



Utrecht University  
Software Architecture

-

## **OpenLaszlo 4.9.0**

-

Kevin van Ingen (3561593)  
Sjors Otten (3558010)  
Mark Rouhof (3553507)  
Robert Vroon (3440516)

-

12-01-2011

## Table of contents

- Introduction..... 3
- Architectural description..... 4
  - Logical view ..... 4
    - Server..... 4
    - Client..... 5
  - Process view ..... 7
  - Physical view ..... 8
  - Development view ..... 9
  - Scenario view..... 10
- Quality aspects ..... 13
  - General quality aspects addressed by OpenLaszlo ..... 14
    - Quality characteristics on functionality..... 14
    - Quality characteristics on portability ..... 14
- Tradeoffs in client-server information exchange ..... 15
  - Advantages and disadvantages of connection strategy..... 16
  - Tradeoffs in proxy settings for OpenLaszlo applications ..... 16
    - Use-case scenario ..... 16
    - Testing proxied and non-proxied communication ..... 18
    - Results ..... 18
- Comparison with two alternatives ..... 19
  - Introducing OpenLaszlo versions and Competitors ..... 19
  - Choosing an comparative version and competitor ..... 21
  - Laszlo Presentation Server Version 2.0 ..... 24
  - Expanz: Silverlight CoreCLR security model ..... 25
- Conclusion and discussion..... 27
- References..... 28
- Appendix A ..... 29
  - Answers to questions after the literature assignment..... 29

## Introduction

A large-scale software project cannot be built without first considering which features it is supposed to have and how these should be implemented. This will result in a specific software architecture, which Clements (1995) describes as the summary result of a set of decisions. These decisions are influenced by several events during the development process of the software, and may occur on different levels (project-related influences, organization-related influences, architecture-related influences). These decisions, resulting in software architecture also result in tradeoffs between quality aspects of a software product. Quality according to the ISO 8402 standard and Bevan (1999) is defined as ‘the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs’. This definition is too generic to directly adopt and apply to a software product quality measurement technique. Therefore the ISO/IEC 9126-standard was developed from a user point-of-view perspective comprising quality aspects such as functionality, reliability, usability, efficiency, maintainability and portability. Security is originally a sub quality aspect of functionality but Jung et al. (2005) stress the fact that this sub quality aspect should be promoted to an additional quality aspect due to the importance of privacy and protection of personal data in the all more increasing world of digital information exchange.

In this paper we are trying to give an architectural overview of OpenLaszlo and identify suspected tradeoffs between a selection of these quality aspects and underlying sub quality aspects for the software development framework OpenLaszlo 4.9.0. OpenLaszlo is a develop-once-run-anywhere development framework and is platform independent, thereby allowing it to run on multiple operating system environments. One can develop an application within the OpenLaszlo development framework on a Linux operating system and deploy it to a Microsoft Windows operating system without any required adjustments from the developer.

In section two of this paper we describe OpenLaszlo’s architecture. Section 3 comprises an analysis of the suspected tradeoffs method and the objects of our research. Within section 4 a comparison with two similar software products is performed. Finally, section 5 comprises a conclusion on OpenLaszlo and its suspected tradeoffs followed by a discussion.

## Architectural description

### Logical view

OpenLaszlo is a big system and contains a lot of classes of architectural significance. That is the reason for making a top level class diagram for the proxied version of OpenLaszlo. The diagram contains 14 class categories.

### Server

The server contains 4 main classes: interface compiler, media transcoder, data manager and the cache.

#### Interface compiler

The Interface Compiler consists of a LZX Tag Compiler, and a Script Compiler. The Interface Compiler invokes the Media Compiler and the Data Manager to compile media and data sources that are baked into the application.

The LZX tag and script compilers convert LZX application description tags and JavaScript into executable (swf) byte code for transmission to the OpenLaszlo client. This code is placed into the cache and then sent to the client. Depending on how the application is invoked, it is transmitted either as a SWF file, or as an HTML file with an embedded SWF object.

The Media Transcoder converts media assets into a single format for rendering by OpenLaszlo's target client rendering engine. Therefore the OpenLaszlo application is able to present supported media types in a unified manner on a single canvas, without the distraction of multiple helper applications or supplemental playback software.

The Media Transcoder automatically transcodes the following media types: JPEG, GIF, PNG, MP3, TrueType, and SWF (art/animation only).

#### Data Manager

The Data Manager is comprised of a data compiler that converts all data into a compressed binary format readable by OpenLaszlo applications and data connectors that enable OpenLaszlo applications to retrieve data via XML/HTTP. OpenLaszlo applications can therefore interface across the network with databases, XML Web Services, and Web-server based files or executables.

#### Cache

The cache contains the most recently compiled version of any application. The first time an OpenLaszlo application is requested, it is compiled and the resultant SWF file is sent to the client. A copy is also cached on the server, so that subsequent requests do not require waiting for the compilation.

## Client

The client contains also 4 main classes: event system, data loader / binder, layout animation system and services system.

### Event system

The event system recognizes and handles application events such as user mouse clicks or server data pushes. Relative to conventional Web implementations, OpenLaszlo applications typically reduce the processing load placed on the host server, by enabling tasks such as client-side sorting, processing, validation, and dynamic display across all application states.

### Data Loader/Binder

The data loader serves as a data traffic director, accepting data streams across the network from the OpenLaszlo Server and binding data to appropriate visual display elements such as text fields, forms, and menu items.

