStructure(String) String
createStructureToZip(String) void
createBasicFolderStructure() void
compileFile(String) boolean
compileAllFiles(List<String>) boolean
compressIntoWar(String) void
includeWitSerializationFile() boolean
sendWarFileToFTP(String, DeploymentServer) void
isWebServiceFile(String) boolean
getPackageAsStringPath(String) String
getPackageForReflectionOperations(String) String
getFullClassName(String) String
modifyWitSerializationPackageForClientCode() void
modifyWitSerializationPackageForWebService() void
copyFileIntoStructure(String) void
compileModifiedWebService(String) boolean
moveCompiledModifiedWebServiceBackToRoot(String) void
deleteBasicStructure() void
deleteLocalFilesAfterSending(String, String) void
deleteFileFromProjectRoot(String) void
InvokerManager
p Properties
generateInvoker(long, String, Str
setUp() void
getInfoForInvoker(long, String, S
executeInvokerOperations(InfoF
SerializationManager
fs String
createModifiedWebservic
getOriginalWebserviceIm
serializationRoutines(long
PipelineManager
webservice String
PipelineManager()
initiateCertProcess(String, String, long, String, Tool, D
initiateCertProcessFromClassUploaded(String, String,
certManager(Message, String) void
outputParser(String, String, String) int
cleaner(Tool, DeploymentServer, Run, String) void
currentCertLvl int
uuid UUID
CertificationLevelInterface
process(Message) Message
1
ClientToServer
process(Message) Message
Compile
process(Message) Message
ServerToClient
process(Message) Message
Generate
process(Message) Message
Compliance
process(Message) Message
CertificationLevels
COMPLIANCE_CHECK
GENERATE_CODE
COMPILE_CODE
CLIENT_TO_SERVER
SERVER_TO_CLIENT
process(Message) Message
WitSerialization
TEST_ID_FIELD_NAME String
SERVER_ADDRESS_FIELD_NAME String
COMPARISON_SERVER_ADDRESS String
COMPARISON_SERVER_ADDRESS_WINDOWS String
COMPARISON_SERVER_PORT int
COMPARISON_SERVER_PORT_WINDOWS int
PLATFORM String
externalFactory PodamFactory
primitiveTypes List<String>
listTypes List<String>
mapTypes List<String>
finalMap TreeMap<Object, Object>
WitSerialization()
WitSerialization(String, int)
connectAndSendToComparisonServer(int, String, String, Map<Object, Object>, int, String) void
serialize(int, String, String, Map<Object, Object>, int, String) void
searchForAllFields(Object, Object, String) void
getArrayValues(String, List) void
getMapValues(String, Map<Object, Object>) void
startProcessFromWebService(Field[], Object, int, String) void
webserviceResponse(Field[], Object, Object, int, String) void
ip String
port int
CertificationLevel
witWorkerInterface WitWorkerInterface
Powered by yFiles

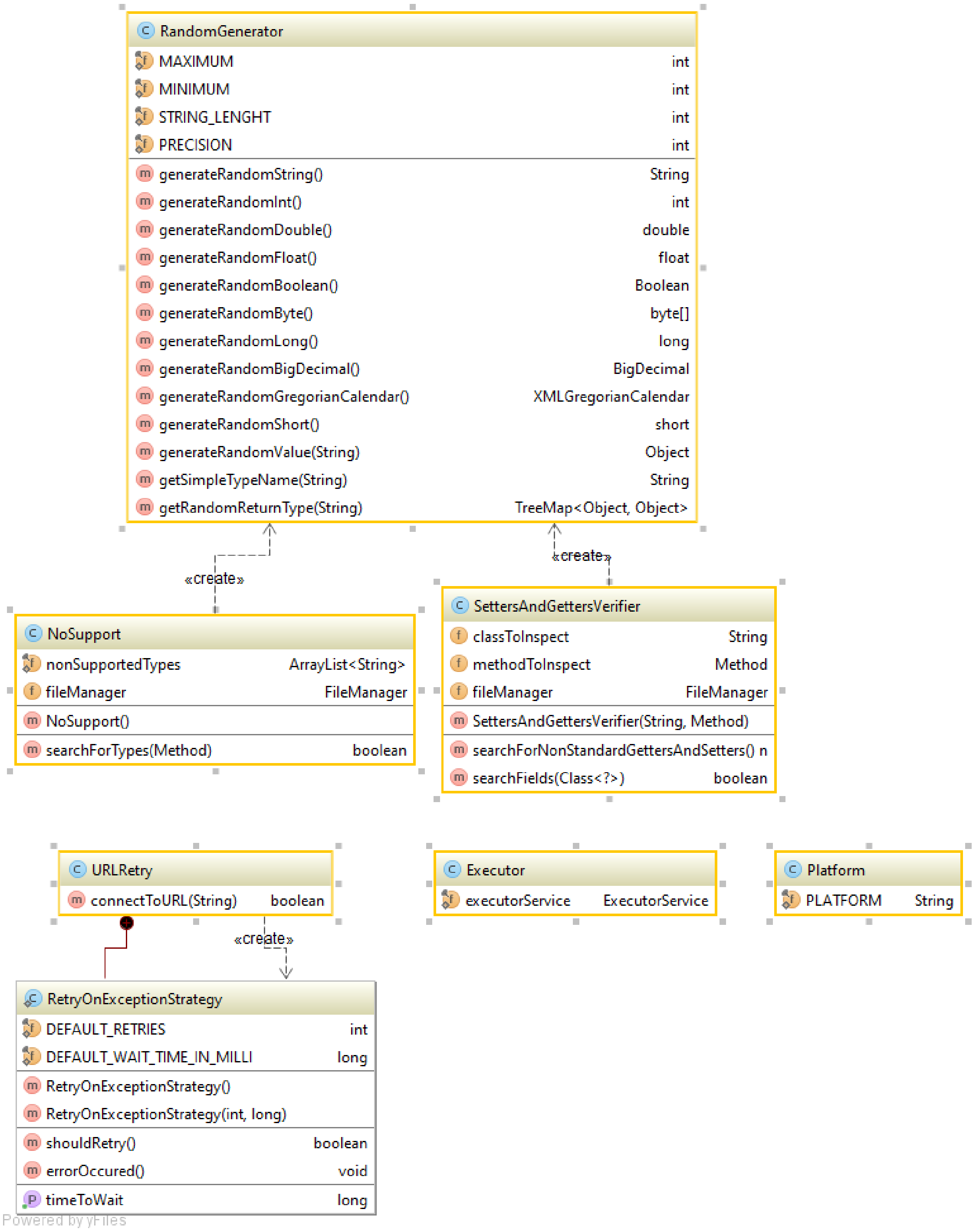
A1.4. Event package



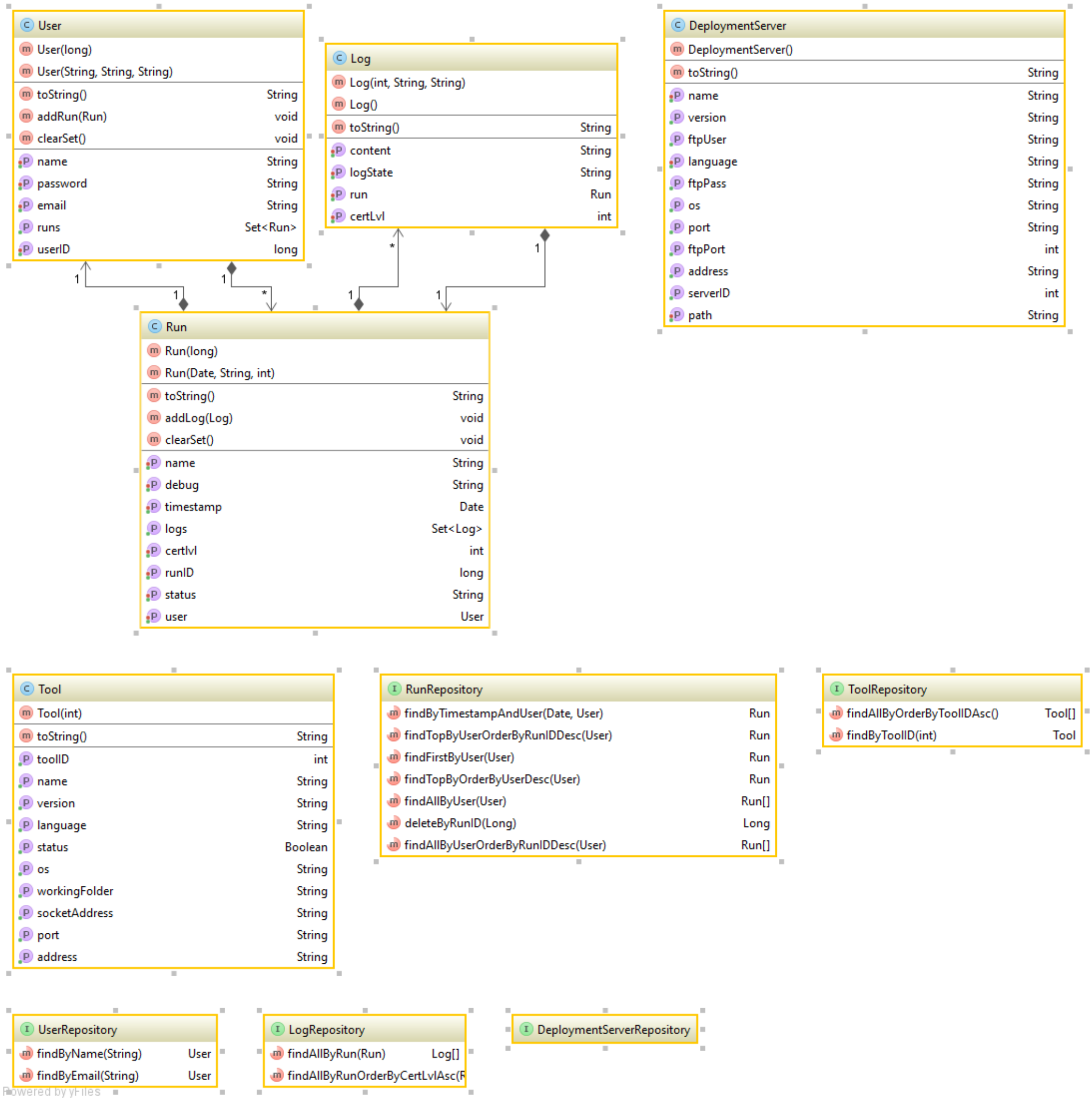
A1.5 Loader package


AggressiveClassLoader
loadedClasses Set<String>
unavaiClasses Set<String>
toFilePath(String) String
loadClass(String) Class<?>
load(String) Class<?>
loadClass(byte[], String) Class<?>
P1
e(A) void
F2
e(A, B) T
F1
e(A) T
«create»
«create»
«create»
«create»
«create»
*
1
DynamicClassLoader
loaders LinkedList<F1<String, byte[]>>
DynamicClassLoader(String...)
DynamicClassLoader(Collection<File>)
loader(File) F1<String, byte[]>
dirLoader(File) F1<String, byte[]>
FsGenerated
p1(P0) P1<A>
f1(P1<A>, R) F1<A, R>
f2(P2<A, B>, R) ?<A, B, R>
Fs
runnable(P0) Runnable
store(Collection<A>) P1<A>
setter(AtomicReference<A>) P1<A>
invokeAll(Collection<P1<A>>, A) void
IOUtil
readData(InputStream) byte[]
readDataNice(InputStream) byte[]
close(Closeable) void
close(Connection) void
close(ResultSet) void
close(PreparedStatement) void
toString(InputStream, String) String
inputStreamToString_force(InputStream, String) String
inputStreamToString(InputStream, String) String
toString(Reader) String
connect(InputStream, OutputStream) long
Cols
isEmpty(Collection<?>) boolean
isNotEmpty(Collection<?>) boolean
getSingle(Collection<A>) A
createList(int, F0<A>) List<A>
join(Iterable<A>, String) String
yield(List<A>, F1<A, T>) List<T>
yield(Iterable<A>, C, F1<A, T>) C
map(Object...) Map<A, B>
FileUtil
readFileToBytes(File) byte[]
fileInputStream(File) FileInputStream
fileInputStream(String) FileInputStream
eachFile(File, P2<File, String>) void
eachFile(File, P2<File, String>, F1<File, Boolean>) I
eachFile(File, F2<File, String, Boolean>, F1<File, Boo
P2
e(A, B) void
P0
e() void
F0
e() T
Powered by yFiles

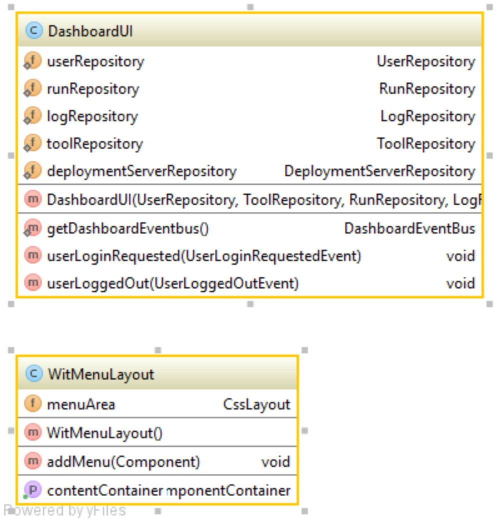
A1.6. Misc package


RandomGenerator
MAXIMUM int
MINIMUM int
STRING_LENGHT int
PRECISION int
generateRandomString() String
generateRandomInt() int
generateRandomDouble() double
generateRandomFloat() float
generateRandomBoolean() Boolean
generateRandomByte() byte[]
generateRandomLong() long
generateRandomBigDecimal() BigDecimal
generateRandomGregorianCalendar() XMLGregorianCalendar
generateRandomShort() short
generateRandomValue(String) Object
getSimpleTypeName(String) String
getRandomReturnType(String) TreeMap<Object, Object>
«create»
«create»
NoSupport
nonSupportedTypes ArrayList<String>
fileManager FileManager
NoSupport()
searchForTypes(Method) boolean
SettersAndGettersVerifier
classToInspect String
methodToInspect Method
fileManager FileManager
SettersAndGettersVerifier(String, Method)
searchForNonStandardGettersAndSetters() n
searchFields(Class<?>) boolean
URLRetry
connectToURL(String) boolean
«create»
Executor
executorService ExecutorService
Platform
PLATFORM String
RetryOnExceptionStrategy
DEFAULT_RETRIES int
DEFAULT_WAIT_TIME_IN_MILLI long
RetryOnExceptionStrategy()
RetryOnExceptionStrategy(int, long)
shouldRetry() boolean
errorOccured() void
timeToWait long
Powered by yFiles

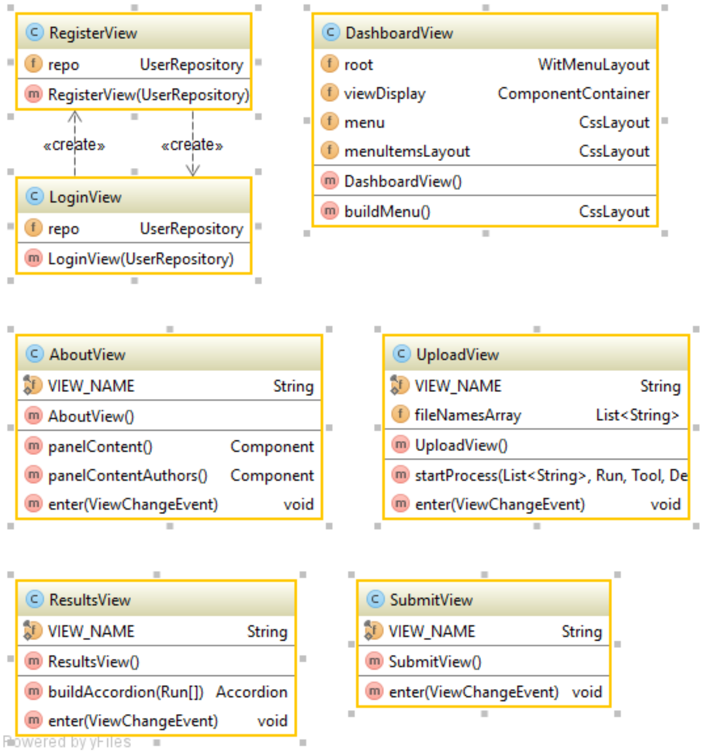
A1.7. Models package



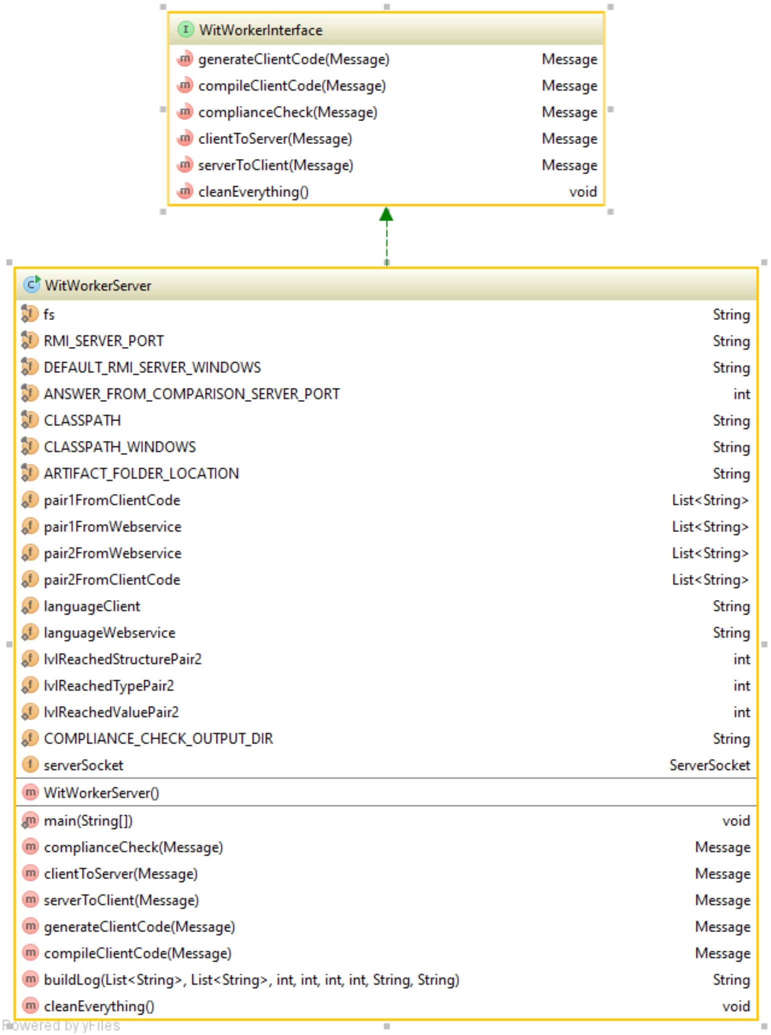
A1.8. UI package


DashboardUI
userRepository UserRepository
runRepository RunRepository
logRepository LogRepository
toolRepository ToolRepository
deploymentServerRepository DeploymentServerRepository
DashboardUI(UserRepository, ToolRepository, RunRepository, LogR
getDashboardEventbus() DashboardEventBus
userLoginRequested(UserLoginRequestedEvent) void
userLoggedOut(UserLoggedOutEvent) void
WitMenuLayout
menuArea CssLayout
WitMenuLayout()
addMenu(Component) void
contentContainer mponentContainer
Powered by yFiles

A1.9. Views package



A1.10. Witworker package

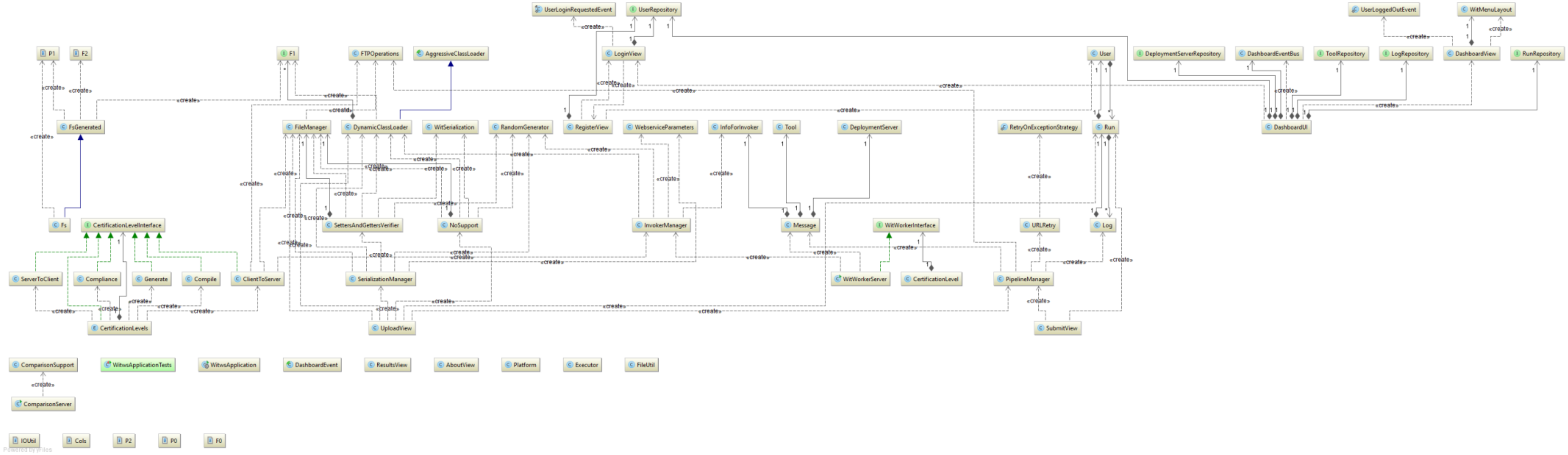


WitWorkerInterface	
generateClientCode(Message)	Message
compileClientCode(Message)	Message
complianceCheck(Message)	Message
clientToServer(Message)	Message
serverToClient(Message)	Message
cleanEverything()	void

WitWorkerServer	
fs	String
RMI_SERVER_PORT	String
DEFAULT_RMI_SERVER_WINDOWS	String
ANSWER_FROM_COMPARISON_SERVER_PORT	int
CLASSPATH	String
CLASSPATH_WINDOWS	String
ARTIFACT_FOLDER_LOCATION	String
pair1FromClientCode	List<String>
pair1FromWebservice	List<String>
pair2FromWebservice	List<String>
pair2FromClientCode	List<String>
languageClient	String
languageWebservice	String
lvlReachedStructurePair2	int
lvlReachedTypePair2	int
lvlReachedValuePair2	int
COMPLIANCE_CHECK_OUTPUT_DIR	String
serverSocket	ServerSocket
WitWorkerServer()	
main(String[])	void
complianceCheck(Message)	Message
clientToServer(Message)	Message
serverToClient(Message)	Message
generateClientCode(Message)	Message
compileClientCode(Message)	Message
buildLog(List<String>, List<String>, int, int, int, int, String, String)	String
cleanEverything()	void

A1.11. Overview

For a better look at the overview diagram, please go to: https://goo.gl/ByZp3E

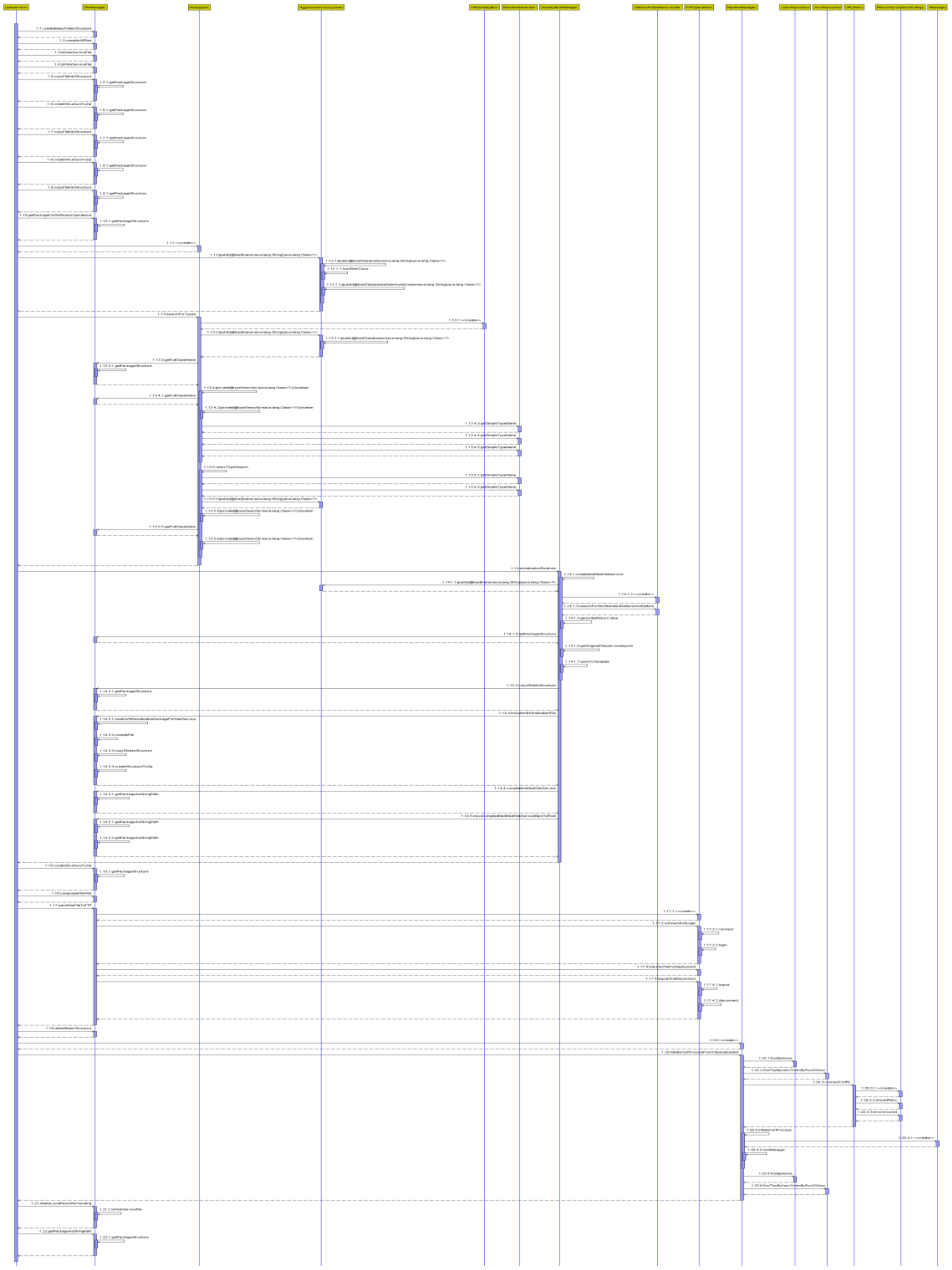


B1. Sequence diagrams

Relevant sequence diagrams are displayed here.

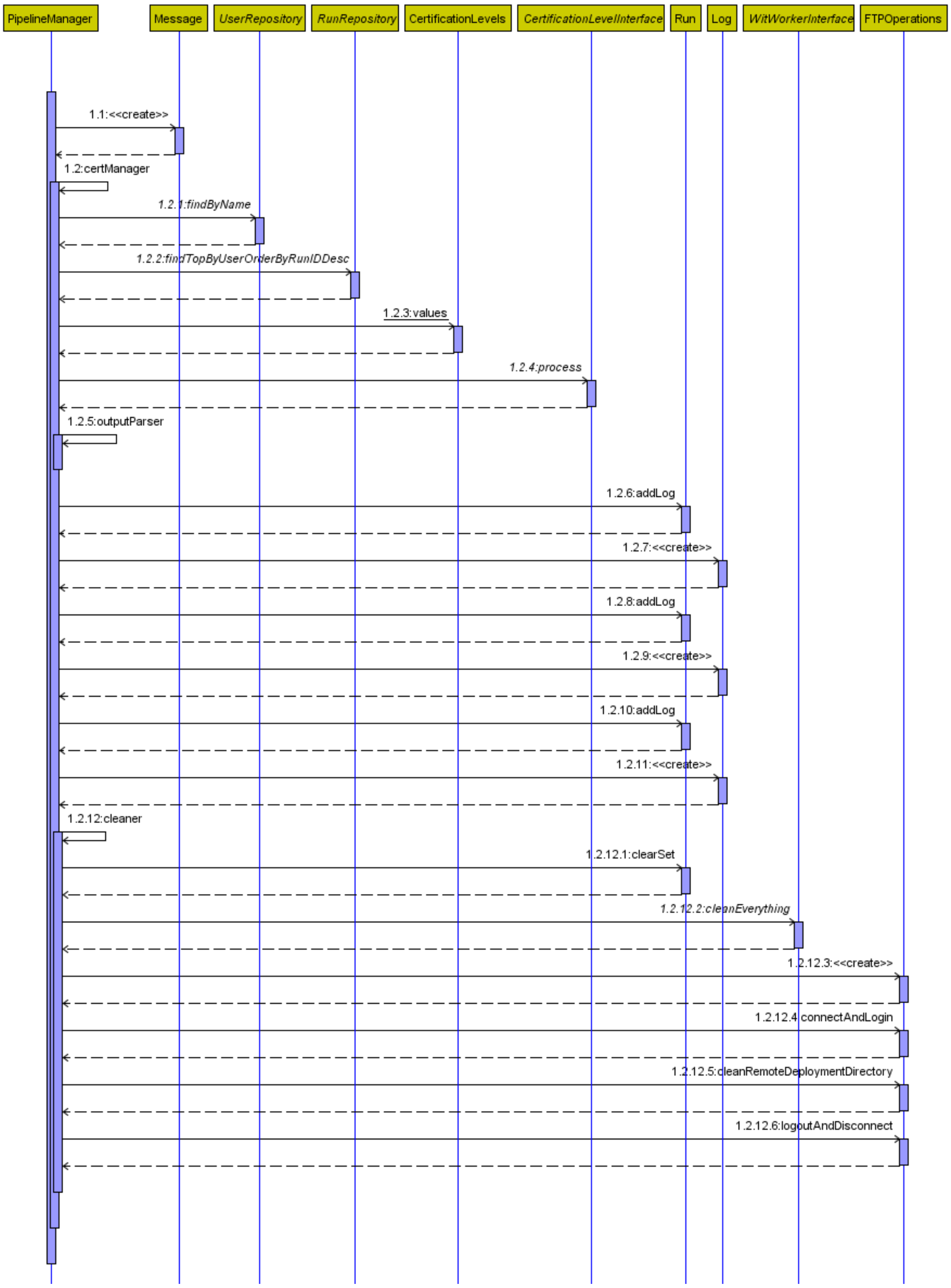
B1.1. Process initiated by uploading files

The original version of this diagram can be found here: https://goo.gl/OYAU2b



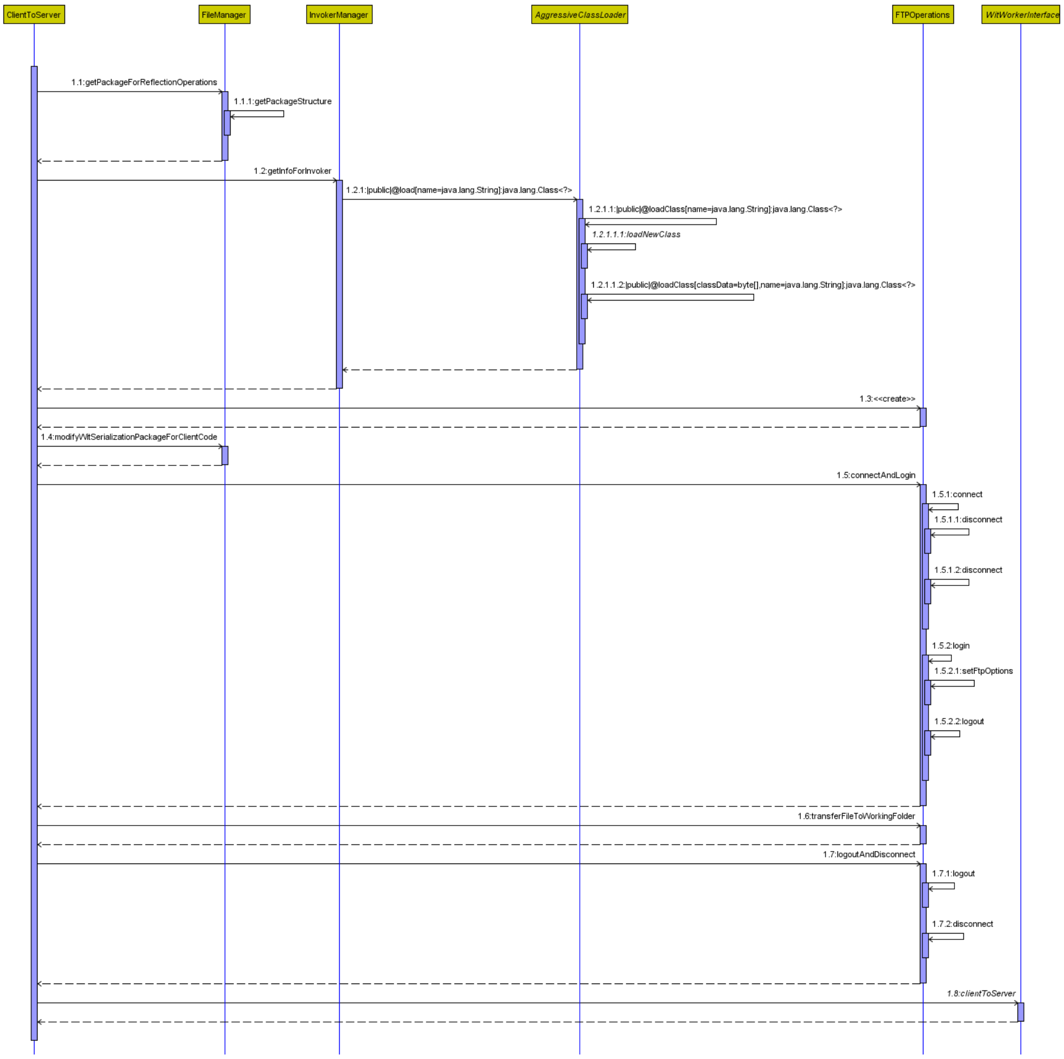
B1.2. Process initiated by submitting a WSDL