### Layout Animation System

This component enables you to build a dynamic application interface with minimal programming. It allows you to position a variable number of interface elements using either relative positioning or absolute pixel positioning. With the animation algorithms, screen interface updates are rendered in a visually continuous manner, clearly communicating the application's state changes to the user.

### Services System

The OpenLaszlo runtime includes support for timers, sound and modal dialogs.

Below figure 1 depicts the logical view of OpenLaszlo's architecture (OpenLaszlo Architecture, 2011)

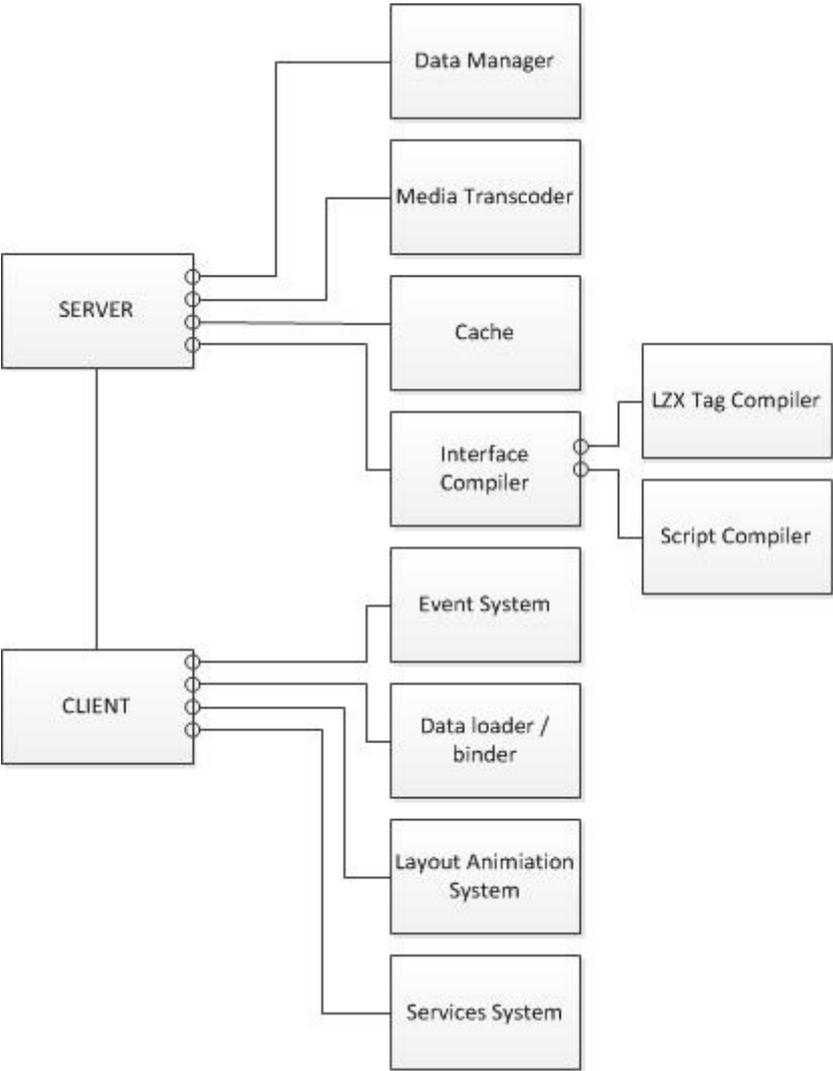


Figure 1 | OpenLaszlo – Logical view

## Process view

In compliance with the logical view, provided in the previous section, the following process view is constructed, which is a high level view of the different processes in OpenLaszlo.