The original version of this diagram can be found here: https://goo.gl/c9fzOr



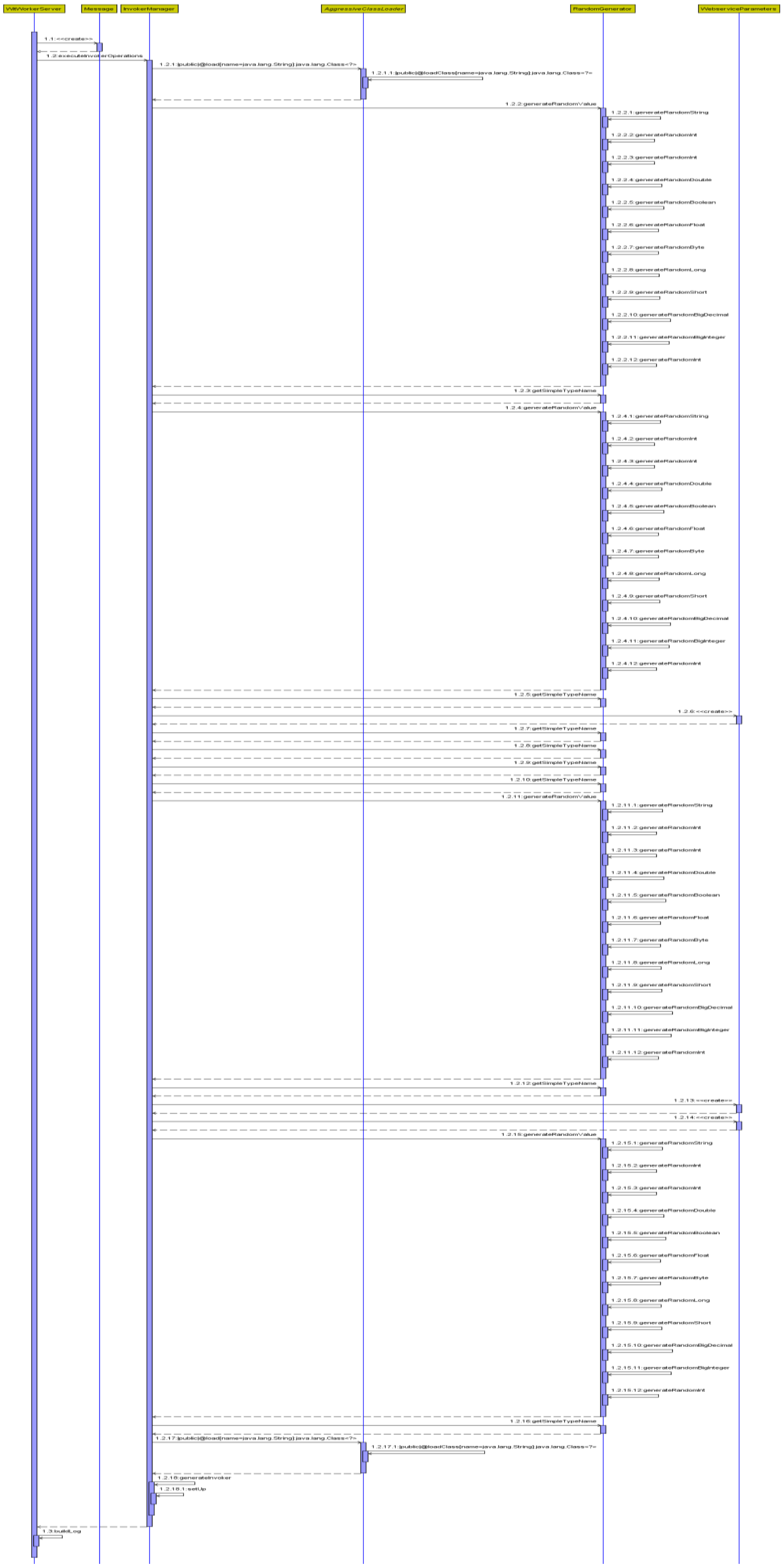
B1.3. Certification phase 3 call from the Pipeline Manager

The original version of this diagram can be found here: https://goo.gl/n634Dn


ClientToServer
FileManager
InvokerManager
AggressiveClassLoader
FTPOperations
WitWorkerInterface
1.1:getPackageForReflectionOperations
1.1.1:getPackageStructure
1.2:getInfoForInvoker
1.2.1:|public|@load[name=java.lang.String]:java.lang.Class<?>
1.2.1.1:|public|@loadClass[name=java.lang.String]:java.lang.Class<?>
1.2.1.1.1:loadNewClass
1.2.1.1.2:|public|@loadClass[classData=byte[],name=java.lang.String]:java.lang.Class<?>
1.3:<<create>>
1.4:modifyWitSerializationPackageForClientCode
1.5:connectAndLogin
1.5.1:connect
1.5.1.1:disconnect
1.5.1.2:disconnect
1.5.2:login
1.5.2.1:setFtpOptions
1.5.2.2:logout
1.6:transferFileToWorkingFolder
1.7:logoutAndDisconnect
1.7.1:logout
1.7.2:disconnect
1.8:clientToServer

B1.4. Certification phase 3 – Client code to web service communication

The original version of this diagram can be found here: https://goo.gl/gZaMOR


WitWorkerServer
Message
InvokerManager
AggressiveClassLoader
RandomGenerator
WebserviceParameters
1.1:<<create>>
1.2:executeInvokerOperations
1.2.1:|public|@load[name=java.lang.String]:java.lang.Class<?>
1.2.1.1:|public|@loadClass[name=java.lang.String]:java.lang.Class<?>
1.2.2:generateRandomValue
1.2.2.1:generateRandomString
1.2.2.2:generateRandomInt
1.2.2.3:generateRandomInt
1.2.2.4:generateRandomDouble
1.2.2.5:generateRandomBoolean
1.2.2.6:generateRandomFloat
1.2.2.7:generateRandomByte
1.2.2.8:generateRandomLong
1.2.2.9:generateRandomShort
1.2.2.10:generateRandomBigDecimal
1.2.2.11:generateRandomBigInteger
1.2.2.12:generateRandomInt
1.2.3:getSimpleTypeName
1.2.4:generateRandomValue
1.2.4.1:generateRandomString
1.2.4.2:generateRandomInt
1.2.4.3:generateRandomInt
1.2.4.4:generateRandomDouble
1.2.4.5:generateRandomBoolean
1.2.4.6:generateRandomFloat
1.2.4.7:generateRandomByte
1.2.4.8:generateRandomLong
1.2.4.9:generateRandomShort
1.2.4.10:generateRandomBigDecimal
1.2.4.11:generateRandomBigInteger
1.2.4.12:generateRandomInt
1.2.5:getSimpleTypeName
1.2.6:<<create>>
1.2.7:getSimpleTypeName
1.2.8:getSimpleTypeName
1.2.9:getSimpleTypeName
1.2.10:getSimpleTypeName
1.2.11:generateRandomValue
1.2.11.1:generateRandomString
1.2.11.2:generateRandomInt
1.2.11.3:generateRandomInt
1.2.11.4:generateRandomDouble
1.2.11.5:generateRandomBoolean
1.2.11.6:generateRandomFloat
1.2.11.7:generateRandomByte
1.2.11.8:generateRandomLong
1.2.11.9:generateRandomShort
1.2.11.10:generateRandomBigDecimal
1.2.11.11:generateRandomBigInteger
1.2.11.12:generateRandomInt
1.2.12:getSimpleTypeName
1.2.13:<<create>>
1.2.14:<<create>>
1.2.15:generateRandomValue
1.2.15.1:generateRandomString
1.2.15.2:generateRandomInt
1.2.15.3:generateRandomInt
1.2.15.4:generateRandomDouble
1.2.15.5:generateRandomBoolean
1.2.15.6:generateRandomFloat
1.2.15.7:generateRandomByte
1.2.15.8:generateRandomLong
1.2.15.9:generateRandomShort
1.2.15.10:generateRandomBigDecimal
1.2.15.11:generateRandomBigInteger
1.2.15.12:generateRandomInt
1.2.16:getSimpleTypeName
1.2.17:|public|@load[name=java.lang.String]:java.lang.Class<?>
1.2.17.1:|public|@loadClass[name=java.lang.String]:java.lang.Class<?>
1.2.18:generateInvoker
1.2.18.1:setUp
1.3:buildLog

C1. Web service source code upload background process

In order to have a better understanding of the system, let us take a look at a typical exchange of information. In this case, we'll be looking at the most complex case, submitting files for the system to analyze. Figure 42 shows a graphical representation of the information exchange between the several components. In this interaction, one WitWorker and one Deployment server is represented because every individual test run only uses one of each. Also, there's no "External server" (as represented in Figure 24 because we're using the File upload option and not the Submit WSDL one and we're deploying the source code submitted (after several modifications) to our own servers. The following explanation will also consider that all steps are completed successfully.

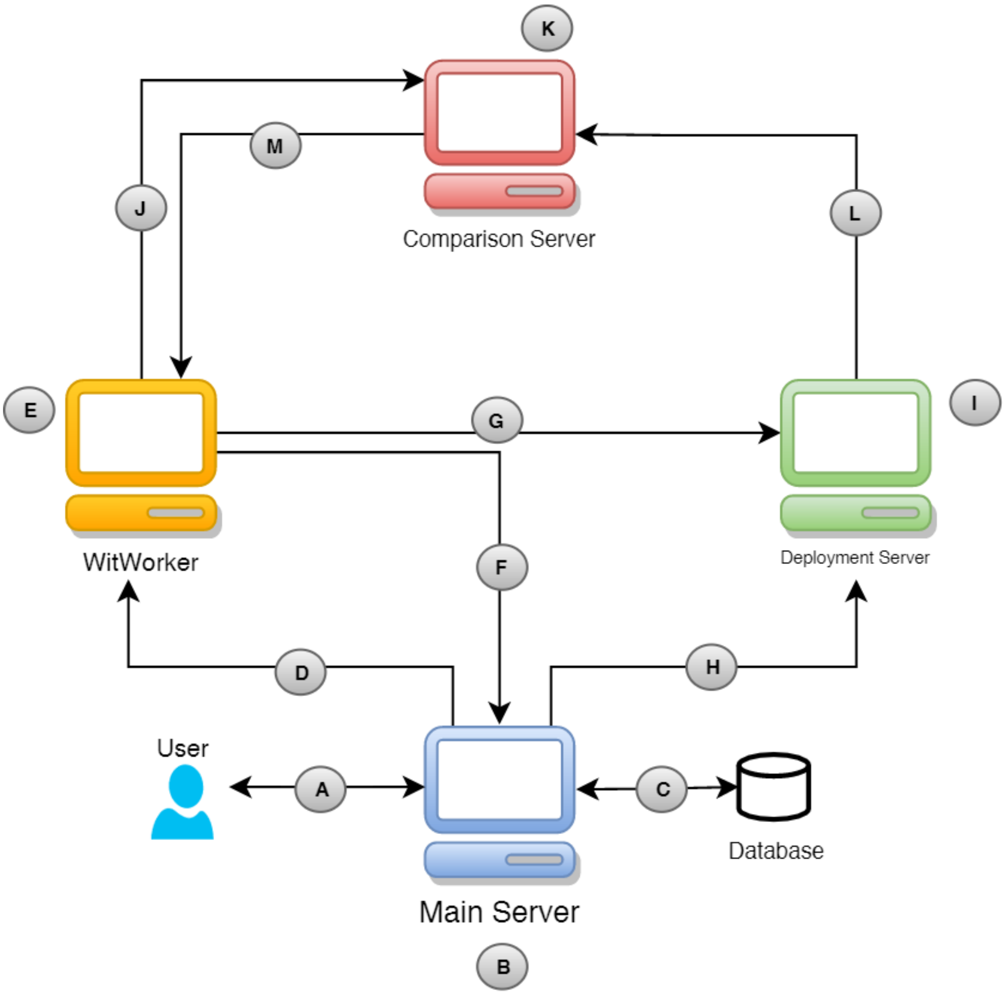


Figure 42. Source code upload background process

1. (A) The interaction begins with the user accessing the Vaadin-built webpage;
2. (B) User tries to login;
3. (C) Login data is queried against the Database;
4. (B) Log in successful and the user is now in the Dashboard view;
5. (B) User selects "File upload" in the menu;
6. (B,C) Main server queries Database for available tools and deployment servers. Shows them on the webpage.
7. (B) User selects some files and uploads them;

8. (B) User selects a client code generator, a deployment server, gives a name to the run and clicks “Start Process”;
9. (B) A new test run is immediately created in the database with the current timestamp, the name we mentioned in step 7 and associated with the user who started it;
10. (B) If there are no other threads running, the process can continue and the status of the run is updated to “executing”;
11. (B) The system creates the basic folder structure for the WAR file – the file that will be uploaded and deployed to the Deployment Server later on – the WEB-INF\classes and META-INF folder.
    a. In this step, the libraries from \resources\lib are also copied to the WEB-INF\lib. These libraries are needed for the serialization process explained later.
12. (B) The files uploaded are compiled to guarantee that the user submitted proper source code and all dependencies are present. If it fails to compile, the run is set to “completed”, the “debug” column associated with the run is filled to warn the user and the process ends.
13. (B) The system looks for the web service file in the submitted files and marks it for further analysis.
14. (B) The rest of the files are analyzed and, based on their package:
    a. Class files get copied to WEB-INF\classes\package\structure\.
    b. Class files get copied to the project folder \target\classes for further inspection later on.
    c. Java files get copied to the project folder \src\main\java\package\structure for further inspection.
15. (B) Main server looks for the first web method it finds in the web service and passes it to a method to search for non-supported field, parameter and return types (more on this on section Limitations.
16. (B) The serialization process begins. This means that the system, through reflection, analyzes the web service and all its dependencies.
    a. Uses the previously selected web method and searches all its parameters and return values for user defined classes with non-standard getters and setters.
    b. Inspects the method parameters;
    c. Inspects the return type of the method and generates a random one.
    d. Using a Velocity template, generates a new web service that differs from the original one in some aspects. Major differences are:
        i. All code from the original web service file is now gone, with the exception of the original imports, the original fields and the selected method.
        ii. The parameters of the web method are also fields. This is necessary so we can get their values during run time in a phase of the testing pipeline.
        iii. The method is now injected with some reflection code in order to get the fields’ values and pass them to the WitSerialization helper class (along with an instance of the method for reflection and recursion purposes).
        iv. Added imports for the serialization package;

    Overall, this template modifies the original web service to be able to execute the following tasks with the help of the WitSerialization class:

    - As soon as the web method is called, it sends the values received as parameters to the Comparison server;

- Returns a randomly generated value to the caller, but not before sending that value to the Comparison server.

17. (B) The modified web service is compiled and then copied to the same structure as above;
18. (B) A helper class called WitSerialization.java is also compiled and moved to the same folder structure.
    a. This class contains several helper methods to receive, serialize and send the values the web service is called with to the comparison server, along with the return value of the web method.
19. (B) All files and folders created above are compressed into a WAR file;
20. (H) The WAR file is sent to the previously selected deployment server by FTP.
21. (I) The deployment server waits until the transfer is complete and auto deploys the WAR file.
22. (B) The process then passes to the Test Manager domain and checks if it can ping the newly deployed web service WSDL endpoint.
23. (B) The Test Manager starts the what can be called as the Testing Pipeline. The process until now can be represented by Figure 43.

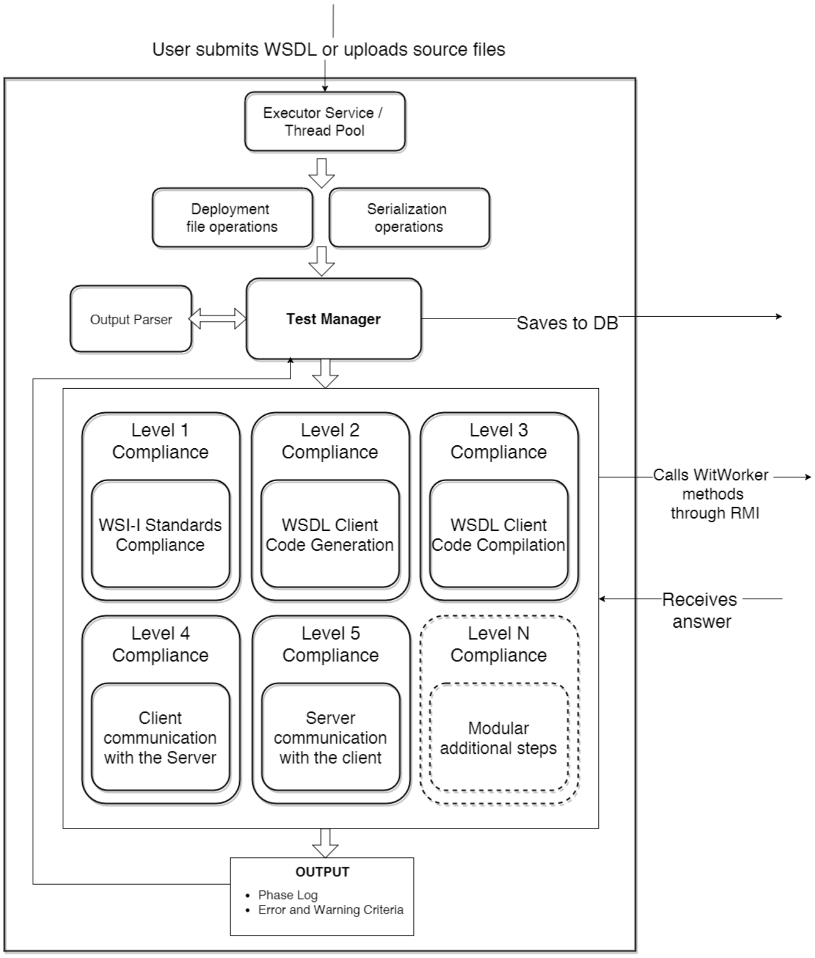

User submits WSDL or uploads source files
Executor Service / Thread Pool
Deployment file operations
Serialization operations
Output Parser
Test Manager
Saves to DB
Level 1 Compliance
WSI-I Standards Compliance
Level 2 Compliance
WSDL Client Code Generation
Level 3 Compliance
WSDL Client Code Compilation
Calls WitWorker methods through RMI
Receives answer
Level 4 Compliance
Client communication with the Server
Level 5 Compliance
Server communication with the client
Level N Compliance
Modular additional steps
OUTPUT
Phase Log
Error and Warning Criteria

Figure 43. Testing Pipeline

24. (B,D,E) The system calls the first phase – the WS-I Standards Compliance phase.
    a. (E, D) The number and order of phases are defined in the main server, but are run in the WitWorkers. Each WitWorker server has a RMI server running and, thanks to its interface, exposes several methods which can then be called by

the main server to run all the necessary operations. The application knows which server to call because of the tool selected in the "File Upload" web page.

25. (C) After analyzing the return value, saves certification level output to the database;
26. (B,D,E) System calls the second phase- The WSDL Client Code generation using the tool selected earlier in the web page which generates java code from the deployed web service;
27. (C) After analyzing the return value, saves certification level output to the database;
28. (B,D,E) System calls the third phase – WSDL Client code Compilation which compiles the previous code;
29. (C) After analyzing the return value, saves certification level output to the database;
30. (B,D,E) System calls the fourth phase, the Client communication with the server. This phase is a lot more complex than the previous ones:
    a. Before calling the method on the remote server through RMI, the main server needs to fetch some info for the WitWorker to build the Invoker class. This class, as the name suggests, will utilize the client code generated in the previous steps to call the web service.
    b. Using reflection once more, we fetch the web service from the previous operations and extract some basic info to pass to the method in the remote server. (So the remote operation knows where to fetch what)
    c. The system then sends the WitSerialization helper class to the tool working folder (through RMI, using the RMIIO library [51]).
    d. At this point, the remote method is called. The WitWorker server initiates the operations to build the Invoker class. All these operations are performed on the tool generated client code.
        i. Using reflection, it fetches all the fields of the class file which corresponds to the web method previously mentioned and generates random values for each one.
        ii. Using the same method as above, it also analyzes the generated class which corresponds to the return value of the web method.
        iii. It then creates the Invoker file using a Velocity template which, similarly to the modified web service, executes the following tasks with the help of the WitSerialization class:
            1. Before calling the web method, it sends the above generated random values to the Comparison server
            2. As soon as it receives the return value from the web method, it sends it to the Comparison server.
    e. Finally, it calls the Invoker.
31. (J) To send the necessary data to the Comparison server, the communication is done through TCP/sockets from the helper class WitSerialization (which is called from the Invoker)
    a. This data consists in, among others, the random generated values we talked before. More details about the protocol in 7.2 - Protocol used by the Comparison server.
32. (L) The Invoker class calls the web method in the web service, and so the modifications mentioned in **16** are triggered.
33. (K) The Comparison server, can be reached by tcp/sockets and waits for the following order of data:
    a. Values passed as parameters to the web method from the Invoker class;
    b. Values received as parameters from the web service / web method;
    c. Return value generated from the web service / web method;
    d. Return value received from the Invoker.

34. (M) After receiving all the data, the Comparison server send everything to the WitWorker via tcp/sockets.
    a. Lists and level of certification reached in each analyzed structure type and value;
35. (E) The WitWorker then builds two user friendly logs based on everything it received from the Comparison server.
    a. As the reader might already have noticed, the certification phase 4 also triggered the operations in which the certification phase 5 consists (sending the info from the web service back to the client code). To keep the Certification Pipeline modular and don't deal with a specific phase in a different manner, the WitWorker, after receiving everything from the Comparison server, keeps the data related to the phase 5 in memory.
36. (F) WitWorker sends the data of the phase 4 back to the main server (through RMI, as the return value of the method called in **30.**
37. (B) After analyzing the return value, saves certification level output to the database;
38. (B) System calls the fifth and last phase – server communication with the client.
39. (E) WitWorker just needs to send the phase 5 related info it already has in memory.
40. (B) After analyzing the return value, saves certification level output to the database;
41. (B) The system detects there are no more testing phases to run and calls its clean-up methods.
    a. Calls a method in the WitWorker server for it to clean the artifacts generated from the client code generation tool.
    b. Connects to the Deployment server via FTP and deletes the deployment directory.
42. (B) User clicks in "Results" on the left menu.
43. (B) Webpage displays the results of the test run, with the possibility of analyzing the log of each individual phase.

D1. Developer Manual

This section starts by explaining the structure of the system's source code and then demonstrates how to run the application (by going through all the necessary components) on other machines.

The code is publicly available on University's servers. [52].

D1.1 Structure

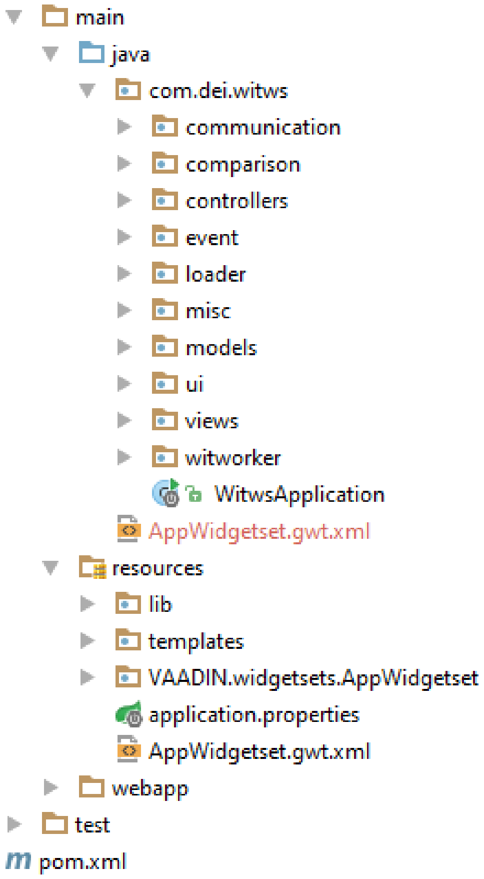
Figure 44 consists in a screenshot of the InteliJ's IDEA folder structure.

In the root of the project (inside the "main" folder) there are three main directories:

- **java** – Where the code of the WitWS Certification Pipeline is written;
- **resources** – External files used by the application;
- **webapp** – Contains the Vaadin themes and custom CSS rules.

Inside each one of those packages are several more:

- **java/com/dei/witws/communication** – Classes used to send information throughout the application;
- **java/com/dei/witws/comparison** – Contains the code necessary to run the Comparison Server.
- **java/com/dei/witws/controllers** – The majority of the application's features are coded in this package. E.g.: contains classes responsible for controlling the Certification Pipeline and to verify the compliance against the setters and getters standard nomenclature.



```
▼ main
  ▼ java
    ▼ com.dei.witws
      ▶ communication
      ▶ comparison
      ▶ controllers
      ▶ event
      ▶ loader
      ▶ misc
      ▶ models
      ▶ ui
      ▶ views
      ▶ witworker
        WitwsApplication
      AppWidgetset.gwt.xml
  ▼ resources
    ▶ lib
    ▶ templates
    ▶ VAADIN.widgetsets.AppWidgetset
      application.properties
      AppWidgetset.gwt.xml
  ▶ webapp
▶ test
m pom.xml
```

Figure 44. Code structure

- **java/com/dei/witws/event** – Event bus events used in *ui/DashboardUI* are listed here as inner classes. Also contains a simple wrapper for Guava event bus which defines static convenience methods for relevant actions.
- **java/com/dei/witws/loader** – Dynamic Class Loader code and its dependencies.
- **java/com/dei/witws/misc** – Helper classes that support the rest of the application. E.g. Random Generator.
- **java/com/dei/witws/models** – The tables of the PostgresSQL database and their corresponding repository methods are created from the classes in this package.
- **java/com/dei/witws/ui** – Contains the class which builds our graphical user interface and the dashboard menu layout. Since the user entry point to our application leads to this class (DashboardUI), it also controls the unauthorized access to the rest

of the views and controls the login/logout process with the help of the Guava event bus events.

- **java/com/dei/witws/views** – Classes that represent the different views in the application.
- **java/com/dei/witws/witworker –** The WitWorker server code is located in this package.
- **resource/lib –** Libraries that get bundled along the deployment file. They are essential to the serialization process.
- **resource/templates –** Velocity templates for the uploaded web service and for the client code Invoker. One for each supported programming language.
- **VAADIN/widgetsets/AppWidgetSet –** Package containing the custom built widgetset. Necessary for Vaadin. Will be explained in more detail in the next section.
- **/webapp –** Contains several themes provided by Vaadin which can be used by the application and one built specifically for this thesis.

Besides the folders and packages, there are also four more files worth of mention:

- **WitwsApplication**: It's the file used to run the main server and it's where we define the application as a Spring Boot application, which helps tremendously with the developing process. Even though the file is just some lines long, it contains some caveats:
    - @SpringBootApplication is a convenience annotation that adds all of the following:
        - @Configuration tags the class as a source of bean definitions for the application context.
        - @EnableAutoConfiguration tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings.
        - Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it automatically when it sees spring-webmvc on the classpath. This flags the application as a web application and activates key behaviors such as setting up a DispatcherServlet.
        - @ComponentScan tells Spring to look for other components, configurations, and services.
    - The main() method uses Spring Boot's SpringApplication.run() method to launch an application.
    - The run() method returns an ApplicationContext and this application then retrieves all the beans that were created either by the application or were automatically added thanks to Spring Boot. It sorts them and prints them out.
    - The amazing thing about SpringBoot and Vaadin is that the developer didn't need to write a single line of XML or HTM.L No web.xml file either. This web application is 100% pure Java and the developer doesn't have to deal with configuring any plumbing or infrastructure.
- **AppWidgetset.gwt.xml** – Generated by Maven with the goal = Vaadin:update-widgetset. Necessary for the custom widgetset the application uses. Will be explained in the next section.
- **application.properties** – Contains some system-wide configurations. E.g. datasource, datasource credentials, spring jpa data configurations, log level, etc.
- **pom.xml** – Maven configuration file. The application uses Maven as its dependency and build manager. All the plugins and libraries used by the app are declared here.

D1.2. Configuration

The system was built in almost in its entirety using Windows so this section contains instructions on how to configure everything on Microsoft's operating system. For other OS's, please go to Chapter 9 Application Limitations and Extensibility.

In order to run the entire application and the servers it depends on, several configurations are needed. Maven is used in several steps so be sure you have it installed.

D.1.2.1. Main server

The platform where the main server will be running needs to be defined:

- **\com\dei\witws\misc\Platform.java**
  - public static final String PLATFORM;

If running on Windows, CoreUtils also needs to be installed:

http://gnuwin32.sourceforge.net/packages/coreutils.htm

D.1.2.2. Database

The application uses a PostgreSQL database called "witws" to store its information. Since PostgreSQL does not support "create if not exists", the database must already exist in the system. The application will create everything else.

To configure it, the developer needs to change the following properties to proper values:

- **resources\application.properties**
  - spring.datasource.url
  - spring.datasource.username
  - spring.datasource.password
  - (Optional) spring.jpa.show-sql
    - To see the sql in the console.

D.1.2.3. Embedded servlet container

There is only one embedded servlet container – Undertow – related configuration:

- **resources\application.properties**
  - server.port = 8080

D.1.2.4. WitWorkers

WitWorkers contain the tools available to the application. It's extremely important to note that the system infers that all of them also include the ability to run the Certification Level 1 – the WS-I standards compliance.

For your server to support this compliance phase, several steps must be followed:

- Download the WS-I Testing Tools available at: http://www.ws-i.org/Testing/Tools/2005/06/WSI_Test_Java_Final_1.1.zip - Java version.
- Extract it to a folder. E.g.: G:\dev\tools\wsi-test-tools

- Add WSI_HOME to the system variables and point it to the folder you just extracted the zip to. E.g: G:\dev\tools\wsi-test-tools
- Add the bin folder to the "Path" system variable. E.g.: G:\dev\tools\wsi-test-tools\java\bin
- Download the XMLStarlet Command Line XML Toolkit: http://sourceforge.net/projects/xmlstar/files/
- Extract it to a folder. E.g.: G:\dev\tools\xmlstarlet-1.6.0
- Add the folder to the system's "Path" variable. E.g.: G:\dev\tools\xmlstarlet-1.6.0
- Install sed : http://gnuwin32.sourceforge.net/packages/sed.htm

- Install grep: http://gnuwin32.sourceforge.net/packages/grep.htm
- Install wget: http://gnuwin32.sourceforge.net/packages/wget.htm
- Go to the "support" folder in the root of the project and copy the files to any location. E.g.: G:\dev\tools\
- Add that location to the system's "Path" environment variable.
- Edit the instructions (paths) inside those files accordingly.

The InitiateComplianceCheck.bat file takes two arguments: the first one is the WSDL and the second one is the output directory which we send the results to. In src\main\java\com\dei\witws\witworker\WitWorkerServer.java modify the public static final variable "COMPLIANCE_CHECK_OUTPUT_DIR" to your liking, as long as it already exists.

At the end of the whole process (either if it went smoothly or it failed on any phase), the main server calls a cleanEverything() method on the WitWorker. This method will call the batch file "rmFilesFromArtifacts.bat" we moved earlier to clean the artifacts generated by the client code generation tools.

To modify the port the WitWorker RMI server listens to, change the following:

- public static final String RMI_SERVER_PORT;

These servers compile Java code with several library dependencies. Make sure you have the ones located in the /resources/lib folder in your classpath. Speaking of classpath, it must also contain the path to the java folder of the project and the classes output folder of the project. With that in mind, change the following variable to an appropriate path:

- public static final String CLASSPATH
  - E.g: ".;G:"+ fs +"dev"+ fs +"witws"+ fs +"src"+ fs +"main"+ fs +"java"+ fs +";G:"+ fs +"dev"+ fs +"jars"+ fs +"*;G:"+ fs +"dev"+ fs +"witws"+ fs +"target"+ fs +"classes"+ fs +"";
    - "fs" is a File.separator String. Useful to keep the implementation operating system-independent.

The client code generation tool folder output might also be changed. A path inside the project's java folder is recommended. Variable to change:

- public static final String ARTIFACT_FOLDER_LOCATION
  - E.g: "g:" + fs + "dev" + fs + "witws" + fs + "src" + fs + "main" + fs + "java" + fs + "artifact" + fs.

The system supports the code generation tool Axis2. Its artifacts are generated to a similar, but slightly different folder structure – regardless of the output folder chosen, it is always created inside a "src" folder. so the following variable also needs to be changed:

- public static final String ARTIFACT_FOLDER_LOCATION_AXIS
    - E.g.: "g:" + fs + "dev" + fs + "witws" + fs + "src" + fs + "main" + fs + "java" + fs + "src" + fs + "artifact" + fs;

WitWorkers, at a certain part of the testing pipeline, also wait for an answer from the Comparison Server (in the form of a socket connection), so to change the port it listens to, modify the variable:

- public static final int ANSWER_FROM_COMPARISON_SERVER_PORT.

After setting up all these configurations, the dev can now add tools to the pool. Wsimport and wsconsume are currently supported so they can be added to the witws_tool table in the database located in the main server. The rest of the columns must match the paths and info configured in the above steps.

On a side note, given that the system was developed under Windows, most command line instructions had to be slightly modified by adding "cmd.exe" and "/c" at the beginning of the call. The application does this automatically, however, some other commands might need that additional those additional strings. Keep that in mind.