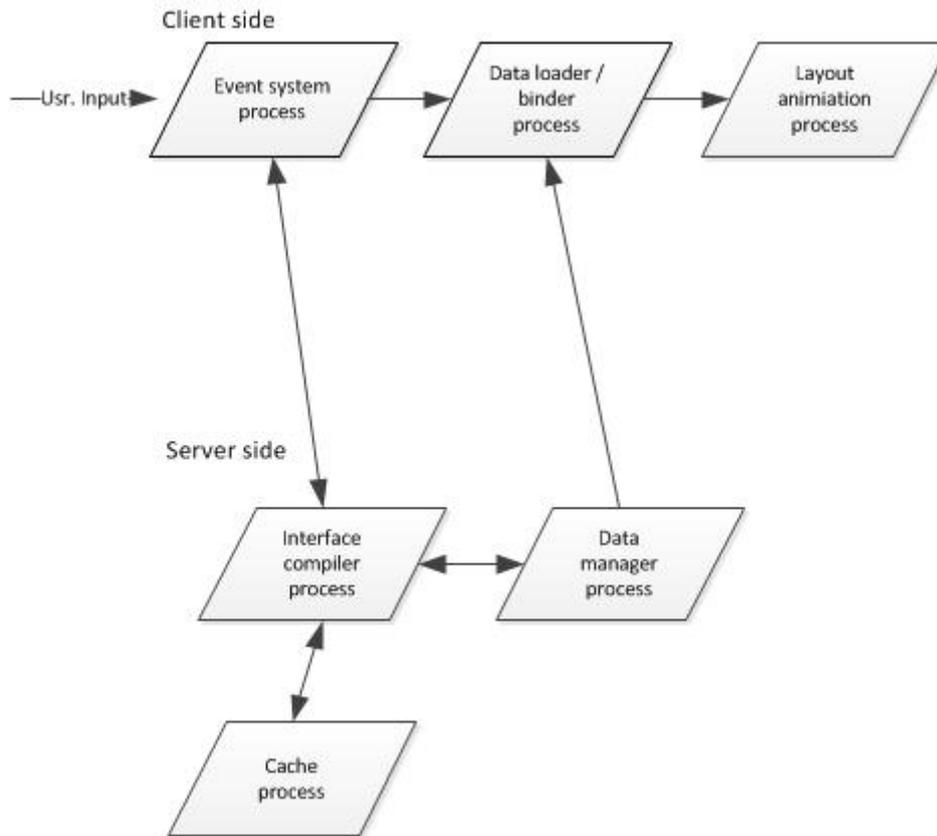


Figure 2 | OpenLaszlo - Process view

All applications are handled by the *event system process* due to requests or commands originating from the user. When a user requests an application the request is sent via the *event system process* to the *interface compiler process*. The *interface compiler process* on its turn communicates with the *cache process* to verify if there is already a compiled application of the latest version requested by the client. If so, that executable is sent back to the client in a SWF-format. If not, the *interface compiler process* locates the LZX-script and compiles all required components including media, code, data and data connectors into a single executable and sends it back to the client in a SWF-format. At the client-side the *data loader/binder process* handles all incoming data streams from the server and binds them to the correct user interfaces. It also provides the client-side with an option to retrieve data from external sources via a proxied- / non-proxied-connection with the included data connectors in the executable. If the latter is the case the *data manager process* is invoked and handles the incoming request for data originating from the *event system process*. The *data manager process* contacts the appropriate data sources, retrieves the data and pushes it to the client in XML-format and the *data loader / binder process* binds it to the correct user interfaces. The *layout animation process* allows the client to build a dynamic application interface with minimal programming. With the animation algorithms, screen interface updates are rendered in a visually continuous manner, clearly communicating the application's state changes to the user.

### Physical view

The physical view of OpenLaszlo describes the mappings of the software onto the hardware and reflects its distributed aspect. Due to the client-server pattern on which OpenLaszlo is based, per definition in a production environment there are two distinct nodes to be distinguished from one another (Client, Server). Multiple servers can be in place and function as a backup / failover in case the primary server-node fails.

Next to that two options are available for deploying applications namely, SOLO-deployment and J2EE deployment (OpenLaszlo server deployment, proxied). Figure 3 depicts the deployment options next to each other.