D.1.2.5. Presentation layer – Vaadin

If the app is being run from Maven or from the IDE, then there should be no aditional Vaadin configurations. However, since the app uses a custom widget some complications might arise. If, when running the app and trying to load the UploadView, an error occurs, the dev should follow the following steps:

1. Delete any *.gwt.xml already existing in the project
2. Create a AppWidgetset.gwt.xml in the src/main/resources:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE module PUBLIC "-//Google Inc.//DTD Google Web Toolkit 2.5.1//EN"

        "http://google-web-toolkit.googlecode.com/svn/tags/2.5.1/distro-source/core/src/gwt-module.dtd">

<module>

        <inherits name="com.wcs.wcslib.vaadin.widget.multifileupload.MultiFileUploadWidgetSet"/>

</module>
```

3. Add the @Widgetset("AppWidgetset") annotation the Main UI
4. Run As - Maven clean
5. Refresh
6. Run As - Maven build ... (create a new run configuration with goal = vaadin:update-widgetset) This will create a new AppWidgetset.gwt.xml in the src/main/java folder.
7. Run As - Maven build ... (create a new run configuration with goal = vaadin:compile)
8. Copy the entire generated AppWidgetset folder from target/com.dei.witws-0.0.1-SNAPSHOT to src/main/webapp/VAADIN/widgetsets.
9. Run As - Maven install

Additionally, to guarantee that the theme is displayed correctly, run the following Maven targets (either inside the IDE or in the command line, on the root of the project):

- mvn vaadin:update-theme vaadin:compile-theme vaadin:compile

D.1.2.6. Deployment servers

The deployment servers aren't exactly part of the system since there is no code about them. They are just servers we have full access to. Their features, credentials and available deployment tools, just like the WitWorkers, must be inserted into the main database located in the same machine as the main server. In order for the system to recognize and use a deployment server, a new entry in the "deployment_server" must be inserted with all the columns correctly filled.

A FTP server needs to be running on port 21.

Additionally, the server has to be reachable and port 21 for FTP communications must be open.

Please be careful about the deployment_server_path column. It is always relative to the path being shared in the FTP server. *Example:* On your FTP server, if the deployment folder of the tool you're adding is "c:\dev\tools\wildfly\standalone\" and you're only giving access to \tools and its subfolders, then the deployment_server_path column in the table needs to be "\tools\wildfly\standalone\"

Out of the box, the system supports Glassfish and Wildfly so both can be installed in the machine.

The configuration of those application servers is out of the scope of the project, so please refer to their official documentation.

D.1.3. How to run

Before trying to run any of the components, a cleanup should be executed. Either delete the /target folder by hand or use the following Maven target (recommended), by typing the following command in the terminal (in the root of the project, where the pom.xml file is located):

- mvn clean

The application was created using InteliJ's IDEA 15, so in order to run it the dev cab import it to the same IDE and everything should work out of the box. To run the several servers just navigate to the corresponding files, right click on them and press "Run"- this is the recommended and supported approach. The dev can repeat this process as many times as necessary throughout several machines in order to deploy more WitWorkers. Don't forget that there can only be one instance of both the main server and comparison server.

D.1.3.1. Main server

Since the app uses Maven, it's also possible to run the application just by calling it in the tool. In the root of the project type:

- mvn exec:java
  - This will run the main server.

To deploy the main server as a JAR file, it must be bundled with all its dependencies, so the following mvn targets must be run in order:

1. mvn vaadin:update-widgetset
2. mvn vaadin:update-theme
3. mvn vaadin:compile-theme
4. mvn vaadin:compile
5. mvn install

This will create a file in the /target folder called com.dei.witws-1.0-RELEASE.jar . Run it by typing in the command line:

- java –jar com.dei.witws-1.0-RELEASE.jar

The main server runs on port 8080. See section D1.2.3 in order to change it.

Sometimes, when running the main server from the JAR and accessing the application, the GUI might not look as expected and the server log displays the following line:

Requested resource [/VAADIN/themes/witws/styles.css] not found from filesystem or through class loader. Add widgetset and/or theme JAR to your classpath or add files to WebContent/VAADIN folder.

It happens because the packaging step (in maven install) didn't include the webapp/VAADIN/themes folder. A workaround for now is to copy the VAADIN folder and its contents to /resources.

D.1.3.2. Deployment server

Assuming the application servers are already configured, running them is trivial. In the command line, type:

- GlassFish: (path_to_glassfish_bin_folder)\asadmin restart-domain
  - To bind to a specific address:

asadmin set server-config.jms-service.jms-host.default_JMS_host.host="hostname"

  - Restart the server.
- WildFly: (path_to_wildfly_bin_folder)\standalone.bat –b=(address_to_bind_to)

D.1.3.3. WitWorkers

As explained earlier, to run the WitWorkers go to the corresponding package, right click on WitWorkerServer.java and select "Run" – this is the supported method because of some Java reflection steps occurring in the code.

WitWorkers are part of the same project, and so, deploying them as a JAR is slightly different but equally easy. Since it's outside of the scope of this project, but refer to [53] This approach is not recommended because it forces the dev to change several more files and configurations (not detailed here) for the system to work as intended.

D.1.3.4. Comparison server

Same process as the WitWorkers-

E1. Mockups

In order to better represent the required functionality of the system, some mockups were made. Since a lot of the work is done in the background and is transparent to the user, the user interface can be simplified.

Figure 45 represents the screen displayed when the user accesses the web application URL. It contains the usual sign in fields: username, password and a link for registration.

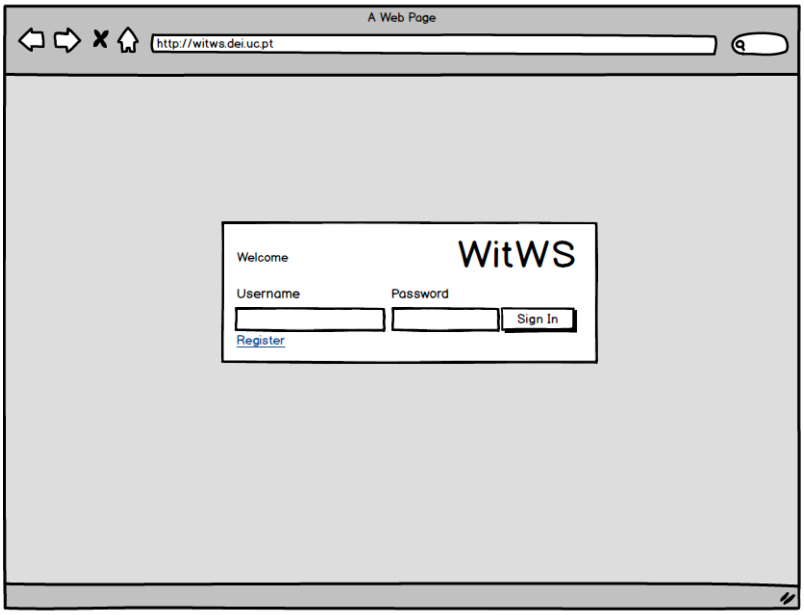


Figure 45. Login mockup

When the user clicks on the previous "Register" link, he/she is taken to the screen represented in Figure 46. In order to sign up, the user needs to input a username, a password and an email.

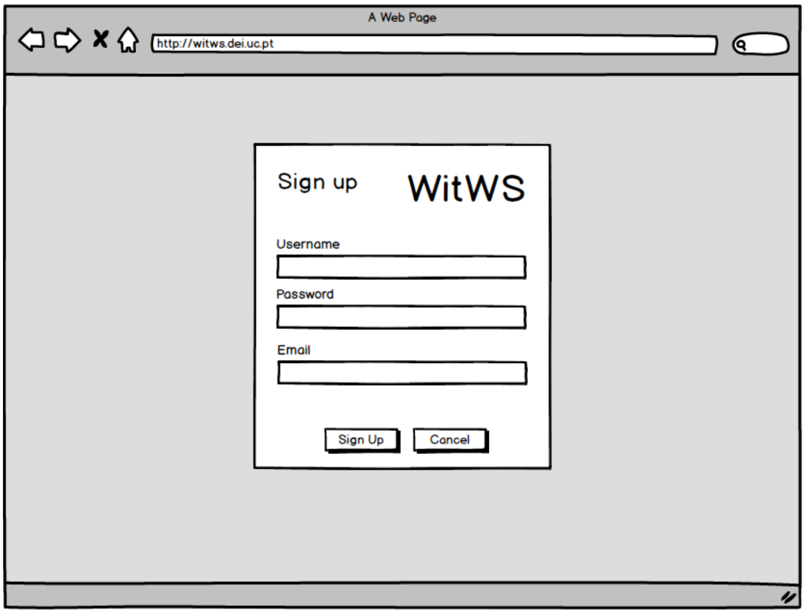
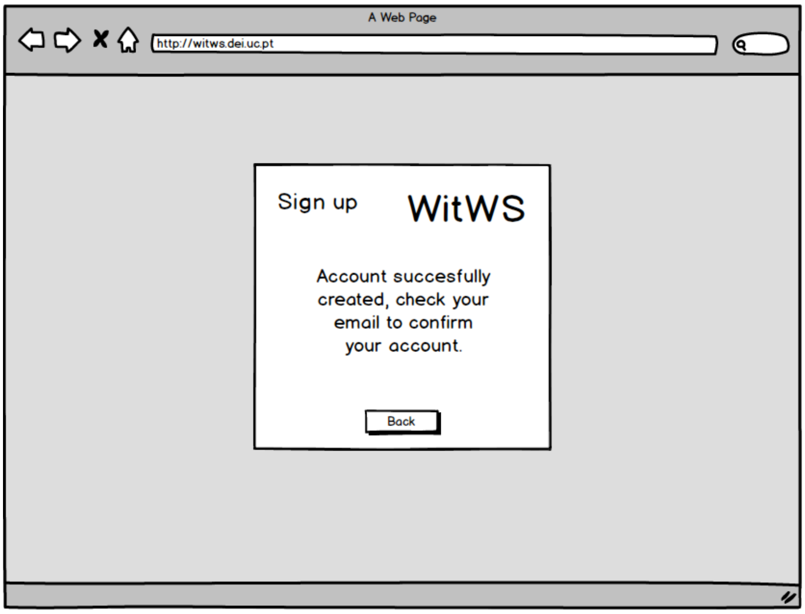


Figure 46. Sign up mockup

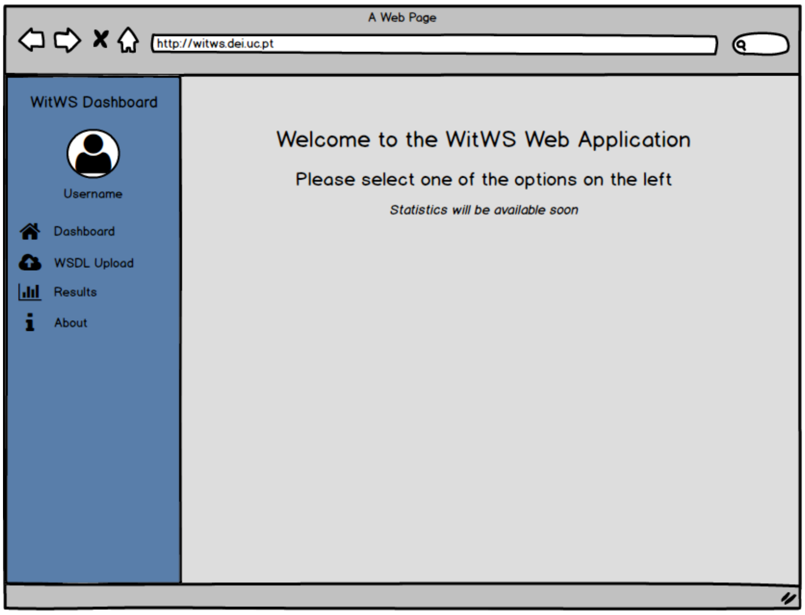
If the user inputs valid data in all fields (non-existing username, password not too short and a well-formed email) and presses "Sign Up", he/she will be taken to a screen like the one in Figure 47.



Account succesfully created, check your email to confirm your account.

Figure 47. Sign up succeeded

After confirming his/her account (by clicking on the link sent to the email address provided) and logging in with valid credentials on the first page, the user is then taken to the main application screen, the dashboard - Figure 48. In an initial phase of the project, the dashboard will only contain a greeting message and the main application features located in the left static menu. This screen can later be extended to include some statistics about the submitted WSDL's (e.g. number of WSDL's that reached the higher certification level available).



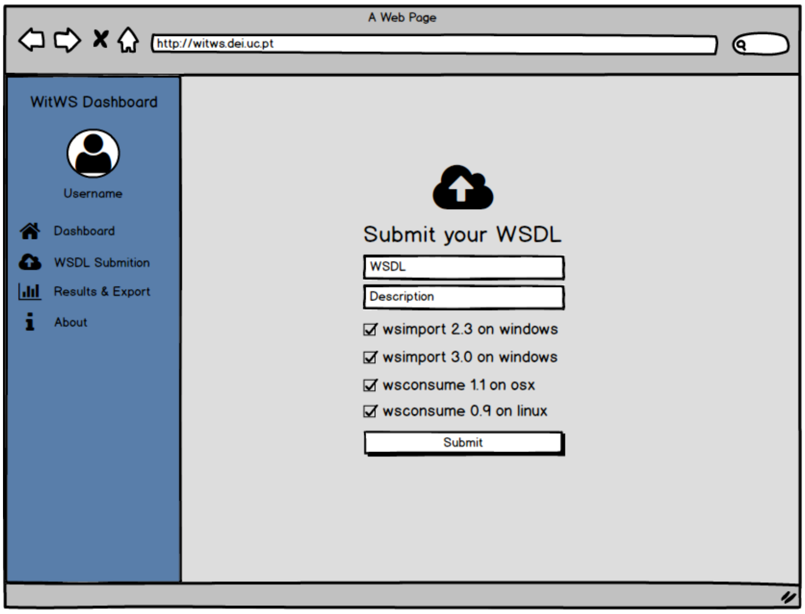
Welcome to the WitWS Web Application

Please select one of the options on the left

Statistics will be available soon

Figure 48. Dashboard mockup

Clicking on "WSDL upload", the user will be taken to a screen like the one in Figure 49, where it is possible to submit a given WSDL, give it a description and select several tools on different operating systems to check it against (by default, all of them will be selected).



Submit your WSDL

WSDL

Description

☑ wsimport 2.3 on windows

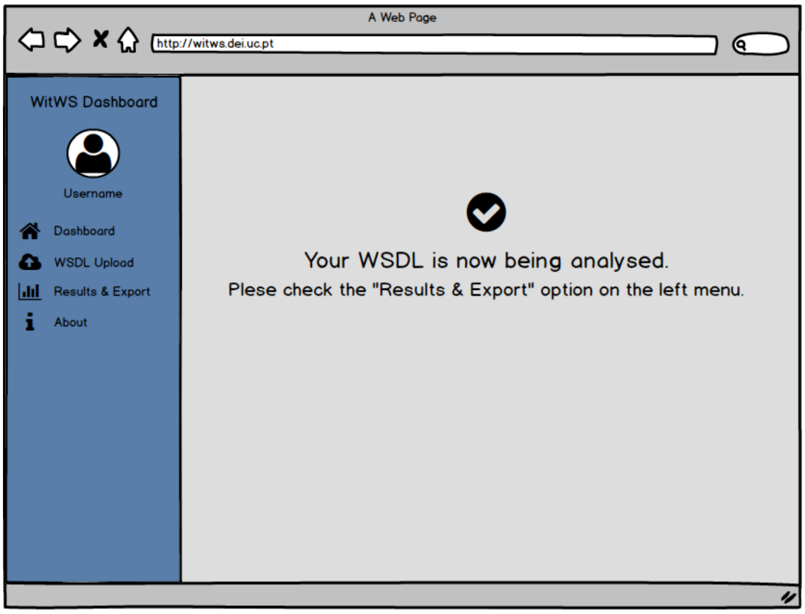
☑ wsimport 3.0 on windows

☑ wsconsume 1.1 on osx

☑ wsconsume 0.9 on linux

Figure 49- WSDL upload mockup

After clicking the "Submit" button and if the URL is valid, the system will display a screen like the one in Figure 50



Your WSDL is now being analysed.

Plese check the "Results & Export" option on the left menu.

Figure 50. WSDL submitted mockup

The screen in **Error! Reference source not found.** represents a scenario where the user already submitted 4 WSDL's. The system has already completed the analysis on the first two WSDL's, it's still analyzing the third and has put the forth on hold since it can only analyze one WSDL at a time. As the reader can see, each WSDL has two options: see its analysis detailed results or export them to a file on the user's machine.

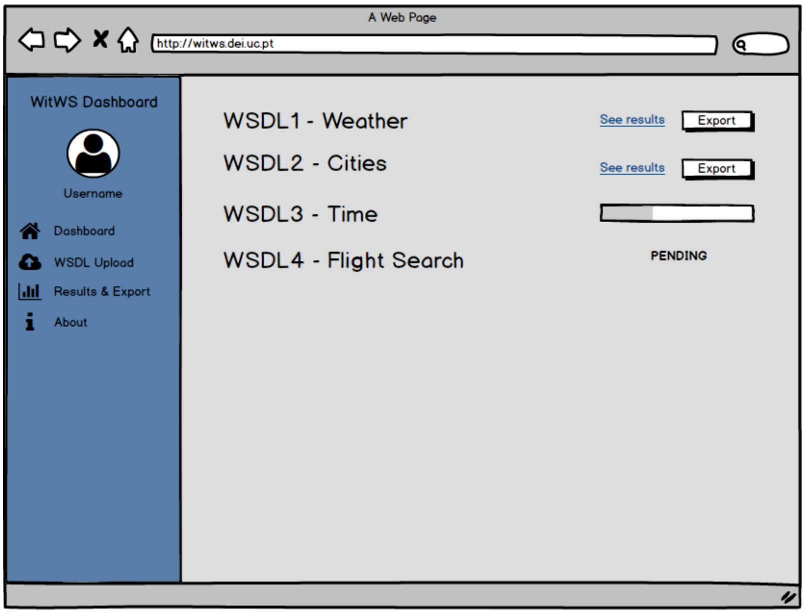


Figure 51. Results overview mockup

By clicking on any "See results" button on the mockup above, the user will be taken to a screen like the one below in Figure 52. The system will then display all the available certification levels the WSDL went through (in this thesis we implement five), along with a simple visual representation of the result of each one: green if the system didn't encounter any error or warning in that level, yellow if it encountered any warnings and red if it produced at least one error. The various levels are all part of a pipeline and cannot co-exit independently, meaning that if a level produces an error, all of the other ones next in the pipeline will automatically fail.

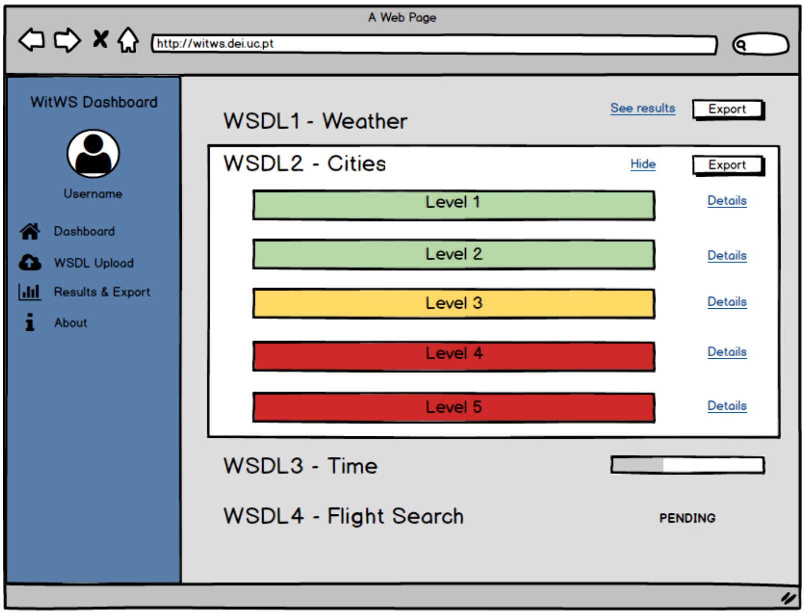
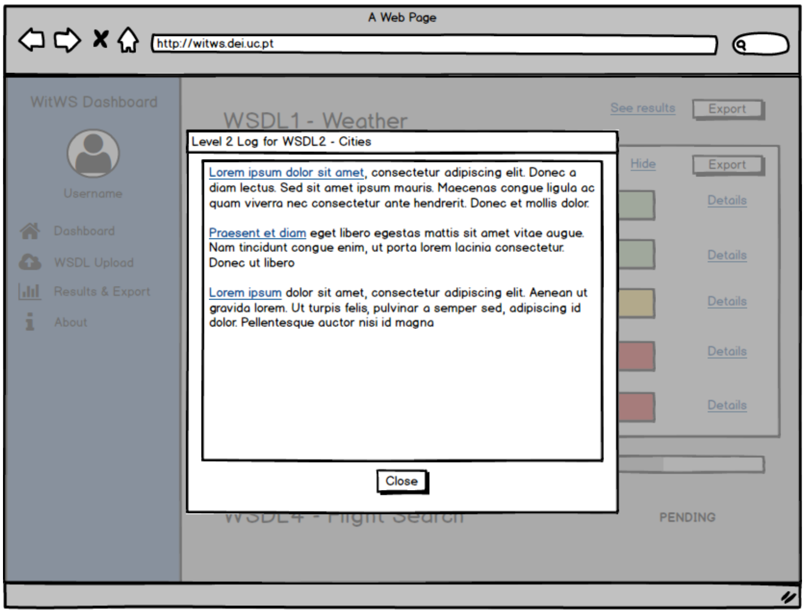


Figure 52. Detailed results mockup

Each certification level will have its own associated details/log. This way the user can investigate went wrong with a certain phase and make the necessary modifications. By clicking on the link "Details", the system will display the log of the select phase, like in the. Figure 53



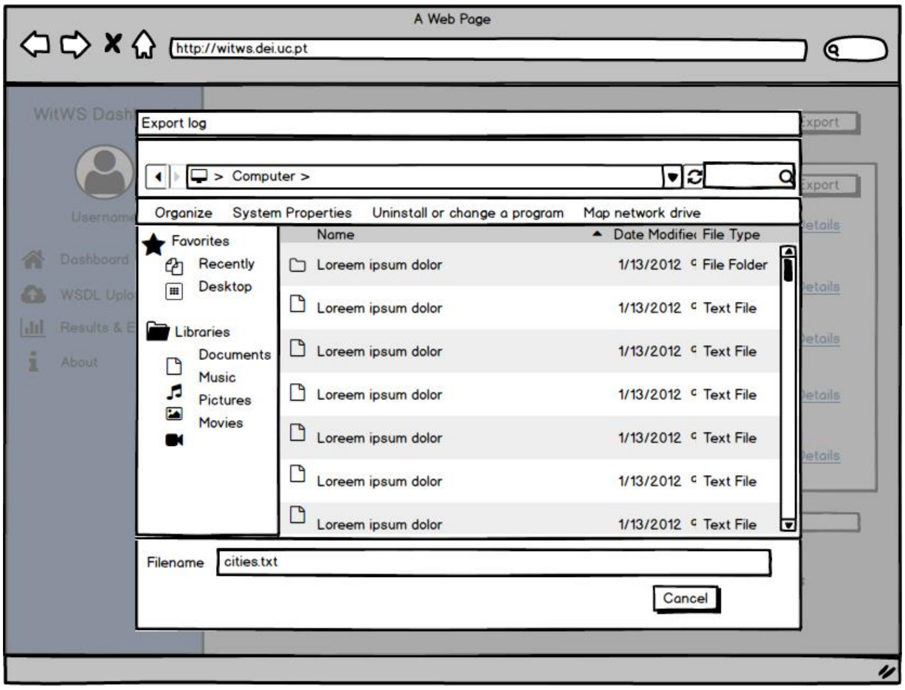
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec a diam lectus. Sed sit amet ipsum mauris. Maecenas congue ligula ac quam viverra nec consectetur ante hendrerit. Donec et mollis dolor.

Praesent et diam eget libero egestas mattis sit amet vitae augue. Nam tincidunt congue enim, ut porta lorem lacinia consectetur. Donec ut libero

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aenean ut gravida lorem. Ut turpis felis, pulvinar a semper sed, adipiscing id dolor. Pellentesque auctor nisi id magna

Figure 53. Log mockup

The user will also have the option to save the entire log of each analysis to a file in his/her own machine by clicking in the "Export" button of each certification level, like in the Figure 54

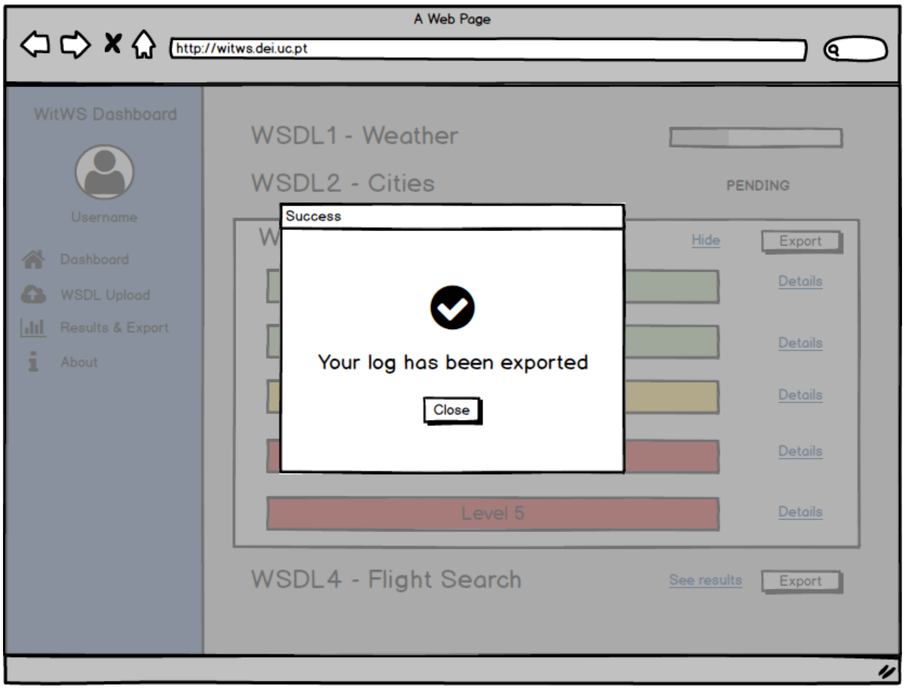


Name	Date Modifie	File Type
Loreem ipsum dolor	1/13/2012 c	File Folder
Loreem ipsum dolor	1/13/2012 c	Text File
Loreem ipsum dolor	1/13/2012 c	Text File
Loreem ipsum dolor	1/13/2012 c	Text File
Loreem ipsum dolor	1/13/2012 c	Text File
Loreem ipsum dolor	1/13/2012 c	Text File
Loreem ipsum dolor	1/13/2012 c	Text File

Filename: cities.txt

Figure 54. Export mockup

After exporting the file, a simple pop will tell the user if everything went ok, like in the screen represented in Figure 55



Your log has been exported

Figure 55. Export succeeded mockup

F1. Research paper: INTENSE - INteroperability TEstiNg ServicE

Starts in the next page.

[INTENSE]: Interoperability Testing as a Service

ABSTRACT

The web services technology has been created to support communication between heterogeneous platforms. Despite its maturity, built upon more than a decade of experience, research and practice show that the technology still fails to connect the two sides of an interaction, even when the programming languages involved are the same. This is especially concerning for providers, as a failure in the inter-operation of web services can bring in disastrous consequences for the services involved, which frequently support businesses. In this paper we present INTENSE, an on-line web application designed to test the interoperability of a web service against specific client-side platforms. The tool is able to test not only the pre-runtime steps involving code generation, but also the end-to-end runtime communication present in a web service interaction with a client. INTENSE has been used to test a set of web services deployed on Glassifsh and WildFly against the well-known Metro JAX-WS, JBossWS, and Axis2 client platforms and was able to disclose severe interoperability issues.

CCS Concepts

• Information systems→Simple Object Access Protocol (SOAP) • Information systems→Web Services Description Language (WSDL) • Applied computing→Enterprise interoperability • Software and its engineering~Software testing and debugging

Keywords

web services; interoperability; testing; reliability

INTRODUCTION

The web services technology was created with the goal of providing interoperable communication between heterogeneous platforms, allowing applications to be brought to the web environment. Nowadays, web services are found deployed on the web and used to provide service to consumers around the world, many times supporting business-critical services. In these environments, a single failure in the interaction of a consumer with a service can directly result in the loss of revenue (e.g., loss of a business transaction) and have disastrous consequences for the businesses involved (e.g., reputation losses).

In a web services environment, a provider offers a service for consumers (i.e., the clients) and announces the service interface in a WSDL file, which can include several operations, each one typically accepting one or more input parameters and returning an output object. When a developer wants to create a client for a web service, s/he typically uses a tool to process the WSDL file and generate code that is specific to the client-side platform (e.g., Java, C#, Python) and allows to invoke the web service in an easy manner. This generated code and supporting platform (i.e., a web service framework) is responsible for, at the client side, translating the application-specific objects (e.g., C# objects) to SOAP and send this SOAP message to the server. At the server-side the SOAP message is translated to the language specific objects (e.g., Java objects).

Experience and research clearly show that despite the effort put in bringing in interoperability between heterogeneous systems it is still very frequent to find cases where the different systems (i.e., client and server) are unable to inter-operate [1]–[3]. Thus, previous research reports cases where message contents are lost between client and server; support code cannot be generated or sometimes cannot be compiled in a specific platform, among many others [1]–[3]. These cases are due to many factors, ranging from poor web service platforms implementations, to issues that can be traced back to the specification documents, which are written in natural language.

Recognizing the interoperability problem, the WS-I organization [4] has been working, for over a decade, on creating standards to promote interoperability, and this includes refining and restricting the web services specifications and producing tools that can be used to test web services [5], [6]. The problem is that, despite the effort, still a WS-I compliant service shows problems to inter-operate with specific client platforms. Also, testing tools nowadays are usually limited to execute the WS-I recommended tests, which are also quite limited and are unable to detect severe inter-operation issues.