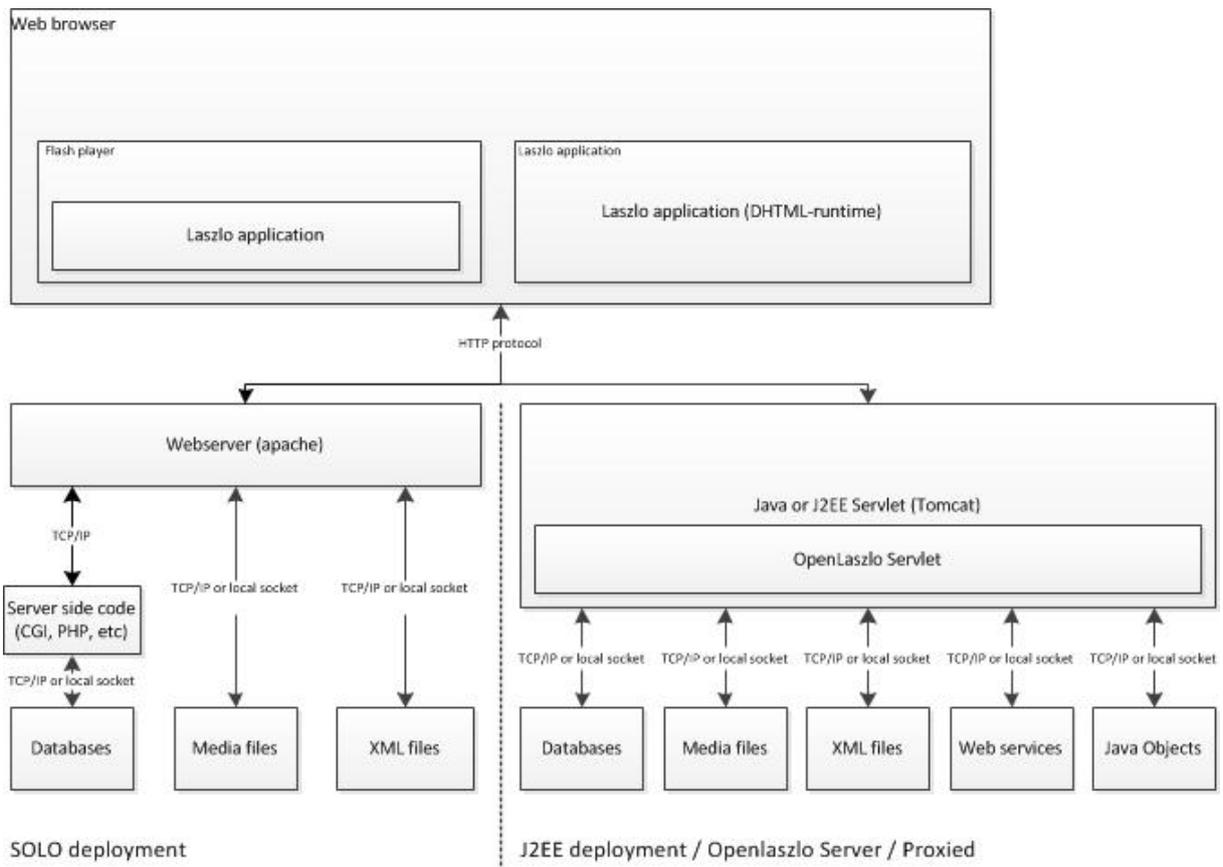


Figure 3 | OpenLaszlo - Physical view

Communication between server and client is done by using the HTTP-protocol. From server to external data sources the TCP/IP-protocol is used in case data is stored on a different physical machine or network. A local socket is used when the data resides on the same physical machine.

## Development view

The development architecture focuses on software module organization at the software development environment. The software is packaged/divided in small chunks (program libraries or subsystems) that can be developed by one or a small number of developers. The subsystems are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it. Figure 4 depicts the OpenLaszlo development view.

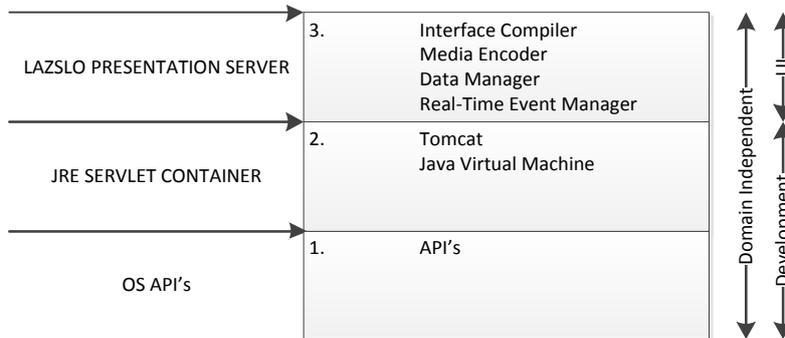


Figure 4 | OpenLaszlo - Development View

OpenLaszlo comprises three layers; OS API's, JRE Servlet Container and the Laszlo Presentation Server. A connection is made between the OS API's and the JRE Servlet Container where the OS API-layer provides an interface to the JRE Servlet Container allowing the latter to append or invoke commands and services of the specific OS. The JRE Servlet Container is used to execute the compiled executable within OpenLaszlo. JRE Servlet Container provides an interface to the Laszlo Presentation Server allowing the latter to present the application to the end-user.

## Scenario view

In order to validate the software architecture of OpenLaszlo we developed a use case which guides us through the whole process of application development until deployment to the client-side and additional data retrieval. The use case comprises an online weather application. The use case has the following script:

1. Developer develops a new version of an online weather application and stores it on the OpenLaszlo server as weather.lzx.
  - a. Weather.lzx is send from the client towards the server
2. End-user invokes browser and types in the URL of the weather application and hits enter
3. The URL-request is send to the OpenLaszlo server and the server locates the weather.lzx file.
4. The OpenLaszlo server also checks whether if the same version is already available in its cache as an executable to prevent additional load to the system. If so, the OpenLaszlo server continues to step 6, If not, step 5 is initiated.
5. The OpenLaszlo server compiles the requested application from its source code (weather.lzx) and adds all required media, data and data connectors.
6. Executable is delivered to the client.
  - a. Executable is send from server to client
7. Browser displays the requested application (in SWF or DHTML).
8. End-user enters zipcode of his area and hits enter.
  - a. Zipcode is send from the client to the OpenLaszlo server with the request to retrieve additional data.
9. OpenLaszlo Server invokes appropriate XML-data connector in order to retrieve the appropriate data corresponding to the provided variable (zip code).
10. OpenLaszlo Server sends the XML-data to the client.
  - a. XML-data send from server to client
11. The Client binds the received XML-data to the appropriate user interfaces and updates the screen elements.

Figure 5 depicts a schematic view of the use case as was written in the above paragraph.