Authors in [1]–[3] studied the inter-operation problem by testing a large set of web services frameworks in which we were able to disclose numerous issues. Despite of the usefulness of the results, the authors carried out the tests in a semi-automatic way (at some points with manual intervention). The authors indicate the difficulty of building a tool that could be used by developers to test services for interoperability in a completely automatic way. In fact, this holds huge difficulties, which include having to generate code for multiple platforms, having to generate a test workload that fits any service, being ready to test a new platform with minimum effort, but especially producing client code that is able to communicate with any service, on any platform.

In this paper we present a web-based tool that implements the interoperability testing procedure discussed in [3] and automatizes the interoperability tests not only for pre-runtime steps (e.g., client-side code generation) but also for the runtime steps (e.g., communication between client and server). The tool is now prepared to automatically execute interoperability tests for web services built in Java and deployed in two major web servers (Glassfish and WildFly) against three client-side platforms (Oracle Metro, JBoss CXF, and Apache Axis2), but has been designed with extensibility in mind, thus it is easy to add support for further platforms or languages. We validated the tool against a subset of the services used by the authors in [1]–[3], obtaining 100% of correctness in the results. In this paper we also added a new set of services, to further study interoperability problems during the communication between client and server, as a means to further showcase the tool's capabilities.

This paper is organized as follows. Section II discusses related work in interoperability testing and Section III presents INTENSE from the user point-of-view. Section IV discusses the architecture and the internal processes that support INTENSE and Section V

Conference'10, Month 1–2, 2010, City, State, Country.

DOI: http://dx.doi.org/10.1145/12345.67890

presents the experimental evaluation carried out to illustrate the tools capabilities. Finally, Section VI concludes the paper.

Web Services and Interoperability

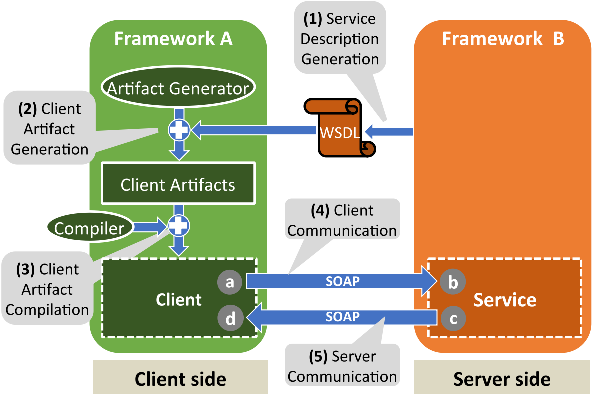
In a web services environment, the provider (i.e., the server) offers a service to consumers (i.e., clients). This service is described in an interface description document (i.e., a WSDL file), which describes a set of operations and their corresponding typed input/output parameters. Clients and servers interact by using a web service framework, which is the software that supports the inter-operation between them. In practice, a web service framework supports the creation and deployment of the service, the generation of support code at the client for easy invocation of the service, the serialization of programing language objects to SOAP messages at the client and the respective deserialization from SOAP messages to programming language objects at the server. Frameworks also support the reverse path taken by the messages from the server to the client. Of course, the service and client implementation, despite making use of the framework, need to be coded by the developer [7].

In our context, interoperability can be defined as the ability of a particular service to work together with some client [8]. In this paper we focus on syntactic interoperability, which ultimately refers to correct communication and exchanging of data between the two systems involved, and overall refers to the whole inter-operation process, which we describe in the next paragraphs.

1.1 *Background on web services inter-operation*

Fig. 1 represents a web services scenario, where client and server try to inter-operate. As we can see, two different frameworks are used in this process; the client uses framework A and the server uses framework B (although they could also use the same framework). For a client to be able invoke a remote operation, five steps have to be performed (numbered from 1 to 5 in Fig. 1). The correct conclusion of steps is critical for the inter-operation process, where in the end both endpoints exchange correct information.

The **Service Description Generation** step (1) usually starts automatically with deployment of the service at the server. The framework present at the server is able to read the source code and generate the corresponding WSDL at deployment time. There is also the possibility of manually coding the WSDL file and then generating the service interfaces, which is not very common. In either case, in the end the result is a WSDL file that can be used as basis for the inter-operation process.



A web services environment.

Using the WSDL produced in the previous step, the developer can proceed with **Client Artifact Generation** (2). The client artifacts are code that at runtime translates application-level calls to SOAP messages and send these messages to the server, thus making it easier to code a client as the low-level protocol details are hidden from the developer. Client artifacts are generated with a tool that is part of framework A (e.g., the wsimport tool from the Metro framework). Some frameworks generate these artifacts dynamically at runtime, i.e., there is no file generation.

The **Client Artifact Compilation** step (3) is necessary for platforms that require compilation before execution (e.g., Java, C#). In languages/frameworks (e.g., Python) where artifacts are dynamically generated at runtime, this step may not be necessary. Upon correct compilation, it is up to the developer to create the client code that calls the artifact methods that, in turn, will allow sending a message to the server. This latter step is named the **Client Communication** step (4), which starts precisely when the client code calls the remote operation using, as arguments, the object (or objects) required by the remote operation. This delegates the control to the framework that serializes the outgoing object (represented by *a* in Fig. 1) to SOAP and sends the message. The step ends when the application-level object is delivered to the service application (i.e., after being deserialized from SOAP to the exact object that the server-side platform expects), which is represented by *b* in Fig. 1. The object is then processed at the server (i.e., business logic is executed).

Finally, the **Server Communication** step (5) begins immediately before the server sends a response object back to the client (represented by *c* in Fig. 1). As in the previous step, control is delegated to the framework that translates the response object to SOAP. This step ends when the response object is delivered to the client application, after being translated from SOAP by the client-side framework.

1.2 *Approaches and tools for interoperability testing*

Research on web services interoperability has disclosed a few problematic issues in the past. The work in [9] analyzes this problem and identifies implementation issues, including problems with the representation of sequences of elements and namespaces in WSDL files. The authors discuss how the WS-I Basic Profile 1.0 [4], [10] addresses the raised issues, but at the same expose the limitations of the WS-I recommendations. Conclusions include the fact that following the WS-I Basic Profile recommendations can be a good starting point for reducing interoperability problems.

The use of natural language to build the specifications that guide the web services frameworks implementations can be the source of interoperability issues [11]. However, problems can happen much later with deficient implementation of the specifications. Inter-operation problems are prone to occur when native data types or specific constructs of the language being used to build a service are present at the interface. Still, there is no standard specifying which types are adequate to be used at an interface. This is corroborated by the work in [12] where the authors confirm the difficult to identify the right constructs to express data types that are in fact supported by all service frameworks.

A framework for auditing interoperability in SOA environments based in web services is proposed in [13]. The perspective is web services composition, so the authors take into account the possibility of creating compositions using different services, rather than a client/server point-of-view. In [14] a framework for the implementation of third party certification of quality properties in

Web Services and SOA environments is proposed. The framework is based on the presence of an external trusted authority that is responsible for checking and certifying systems. The target of the work in [15] is certifying security properties of Web Services. Modeling drives the generation of tests for the evaluation and certification of those properties. It is a test-based approach but still the goal differs from the one in this paper.

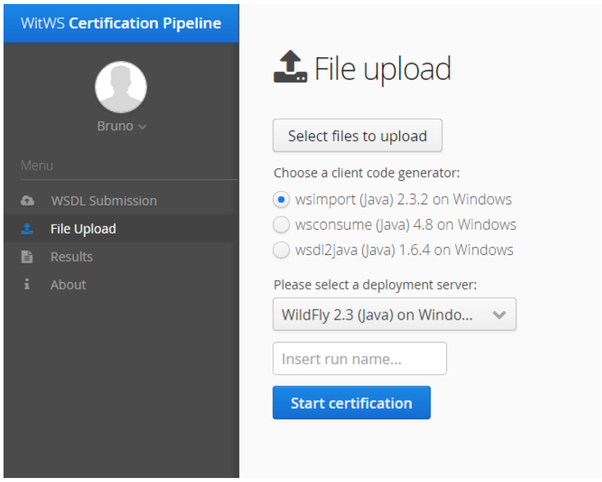
The industry has produced several **tools to test web services interoperability.** The WS-I Testing Tools Working Group created tools that can check if a web service complies with a set of profiles [4], [6], [16]. These profiles are essentially a set of recommendations, including clarifications and refinements of the web services specifications with the goal of promoting interoperability. Since our focus is inter-operation, the WS-I Basic Profile and respective tools are closely related to our work [6], [16]. Note however that the limitations of the WS-I standards and tools are well known. Even services that pass the tests defined by the WS-I tools are prone to show interoperability problems, which just shows the need for a superior approach and tool that can reveal such problems before systems are deployed.

In what concerns interoperability, most testing tools are limited to the execution of the WS-I tests. SOAPSonar [17] is a quite popular web services testing tool. The tool is able to check services for compliance with the WS-I Basic Profile 1.1, which is known to be largely insufficient when the goal is to disclose issues.

SoapUI [18] is another well-known tool that can be used to execute functional, load, and security tests on web services. The tool also supports the execution of WS-I Basic Profile 1.1 tests. In addition, SoapUI also supports the generation of client-side artifacts using a pre-defined set of client frameworks. Adding a new framework to the set is not possible without changing SoapUI's code, which is not a trivial task. One of the goals of INTENSE is that the tool can be easily extensible, so that adding a new framework can be reduced mostly to a configuration effort.

THE INTENSE APPLICATION

In this section we present an external view of our tool, mostly from the user point-of-view. The tool is available at *intense.dei.uc.pt*, and requires a simple registration procedure to be used. After authentication, the user has access to the tool's main interface, which is presented in Figure 2.



Choose a client code generator:

● wsimport (Java) 2.3.2 on Windows
○ wsconsume (Java) 4.8 on Windows
○ wsdl2java (Java) 1.6.4 on Windows

Please select a deployment server:

WildFly 2.3 (Java) on Windo...

Insert run name...

Figure 2. Starting an interoperability test.

As we can see in Figure 2, the user can **initiate a test** by:

1) Adding a WSDL file;

2) Upload the service source code (selected option in Figure 2).

Selecting *option 1)* means that the user wants to test a remotely deployed service, for which there will be no code access. The direct implication is that, as we do not have access to the service code, we will not be able to understand which information is arriving at the service code (sent by the client) and which information is leaving the service code (sent to the client). Thus, option 1) will result in testing only the first 3 steps of the inter-operation process, described in the previous section (service description generation, client artifact generation, and client artifact compilation), as we cannot evaluate the correction of the internal information at both endpoints. *Option 2)* gives full control to INTENSE and allows performing the full set of tests, as described in the next paragraphs. When the user selects this option 2), it also is prompted to **select the platform that will provide the service** (e.g., WildFly 9.1.0.Final on Java 1.8.0_20 on Windows 8). INTENSE will use this information to internally select the right platform and mechanisms for deploying the code and producing the WSDL file that will be used to begin the tests.

Regardless of the user option for the server-side platform, s/he can also **select the client platform for which the service should be tested against** (e.g., testing a given service against Metro 2.3.1 on Java 1.6 on Windows 7). After the user provides the necessary input for the tests (i.e., the WSDL or the code, and necessary options), the interoperability assessment will begin and will include tests at each of the possible inter-operation steps. Note that the interoperability assessment will test a service against the selected client platform, so the rest run is composed of 5 tests (executed for each of the 5 levels). Executing a test at any given level, can result in one of the following outcomes:

- **Error:** a fatal event, i.e., an event that prevents further inter-operation, occurs during the tests (e.g., a WSDL cannot be generated).
- **Warning:** one or more non-fatal events (i.e., events that do not prevent further inter-operation) occur during the execution of a particular test. This type of events suggests or indicates the presence of some issue, which has potential to later result in an inter-operation problem (e.g., client code can be generated but fails to compile).
- **Correct:** the test is able to finish without any visible issue (fatal or non-fatal).

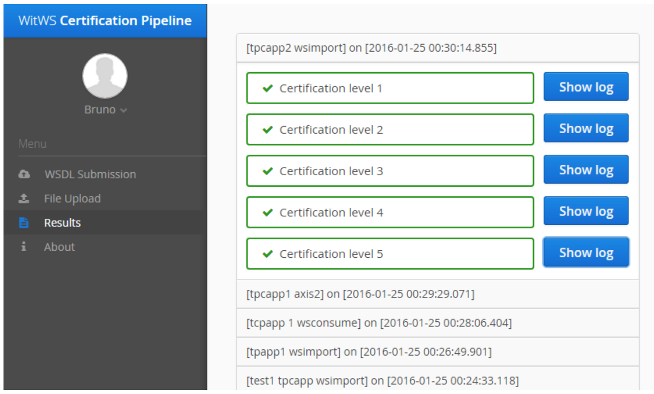
The result of testing a service against a particular client framework is named an interoperability level (from I to V), which describes the ability of that service to inter-operate with that client framework. This level refers to the last level successfully achieved during the inter-operation tests. Thus, after a new service is submitted to INTENSE for testing, one of the following outcomes can be achieved:

- **Level I:** this level will be reached if the server-side framework is able to generate a WSDL file. The WSDL (which may be the only input of the tests) is then checked for WS-I compliance.

- **Level II**: the framework is able to generate client-side code starting with the WSDL mentioned in the previous level.
- **Level III**: the framework, or the compilation tool required by the framework, is able to compile generated client-side code.
- **Level IV**: the client is able to use the generated client artifacts to correctly communicate with the server. A message that is sent from the client is correctly delivered at the service application, i.e., the contents, structure and data types are equivalent (please refer to Section 4 for further details on this comparison).
- **Level V**: the service is able to communicate correctly with the client. A reply that is sent from the service is correctly delivered (i.e., it holds an equivalent structure, types, and values) at the client application.

As an example, if service *ψ* is classified with Level III during the tests against framework *α*, this means that it is possible to: i) generate a WSDL file from service code; ii) to use framework *α* to generate client-side artifacts; iii) and to compile those artifacts with the tools required by framework *α* (although these tools are not necessary part of framework *α*, they might simply be the recommended or typical tools used). Thus, in the case of this example, one failure occurs when sending messages from client to server. A failure in this step means that the structure, types, and values of the objects being transmitted are not equivalent (please refer to Section 4 for the technical details).