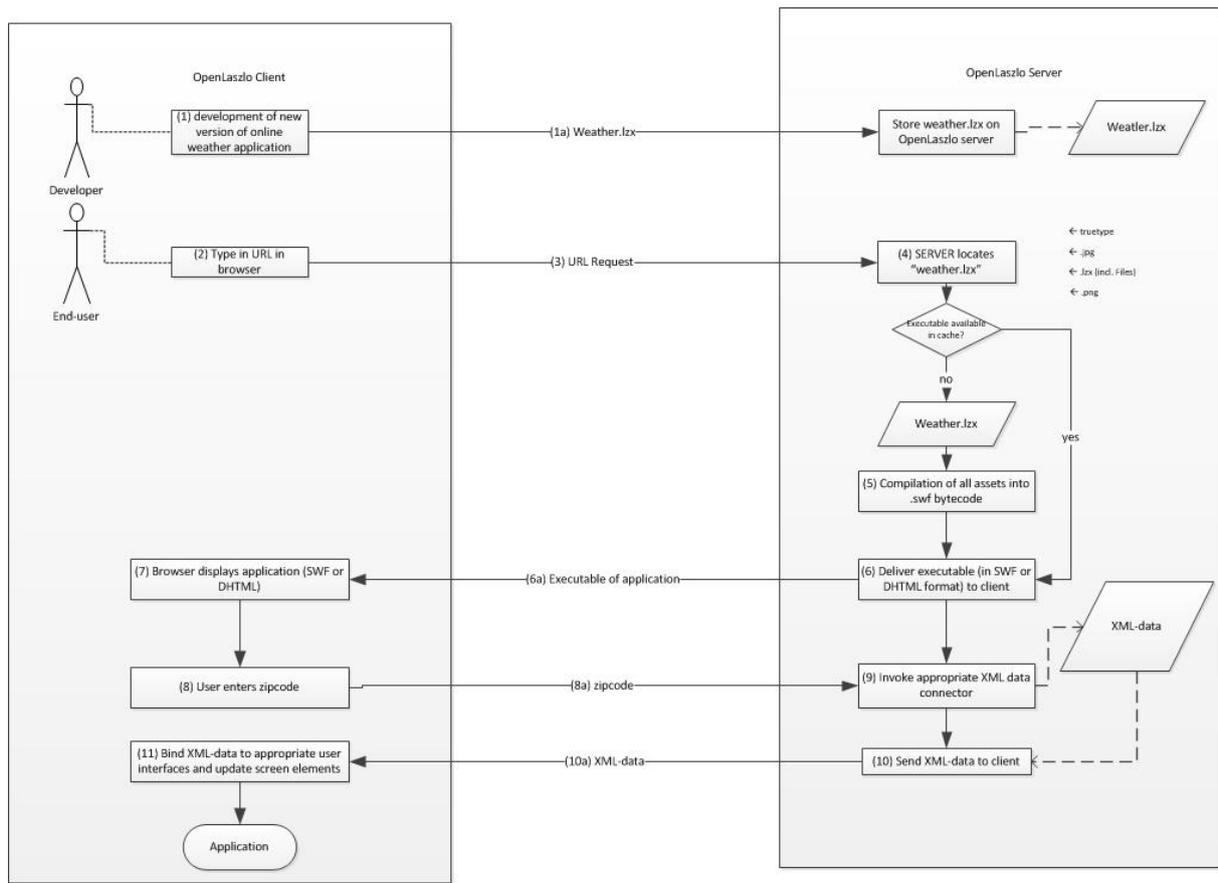


Figure 5 | OpenLaszlo - Scenario view - Use case

For validation purposes of the software architecture we elaborate on how the architecture will support the use case depicted in figure 5 and which classes from the logical view are being invoked. Concerning step 1 and 2 the *event system* is being used on the client side to handle all the input provided by the developer and end-user. The event system passes the input to the correct classes on the server-side. In follow up of step 2, step 3 is initiated and a URL-request is send towards the server. The server locates 'weather.lzx' in step 4 by looking if there is an executable of the latest version is available in the *cache*. If not, step 5 is initiated and the *interface compiler* compiles a byte code of all assets by using the LZX tag and script compiler. Eventually on the server side the server sends the executable towards the client (step 6) and the client presents the application to the end-user by using the *layout animation system* and binds the appropriate data to the appropriate interface elements via the *databinder/loader* (step 7).

In step 8 the end-user wishes to retrieve up-to-date weather information on his location, therefore he enters his zip code and pushes enter. The *event system* on its turn sends the input to the appropriate class at the server-side called the *data manager* (step 8, 8a). On the server-side the *data manager* invokes the appropriate data connector in order to retrieve the data requested by the end-user on the client-side using its zip code as a variable in the lookup-query (step 9). From there on out the data is being sent back from the server to the client (step 10, 10a). On the client-side the *databinder/loader* will load the received XML-data and binds it to the appropriate visual elements in the user interface.

The process view has been left out of the elaboration due to the fact that it is a main abstraction of the logical view and thereby making it insignificant.

Regarding the physical view of OpenLaszlo and corresponding deployment diagram, the weather application asks the OpenLaszlo server to communicate with the external datasource and act as a mediator between them. The weather application thereby qualifies as a proxied-webapplication and utilizes the full potential of OpenLaszlo. Another possibility would be to port the weather application to another server-side scripting language (i.e. PHP, CGI) and use a non-proxied connection, where the application connects directly to the datasource. If done so, the advantage of OpenLaszlo will be lost and additional security issues may manifest and possible portability of the application may be lost.

## Quality aspects

In this section quality aspects of the OpenLaszlo framework are addressed using the ISO/IEC 9126 standard (Jung, Kim, 2004). An ISO/IEC 9126 quality model is defined by means of general characteristics of software, which are further refined into sub characteristics, which in turn are decomposed into attributes. The ISO/IEC 9126 is currently one of the most widespread quality standards (Marco, C. Quer, 2004). The list of ISO/IEC 9126 software quality characteristics and sub characteristics is depicted in **Error! Reference source not found..**

Characteristics	Sub characteristics
Functionality	Suitability
	Accuracy
	Interoperability
	Security
	Functionality Compliance
Reliability	Maturity
	Fault Tolerance
	Recoverability
	Reliability Compliance
Usability	Understandability
	Learnability
	Operability
	Attractiveness
	Usability compliance
Efficiency	Time behavior
	Resource utilization
	Efficiency compliance
Maintainability	Analyzability
	Changeability
	Stability
	Testability
	Maintainability compliance
Portability	Adaptability
	Installability
	Co-existence
	Replaceability
	Portability compliance

Table 1 | ISO/IEC 9126 standard for software product quality attributes

## General quality aspects addressed by OpenLaszlo

### Quality characteristics on functionality

- **Security:** The OpenLaszlo server uses J2EE container managed authentication to manage the storage of compiled application on the server.
- **Security:** Security of client-server communication on two levels. OpenLaszlo support SSL connection over HTTPS. When HTTPS is not possible OpenLaszlo has data encryption/decryption on the application level to ensure a save connection to the server.
- **Security:** There is proxy support for retrieving external data into an OpenLaszlo client. In this way the only communication coming out of the client is send to the OpenLaszlo server. This enables the possibility to exert control over client communication on the TCP level through a firewall.
- **Interoperability:** OpenLaszlo support multiple ways of incorporating information into client applications. OpenLaszlo clients support the following communication protocols: XML-RPC, JavaRPC, and SOAP.
- **Interoperability:** OpenLaszlo is designed be a generic framework that deploys clients for multiple platforms. Platforms supported are Shockwave 8, Shockwave 10 and DHTML.

### Quality characteristics on portability

- **Installability:** OpenLaszlo server is compliant to the J2EE servlet standard and therefore should run on every J2EE compliant server.

## Tradeoffs in client-server information exchange

This section discusses a tradeoff found in the OpenLaszlo framework in the responsiveness of client-server communication. Clients are rich-interface front-end applications developed in the OpenLaszlo framework that are deployed to the user.

OpenLaszlo support multiple ways of incorporating information into client applications. Information can be hardcoded into a client or loaded from external sources. This means that the client itself does not have to contain the information it processes at the moment it deploys. It can obtain this after deployment. In this paper this is called, external information. This implicates that when deploying the application, a method for information retrieval or exchange is built into the client application.

When a client makes use of external information an OpenLaszlo application architect has to decide about the communication method which could be used. OpenLaszlo had different remote procedure call – RPC in short – implementations. A common method for exchange information is the Simple Object Access Protocol (SOAP). SOAP is based on the functionality provided by the generic XML-RPC API. The third information exchange standard is JavaRPC which is built on XML-RPC as well.

When a decision about a protocol is made the architect should reflect on the strategy of connectivity to the external source. OpenLaszlo support two strategies of dealing with this: proxied and non-proxied retrieval. In a proxied environment the OpenLaszlo client makes a connection to a OpenLaszlo server with a request for external information. The OpenLaszlo server requests this information and returns it to the client. In a non-proxied environment the OpenLaszlo client performs requests to external information by itself. In a non-proxied environment there is no need for a OpenLaszlo server at runtime, only for development. The two different ways of retrieving external information are depicted in **Error! Reference source not found..**

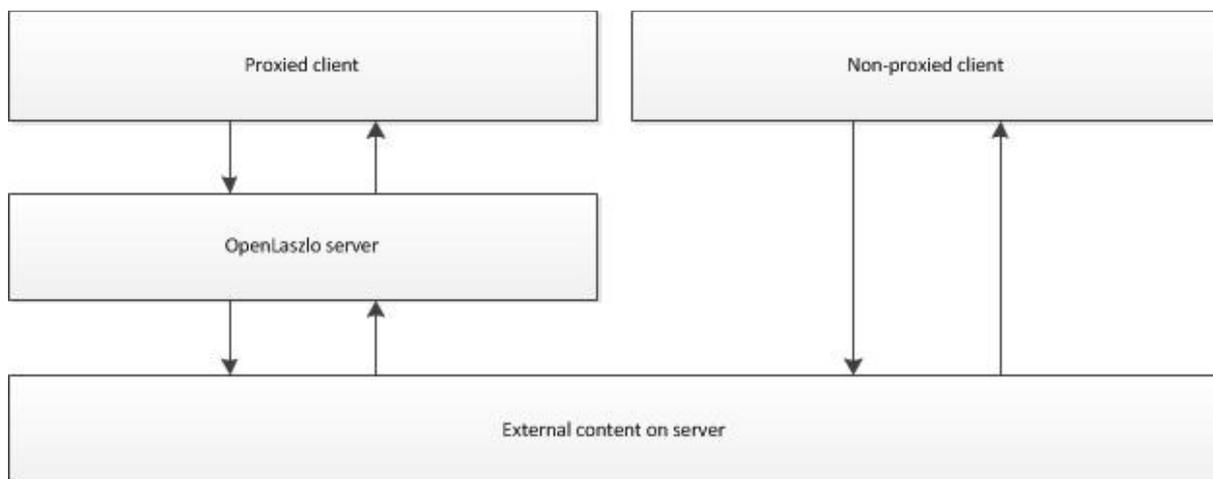


Figure 6 | OpenLaszlo connectivity strategy

Each OpenLaszlo application had one or more possibly nested canvas declarations. Each canvas can be seen as an application component. OpenLaszlo gives developers to declare the utilization of a proxied connection at the level of a canvas. It’s even possible to proxy only a part of the application.

On the code-level this is done in using the proxied attribute of the canvas tag. A code excerpt of this is depicted in Figure . In this case the use of a proxied connection is set to true. This can also be false or inherit. Inherit uses parent proxy settings or when unavailable the default unproxied setting.

```
<canvas proxied="true">
```

Figure 7 | Code excerpt of a proxy declaration

The choice between deployment methods results in a tradeoff between functionality security and efficiency of resource utilization according to the ISO/IEC 9126 quality model (Jung, Kim, 2004).