Figure 3 shows an example of a concluded test in INTENSE, with a service that reached only Level II, when being tested against the Axis2 framework. As we can see, the service passes Level I and Level II without a single warning, but unfortunately it is impossible to compile the client artifacts produced by this tool. INTENSE stores all test information, so it's possible to inspect the tests results at each of the inter-operation levels by clicking the respective level. Figure 3 shows precisely this scenario, where the user can understand which type of error was generated in Level III and use that information to understand if some correction is required.



[tpcapp2 wsimport] on [2016-01-25 00:30:14.855]

✓ Certification level 1

✓ Certification level 2

✓ Certification level 3

✓ Certification level 4

✓ Certification level 5

[tpcapp1 axis2] on [2016-01-25 00:29:29.071]

[tcpapp 1 wsconsume] on [2016-01-25 00:28:06.404]

[tpapp1 wsimport] on [2016-01-25 00:26:49.901]

[test1 tpcapp wsimport] on [2016-01-25 00:24:33.118]

Figure 3. The detailed results of a test.

ARCHITECTURE OF INTENSE

In this section we describe our tool's architecture and the internal mechanisms that allow executing the tests. We put particular emphasis on the mechanisms used for Level IV and V (communication), due to the technical challenges involved.

Figure 4 presents the architecture of INTENSE, which in practice is kept in two parts, one corresponding to the INTENSE application (left hand-side of the figure) and the other one (right hand-side of the figure) holding three components that are strictly responsible for conducting activities specific to the interoperability tests. The INTENSE application (left hand-side of the figure) is composed of three main layers: the presentation layer, which contains all aspects related with information display and user interaction; the core layer, which is the part of the application responsible for coordinating and executing the tests and contains, in this way, the central functionality of the tool; and the persistence layer, which handles all persistence requirements of the tool. In this section we describe the core layer architecture and operational details, including its interface with external systems, used by our tool.

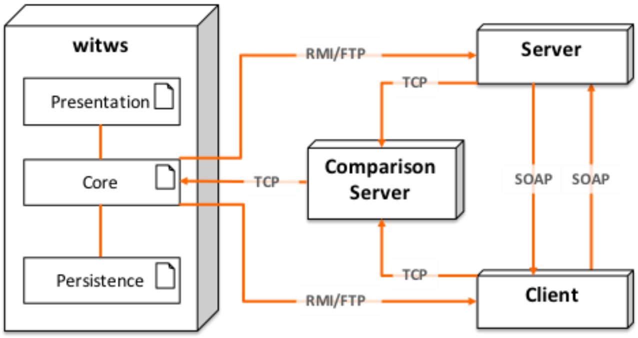


Figure 4. INTENSE architecture.

As we can see, the core layer architecture has contact with several components. These components are a *client* which will be responsible for executing all consumer-side tasks (i.e., generating artifacts, compiling artifacts, and generating and executing client code); a *server*, which contains a web server and the server-side framework, which allows deploying the service to be tested; a *comparison server* which is responsible for analyzing the equivalence of the information sent by the client and received by the server (and vice-versa) at runtime (i.e., during steps 4 and 5).

Figure 4 also shows RMI, FTP, and TCP communication between the different components. In short, RMI is used simply as a means to send commands that will be executed directly on the destination operation system; FTP is the service used for sending code files (i.e., the service code for deployment, or the client code for execution); and TCP is used to send lower-level information such as variable values received by a service during the tests, or the result of comparing objects coming from both ends of an interaction. All machines are connected via an isolated Ethernet network. Although most of the tests could take place in the same physical machine, we opted to have further isolation when using tools that might have bugs that can affect the availability of the machine.

As discussed in the previous section, the user can start a job in two ways, by submitting a WSDL, or by submitting the code. As this latter option includes all testing steps, we use it to explain the functionality of the tool from this point onwards. In the core layer of INTENSE there is a central component, which we refer to as *TestManager*, that is responsible for: waiting on a queue for new jobs (each job represents an interoperability test added by a user); starting the execution of a new job; and for coordinating all steps necessary to conclude the inter-operation tests that compose a job

When a new job is created, by submitting the code interface, and the *TestManager* has concluded any other previously submitted jobs, the *TestManager* starts the 5 tests that assess each of the steps of the whole inter-operation process. This corresponds to passing control to a pipeline of 5 components, which execute in sequence. Each of these components implement the tests required at each inter-operation step. We will not further detail the internal components of the tool in this paper, as they strictly reflect implementation choices. Thus, we focus on the main tasks involving the inter-operation tests. The next subsections describe the technical details involving the implementation of the tests required by the 5 inter-operation steps.

Service Description Generation

After the user uploads the service source code s/he also identifies the server platform required to deploy the code and initiate the tests. Before deploying the service, the service code is pre-processed to remove any business logic and further prepared to handle the next phases of testing. In particular, INTENSE adds code that will inspect the objects arriving and leaving the server, thus helping assessing steps 4 and 5 (*client communication* and *server communication*), respectively. We will explain this object inspection in detail in Section 4.4 and Section 4.5, which explain the steps 4 and 5, respectively.

After the code is pre-processed, our platform builds the necessary package for deployment, when required by the server platform (for instance, it compiles the code and builds a WAR file if the service is Java-based, which includes creating all necessary directories for the different uploaded files) and uploads the package to the necessary server by FTP, which was previously identified by the user. INTENSE then waits for correct deployment of the service, expecting a WSDL file to be produced within a configurable time period. If the WSDL is not produced in that time period, tests for that service cannot begin and cleanup follows. In this case, the *TestManager* is free to start the next pending job.

If the WSDL is not generated, we mark this as an *error*, as it is an event that prevents the execution of further inter-operation tests. Otherwise, if the WSDL is successfully produced, a remote machine, which will act as client for that server in the next inter-operation steps, is ordered (via RMI) to run the WS-I tests using the generated WSDL as input. The output of the WS-I tests is gathered and sent back to the test manager. At this point, these tests might produce some *warnings*, but will never prevent further inter-operation steps (i.e., will not generate a *errors*).

Client Artifact Generation

After passing the *service description generation* step, the *TestManager* commands the remote machine involved in the client-side tests (i.e., the machine that holds the user selected platform) to generate client artifacts. The output produced by that process (e.g., the wsimport tool, or wsconsume) in the remote machine is gathered and sent back to the *TestManager* that dispatches the message to the respective problem classifier that will understand if the tool failed, produced warnings, or was successful. Of course, this classification task is highly dependent on the tools being used, as tools fail in different ways, and their output can be quite diverse. Thus, in the worst case, we must provide a classifier for any new specific tool that is added to the system, which is essentially a function that receives text as input (i.e., the text that is the output of the tool) and classifies that text as an *Error*, *Warning*, or as *Correct*.

Client Artifact Compilation

In the case of platforms that require code compilation (e.g., Java, C#), the compilation tool that is required by the client-side framework to compile the code is used in this step. Again, this is something run remotely and the outcome is sent back to INTENSE to be checked for the presence of *warnings*, or *errors* (as in step 2).

Client Communication

This step is centered around having a way to understand if the information at the client and before passing through the client-side framework is equivalent to the one that is received at the service, i.e., after passing through the server-side framework. If the frameworks are able to inter-operate correctly, the information at both ends will be the same.

Understanding if the data is the same at both ends holds significant challenges, as we may have different programming languages involved, or simply different data types when using the same language. In this context, comparing the information at both sides holds specificities that exclude or make extremely difficult the application of an already established serialization mechanism. In the context of web services, and considering the specific goals involved in this step, the requirements for having **correct communication** are the following:

a) *The values sent by a client to a server must be the same at both ends of the interaction.* This implies that they are also placed in the right placeholders, i.e., the values are associated with the right variables, which means that we need to know values and be able to compare a given value in a client-side object with another value in the equivalent position in the server-side object.

b) *The data types of the variables that hold the values that will be sent to the server must be equivalent, although not necessarily the same*. If the inter-operation process goes well, we will find equivalent data types at both sides of the interaction. This means that, for instance, if we have a Java-based operation that accepts a *byte* as input argument, we will expect to see an *sbyte* being used in a C# .NET client to invoke the remote operation. This of course depends on the correct behavior of the client artifact generation tool, which will map the XML Schema data type to the specific platform data type and that, in the case of this example, will be *sbyte*.

c) *The object or set of objects involved in the invocation must have the same structure at both ends of the interaction.* We must be able to check if the structure of a specific complex object is the same at both ends, which implies that we also need to represent and process such information.

In addition to the comparison requirements abovementioned, this step requires that we create and execute a client that invokes a remote operation. The correctness of this step is evaluated with the help of hook code that we add to the client (placed immediately before the remote operation invocation) and that we have previously added (during the pre-processing of the service code) to the service (immediately after the operation arguments are received). The purpose of this hook code is simply to collect the information that is about to leave the client, or has just reached the server, for comparison purposes. Thus, this *Client Communication* step involves the execution of the following tasks, which we describe in further detail in the next paragraphs:

- **Task I:** Creation of invocation code that uses the client artifacts to invoke the remote operation and send the data to a *Comparison Service*.

- **Task II:** Verification of the correctness of the client-server communication.

Task I: Creating invocation code

Before executing the web service consumer, we need to create code, which, at runtime, will use the generated artifacts to invoke a remote operation. As this code must work for any service, INTENSE generates code that performs the following functions:

i. Inspects artifact objects to discover what is the operation to be invoked and its required arguments;

ii. Initializes service port objects that will allow invoking the operation;

iii. Inspects the artifacts to automatically fill the operation arguments with random values;

iv. Serializes and transmits all information regarding the operation inputs to the *Comparison Server* (please refer to the description of Task II for details);

v. Invokes the remote operation.

INTENSE locally generates code that performs all of the above functions. This code is specific for the client-side platform being used, and will be uploaded to the machine where step 3 was carried out to be compiled and executed. The client code is generated using a Velocity template, which is expanded at runtime with the right values to create functional code (as described in the the next paragraph). The decision of including this information in a template is due to the fact that we need to perform modifications whenever we intend to expand INTENSE to support further tools. Thus, we limit the number of changes in the source code by moving these aspects to a template that lives outside the code.

For Java-based clients, the generated code makes use of reflection, which means that it is prepared to handle any kind of service (i.e., a service using any number or type of arguments for its operations), as the code is dynamically inspected at runtime and the proper actions are taken, according to the structures found. In other mainstream languages similar mechanisms exist and can be used in the future, when the platform is extended.

Task II:Verifying communication correctness

In order to understand if the information at the client and server endpoints is the same, we designed a serialization mechanism that deep-serializes objects and transforms their basic constituents (e.g., integer values, float, boolean) into text while keeping information regarding object structure and data types. **At the client-side** (and also at the service-side as described next) the outgoing object information is sent in a series of *basic messages* to the *Comparison Service*, according to the following general conversation pattern. A first *basic message* includes the following main information: an identifier of the interoperability test (for correlation and verification purposes at the *Comparison Service*); information regarding the endpoint so that the *Comparison Service* can understand if it there is another endpoint to wait for or if it is allowed to proceed with the comparison; and most importantly, metadata regarding a set of N messages that will follow and that describe each of the values that compose the operation arguments. Using this last part of the information, the *Comparison Server* performs the corresponding number of reads, according to the information received in the first *basic message* and acknowledges reception.

The abovementioned exchange of messages is a conversation, for which we provide minimal delivery guarantees. Each *basic message* actually initiates with 2 bytes. These 2 bytes express the number of 8-bit characters, which will follow up, so that the other side is able to read the exact quantity or fail. This is the same contract used in the Java language in the methods *writeUTF* and *readUTF* of the data stream classes and can be implemented in any language that supports the sockets abstraction. Also, we are not interested at the moment in a more complex protocol, which would allow us to recover from failures (i.e., logging and re-sending lost messages, using negative acknowledgments, etc.). We are simply interested in understanding if the information is being received as a whole or not. If there is a problem in the communication, INTENSE interrupts the interoperability test.

As mentioned, each of the *basic messages* that follow the first one hold information regarding a particular parameter. The information is used to analyze the three correctness requirements described in the beginning of Section 4.4 and contains data regarding: i) the structure of the operation argument as perceived at the client side, as each particular variable can be found at several degrees of depth (e.g., a String wrapped in a custom class, in turn part of a List); ii) the type of the argument being analyzed (*java.lang.String*); iii) the value for that particular argument (which was generated by INTENSE).

Some objects hold special characteristics and this is the case of Lists and Maps. In the case of Lists, each member of the list is simply viewed as an attribute of this container and is given a unique numeric name. In the case of Maps, each map entry is viewed as an attribute of this container and given a name that is equal to the map key. The corresponding value is naturally the value that is associated with that particular key.

At the service-side, if the remote operation is invoked successfully, i.e., if the SOAP message sent by the client is delivered at the service, then we are able to get the arguments and also serialize them and send them to the *Comparison Server*. The difference is that, only the eligible arguments of that operation are sent to the Comparison Server. For instance, the eligible arguments are the ones that, in the case of the Java language have corresponding getters and are in fact announced as arguments of the operation (i.e., they respect the JavaBeans specification). As an example, if a Java service accepts a complex type as argument we may find fields in the class that are not present in the announced interface. These fields will not be serialized, as they are not involved in the process.

At the Comparison Server we will have an application-level representation of the operation arguments at the client-side (i.e., before they were serialized to SOAP) and also the application level representation of the arguments at the service-side (i.e., after being transmitted in a SOAP envelope and after being deserialized from SOAP to a set of application-level objects). This pair of information is scanned in the following three phases:

i. We verify if the structure of the objects is the same on the two sides. Thus, we perform a depth analysis on the nodes that contain values, and we must find the same relative depth at all points, otherwise, the structure of the objects at the client-side is different from the one found at the server-side.

ii. We verify the data types are equivalent (but not necessarily the same). For this we use the established language mapping used by Google Protocol Buffers [19], which already specifies, for a few programming languages, a mapping between data types (e.g., a double in C# is equivalent to a Float in Ruby).

iii. We check if the values present at each side, at each corresponding position, are the same.

If any of the above scans fails (e.g., a value is missing on one of the sides of the interaction), the inter-operation process has failed. In the end, the result of the comparison will reach INTENSE and will include the above three levels of detail, so that the tester can take the proper actions.

Server Communication

The server communication step is similar to the previous one. In this case, there is a single object (i.e., the operation response) involved in the communication, however it can be complex and contain various sub-objects. Similar to the client code, this object is also instantiated by INTENSE, its constituents are recursively filled with random values, and it is serialized and sent to the Comparison Server. After this, it will follow the regular path and will be converted to SOAP, received at the client-side framework, and converted from SOAP back to a programming language object. When it reaches the client code, this object will also be serialized and sent to the Comparison server, which will perform the same comparison functions as described in the Client Communication step. If some difference is detected (i.e., in structure, types or values) then inter-operation process has failed in this step. After the inter-operation tests conclude (successfully or not), cleanup procedures follow, so that a new interoperability test can begin from a clean state.

EXPERIMENTAL EVALUATION

In this section we describe the experimental evaluation, which was designed essentially to verify and validate the tool's capabilities in detecting interoperability issues. The next sections describe the scenarios used and main results obtained during the experiments.

Experimental Setup

We selected the following set of well-known platforms for deploying and invoking web services:

- **Server-side:** Metro 2.3.1 on the Glassfish 4.1 server; JBossWS 5.0.0 on Wildfly 9.0.1.Final
- **Client-side:** Metro 2.2.9; JBossWS 5.0.0; Axis2 1.6.4

We used every possible combination between the client-side platforms and server-side during the interoperability tests, thus resulting in a total of 6 combinations. Regarding the services, we considered the following scenarios (the whole set of services is available at [20]]).

- **Synthetic services:** A set of 10 custom services, created to provide initial different test cases to exercise the tool. The services range from simple cases (a single and simple argument for service operation) to more complex cases (service operations accepting complex parameters involving lists, maps, and custom complex objects with nested complex objects).
- **Realistic services:** A set of 10 web services specified by the TPC-App benchmark [21]. The goal here is simply to demonstrate the application of INTENSE to a realistic scenario and further exercise the tool and disclose any possible bug in the tool.
- **Real services:** Selected cases of 80 web services publicly available on the Internet. In this case the goal is to, based on the authors results in [1], show that INTENSE is based to detect known web service framework bugs and at the same time does not signal inexistent problems.
- **Faulty services:** The most complex service of the synthetic services was selected to be used with a fault injector. This fault injector applies three different types of faults into the data being exchanged between client and server and vice-versa. Thus, it emulates 10 different cases (5 problems detectable at the server-side plus 5 at the client-side). The fault types are: i) structure changing faults; ii) data type changing faults; iii) value mutation faults (which can be value replacement, value addition, or value removal).

We executed the inter-operation tests for the above sets of services (Axis2 was not used to test the communication steps), resulting in a total of 420 inter-operation tests, which are discussed in the following section.

Results

Table 1 summarizes the results of the experimental evaluation, showing the results per service set and per framework used. The results for each of the inter-operation levels (regarding each combination service set / framework) are further detailed. Note that we do not show the results per server-side framework, as they were found to be the same. In the table, *w* means one or more *warnings* were found; *e* refers to one or more *errors* being found; and a dash indicates that the tests were not run for that level.

As we can see in Table 1, INTENSE was able to detect different problems in all steps of the inter-operation process, with exception of the deployment step where we only uncovered warnings (resulting from the execution of the WS-I tests). In fact, authors in [2] emphasize that the deployment in current major platforms is quite robust, failing whenever the rules are not fulfilled (e.g., a class not being provided, or a wrong structure in the deployment file), but if the deployment complies with the basic rules it's very difficult to find a fatal problem. Obviously we also tried to deploy undeployable services, which served to verify the correctness of INTENSE. Anyway, we did not manage to get an error in step 1,

Table 1. Results of the interoperability tests.

Sets	Metro					JBossWS					Axis2				
	I	II	III	IV	V	I	II	III	IV	V	I	II	III	IV	V
Synthetic													*w*	–	–
Realistic													*w*	–	–
Real	*w*	*e*		–	–	*w*	*e*		–	–	*w*	*e*	*e*	–	–
Faulty				*e*	*e*				*e*	*e*			*w*	–	–

using a service that complies with the deployment rules (e.g., having a java class annotated with *@WebService*).

Regarding the **synthetic set**, no error was found, in any of the five inter-operation levels, probably due to the small size of the experiments, but anyway the set was useful for verifying INTENSE basic functionalities. However, we did observe *warnings* using Axis2, related with the absence of typed collections during the compilation process, for all services in this set.

The results regarding the **realistic set** were similar to the above. Clearly, we were expecting detecting more problems in the Real services, as they include services built on many different platforms, and also obviously in the Faulty services. In the case of the **real services**, we selected cases from the set used by the authors in [1], for validating our tools results. 12.5% of the services in this set were selected for being known to not generate any kind of problem [1], which was confirmed by our tool. Of the remaining, we highlight one case resulting in an *error* in Level II with JBossWS (which however did not fail with Metro or Axis2). We also highlight 10 services failing with Metro and JBossWS but not failing with Axis2 in Level II. In 9 of these services a compilation problem was detected when using Axis2 in Level III. Also present in this more heterogeneous set, were some WS-I warnings, which we used to fine-tune INTENSE warning detection capabilities. Obviously, when using the real services set we are unable to reach steps 4 or 5 (marked with a dash in Table 1), as we do not have access to the service code or infrastructure.

Finally, we used the faulty service jointly with a fault injector that emulates a framework holding bugs. As we were not using a weakly statically typed programming language in these experiments, we were not able to change the structure or object types, although we simulated these two situations by changing the information travelling to the comparison server, which handled these cases properly. However, we managed to detect three kinds of problems in the communication, even when the communication is slightly different due to the use of the different frameworks. Thus, we were able disclose errors due to missing values, extra values and modified values.

CONCLUSION

In this paper we presented INTENSE, a web-based testing tool that allows testing a web service against multiple client platforms. The tool can be used without the presence of the code of the service being tested (i.e., it only requires a WSDL file to perform tests), although it can perform extended communication tests if the service source code (i.e., the service interface code) is provided. INTENSE was used to test 4 sets of services deployed on very popular implementations of the web services stack – Metro, the JAX-WS reference implementation on Glassfish, and JBossWS on the WildFly server. Tests were run against the client-side implementation of Metro, JBossWS, and also against Axis2. The problems disclosed during the experiments, including problems introduced by our custom fault injector, served to illustrated the utility of our testing service and its problem detection capabilities. Without this type of testing, many of these problems usually pass unaware to developers, only to be found at runtime, when a particular client interacts with the service. Future work includes extending INTENSE to support further web service frameworks, which will allow the community to take further advantage of this testing service.

ACKNOWLEDGMENTS

This work has been partially supported by the Project DEsign, Verification and VAlidation of large-scale, dynamic Service SystEmS (DEVASSES), Marie Curie International Research Staff Exchange Scheme (IRSES) number 612569, within the context of the EU Seventh Framework Programme (FP7).

REFERENCES

[1] I. A. Elia, N. Laranjeiro, and M. Vieira, “A Field Perspective on the Interoperability of Web Services,” in *11th IEEE International Conference on Services Computing (SCC 2014)*, Anchorage, Alaska, USA, 2014.

[2] I. A. Elia, N. Laranjeiro, and M. Vieira, “Understanding Interoperability Issues of Web Service Frameworks,” in *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*, Atlanta, Georgia, USA, 2014.

[3] I. A. Elia, N. Laranjeiro, and M. Vieira, “Test-based Interoperability Certification for Web Services,” in *The 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2015)*, Rio de Janeiro, Brazil, 2015.

[4] Web Services Interoperability Organization (WS-I), “Web Services Interoperability Organization (WS-I),” 2002. [Online]. Available: http://www.ws-i.org/. [Accessed: 16-Sep-2008].

[5] C. Ferris, A. Karmarkar, P. Yendluri, K. Ballinger, D. Ehnebuske, M. Gudgin, C. Liu, and M. Nottingham, “WS-I Basic Profile - Version 1.2,” 24-Oct-2007. [Online]. Available: http://www.ws-i.org/Profiles/BasicProfile-1_2(WGAD).html. [Accessed: 14-Feb-2008].

[6] Web Services Interoperabily Organization (WS-I), “Deliverables - Basic Profile Working Group,” 2014. [Online]. Available: http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile. [Accessed: 13-Jan-2014].

[7] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, “Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI,” *IEEE Internet Computing*, vol. 6, no. 2, pp. 86–93, 2002.

[8] IEEE, “IEEE Standards Glossary,” 2015. [Online]. Available: https://www.ieee.org/education_careers/education/standards/standards_glossary.html. [Accessed: 29-Sep-2015].

[9] K. M. Senthil Kumar, A. S. Das, and S. Padmanabhuni, “WS-I Basic Profile: a practitioner’s view,” in *IEEE International Conference on Web Services, 2004. Proceedings*, 2004, pp. 17–24.

[10] K. Ballinger, D. Ehnebuske, M. Gudgin, M. Nottingham, and P. Yendluri, “WS-I Basic Profile - Version 1.0,” 2004. [Online]. Available: http://www.ws-i.org/Profiles/BasicProfile-1.0.html. [Accessed: 06-Dec-2013].

[11] P. Ramsokul and A. Sowmya, “A Sniffer Based Approach to WS Protocols Conformance Checking,” in *The Fifth International Symposium on Parallel and Distributed Computing, 2006. ISPDC ’06*, 2006, pp. 58–65.

[12] C. Pautasso, O. Zimmermann, and F. Leymann, “Restful Web Services vs. ‘Big’’ Web Services: Making the Right Architectural Decision,” in *Proceedings of the 17th International Conference on World Wide Web*, New York, NY, USA, 2008, pp. 805–814.

[13] A. Bertolino and A. Polini, “The audition framework for testing Web services interoperability,” in *31st EUROMICRO Conference on Software Engineering and Advanced Applications, 2005*, 2005, pp. 134–142.

[14] E. Damiani, N. El Ioini, A. Sillitti, and G. Succi, “WS-Certificate,” in *2009 World Conference on Services - I*, 2009, pp. 637–644.

[15] M. Anisetti, C. A. Ardagna, and E. Damiani, “Fine-Grained Modeling of Web Services for Test-Based Security Certification,” in *2011 IEEE International Conference on Services Computing (SCC)*, 2011, pp. 456–463.

[16] K. Ballinger, D. Ehnebuske, C. Ferris, M. Gudgin, C. K. Liu, M. Nottingham, and P. Yendluri, “WS-I Basic Profile - Version 1.1,” *ProfilesBasicProfile-11 Html April*, 2006.

[17] CrossCheck Networks, “SOAPSonar - Advanced Client Simulation for Service Testing.” [Online]. Available: http://www.crosschecknet.com/products/soapsonar.php. [Accessed: 13-Jan-2014].

[18] Eviware, “soapUI,” 2011. [Online]. Available: http://www.soapui.org/. [Accessed: 04-Jul-2011].

[19] Google Inc., “Protocol Buffers Language Guide (proto3),” 2015. [Online]. Available: https://developers.google.com/protocol-buffers/docs/proto3. [Accessed: 01-Dec-2015].

[20] “witws_tests.zip,” *Google Docs*. [Online]. Available: https://drive.google.com/file/d/0B4SrxuwR8GJ2V3VtcHpYSE1jZlU/view?usp=embed_facebook. [Accessed: 29-Jan-2016].

[21] Transaction Processing Performance Council, “TPC Benchmark App (Application Server) Standard Specification, Version 1.3,” 2008. [Online]. Available: http://www.tpc.org/tpc_app/. [Accessed: 05-Jul-2008].