## Advantages and disadvantages of connection strategy

The OpenLaszlo framework support proxied and non-proxied connections as equal valid options.

### Advantages for non-proxied connection

- Non-proxied connections do not stress any load on the OpenLaszlo server.
- The OpenLaszlo server does not have to be available at runtime. Only in the development phase.
- Precompilation of the entire client application is possible for non-proxied connection.
- OpenLaszlo applications can be compiled from the command line. In this case there is no need for a J2EE compatible webserver. A lightweight webserver like Apache to distribute the OpenLaszlo client to its users will suffice.

### Advantage for proxied connection

- Proxied applications can handle browser inconsistencies. OpenLaszlo can dynamically load browser specific code. This means that behavior of some components is specified in general but for some components browser specific enhancements can be declared.
- Proxied applications can be secured on a low level using HTTPS and SSL or on the data level using encryption for sensitive data like user credentials.
- Proxied applications need only one address to access for remote information retrieval. System administrators can create a firewall around all other communication for security reasons.
- Support for on the fly media transcoding.

## Tradeoffs in proxy settings for OpenLaszlo applications

### Use-case scenario

An OpenLaszlo book selling application retrieves information about the contents of books from Amazon. The client uses the SOAP implementation to access a webservice exposed by Amazon. This information is used in the application so potential buyers can review information about books before buying the book. In the default non-proxied situation the client performs requests to the Amazon webservice directly. After a while the company exploiting this online shop decides that it's important to know what information is requested and decides to use the proxy method for communication

through the OpenLaszlo server. The use-case depicted in **Error! Reference source not found.** shows the different communication situations.

- Non-proxied version:
1. A user performs a request for books from the catalogue of the selling company
  2. A user requests the detail page of a book
  3. The OpenLaszlo client performs a request the Amazon webservice for the details of a book.
  4. The OpenLaszlo clients presents the book details to the user
- Proxied version:
1. A user performs a request for books from the catalogue of the selling company
  2. A user requests the detail page of a book
  3. The OpenLaszlo client performs a request to the OpenLaszlo server for the details of a book.
  4. The OpenLaszlo server requests book details from the Amazon webservice;
  5. The OpenLaszlo server responds to the client with the details of a book
  6. The OpenLaszlo clients presents the book details to the user

Figure 7 | Use case - Amazon communication

The non-proxied communication between the client and server is depicted in figure 9. This figure describes the situation matching the use-case depicted in figure 10.

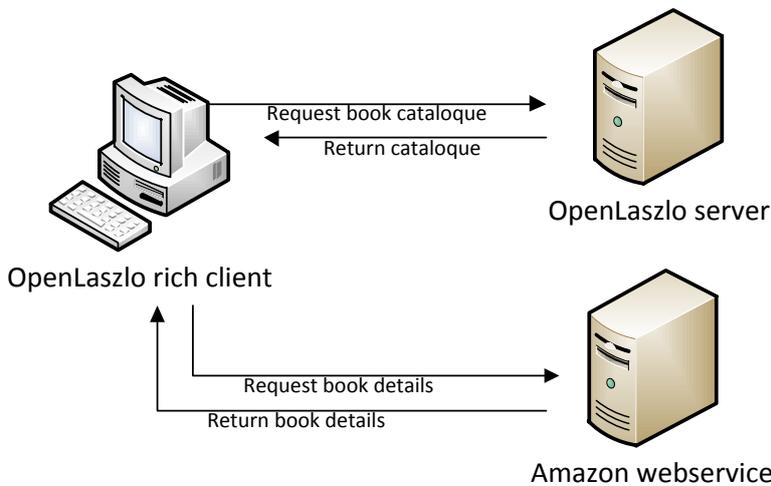


Figure 8 | Non-proxied communication

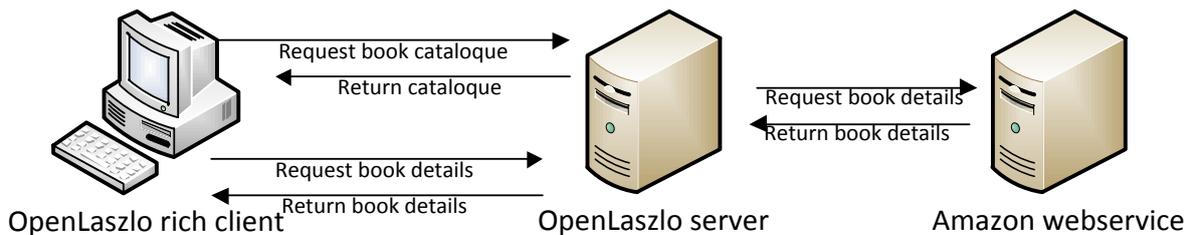


Figure 9 | Proxied communication

### Testing proxied and non-proxied communication

In order to test the difference between proxied and non-proxied communication the use-case depicted in **Error! Reference source not found.** was put into an OpenLaszlo application. After coding the application it's tested manually for intended use (blackbox testing).

After a successful completion of the application the use-case is performed and the HTTP requests are recorded. This resulted in a dirty dataset with the overall HTTP invocations for each mime-type the browser had to load, including the OpenLaszlo application itself. The remote invocations are isolated from the total recording and saved to the Apache JMeter test suite.

The following precautions are made in the comparison to ensure a comparable dataset:

- A comparison is made by using the different modes of application deployment platforms supported by OpenLaszlo. These platforms are SWF8, SWF10 and DHTML5.
- The browser influence is left out of the test because the sample used in the test is retrieved through the webservice invocations recorded on the HTTP layer so the actual performance of the browser running the client has no positive or negative influence on the data.
- The test is performed and recorded digitally by load test automation software.

The trade-off is evaluated for performance measured in average response time measured in milliseconds for both the proxy and non-proxy method of information retrieval. The average response time is calculated using the result from an output listener in the test suite. An excerpt of the output of Apache JMeter is depicted in table 2.

Sample #	Start Time	Thread Name	Label	Sample Time (ms)	Status	Bytes
40	19:31:00.006	Test Amazon DHTM..	/lps-4.9.0/demos/..	2009	!	2030
41	19:31:00.111	Test Amazon DHTM..	/lps-4.9.0/demos/..	2008	!	2030
42	19:31:00.106	Test Amazon DHTM..	/lps-4.9.0/demos/..	2009	!	2030
43	19:31:00.510	Test Amazon DHTM..	/lps-4.9.0/demos/..	2007	!	2030
44	19:31:00.513	Test Amazon DHTM..	/lps-4.9.0/demos/..	1972	!	2030

Table 2 | Excerpt of Apache JMeter output

### Results

The Apache JMeter test suite performed 1000 calls to the Amazon webservice simulating invocations from different platforms (SWF8, SWF10, DHTML5) using the proxied and non-proxied connection method. The average response time for the different connections and platforms are depicted in table 3

	SWF8	SWF10	DHTML5
<b>Proxied</b>	2915	2848	2862
<b>Non-proxied</b>	1605	1865	1587

Table 3 | Testresults of proxied versus non-proxied connection

As depicted in table 3 the non-proxied connection had faster response times on all platforms tested.

## Comparison with two alternatives

OpenLaszlo is not a system that just exists just on its own, it has predecessors in terms of past versions and in terms of its competitor's. Two similar systems will be chosen to compare it with OpenLaszlo 4.9.0, one system will be a predecessor and one system will be a competitor.

## Introducing OpenLaszlo versions and Competitors

OpenLaszlo was founded by David Temkin in 2000 and started as a closed project where only a selected number of partners had seen the preview versions. Within 2002 the premium version of OpenLaszlo was launched where Behr (Behr, 2011) was the first application developed within OpenLaszlo. As from 2004 OpenLaszlo became an open source project under the GPL open source license. This change in license allows finding detailed version changes (table 4).

Version 1	1.0.1	LZX source for redmond components refactored and made more readable.
		New Redmond horizontal scroll bar
		Server statistics request
	1.0.2	There are individual properties for specifying http back-end timeouts. See lps.properties.
The default timeout for back end http requests is now 5 seconds. (In previous releases the default timeout was infinite.)		
Version 2	Beta1	The Laszlo Component Set
		Laszlo Presentation Server 2.0 introduces the Laszlo (lz) component
	2.0	Krank Optimization
		Laszlo Presentation Server 2.0 introduces a new feature, called Krank, for optimizing the startup time of an application. Krank allows you, without changing the source code, to construct an application that doesn't need to run any initialization code when it starts. Kranking an application thus can often result in a dramatic improvement in user-perceived startup time.
		New components: tree and slider
		Faster initial compile of applications that use components.
	2.1.2	The compilation cache is now optimized to avoid recompiles when all that is needed is a re-encode (from gzip to uncompressed or vice-versa).
		New license key format. Existing 2.0 license keys must be upgraded to 2.1 keys in order for them to work with 2.1.
	2.2	Updated lzx.dtd verified for use with Eclipse
		LPS no longer supports 1.3 JREs/JDKs.
LPS requires container support for the 2.3 (or later) version of the servlet specification		
Version 3	3.0	There are now separate hscrollbar and vscrollbar components that can be used to minimize application size.
		SOLO deployment
		Smaller file sizes (using internal gzip compression)
		Dynamic libraries
		Unicode support
		Optimized for Flash Player versions 6 and 7
		Settable timeouts
		Drawing API
		Client fonts
		Resizable canvas
		Integration with browser JavaScript
		Debugging improvements
		Performance enhancements
	Separate Designer and Deployer guides	
The name of the product is now "OpenLaszlo"		
3.1	Charting and Graphing Components	
	Flash 8 file generation	
	XMLHttpRequest ("ajax") API	
	Rich Text Class	
	Local datasets	

		Version detection
		Developer console enhancements
		Global "hand cursor" parameter
		Backtrace facility in the debugger
		Other debugger improvements
	3.2	Since the 3.1 release, the Krank feature has been removed.
		New contributions in the incubator
		MSAA-compatible accessibility
		JavaScript compiler ported from Jython to Java
		Automated unit tests
	3.3	Drag improvements for basewindow
		Compiler support for streaming video
	3.4	CSS2 support (limited runtime style support)
	Streaming Audio and Video	
Version 4	4.0	Multi-runtime Architecture. The entire runtime architecture has been overhauled. The most important new developer-visible feature is the change in how runtime can be selected when an OpenLaszlo application is fetched from the server.
		New DHTML Runtime and new URL parameters.
		JavaScript language improvements.
		Wrapper pages and JavaScript include files.
		Resource Loading. OL4 includes a major rewrite of the resource loading system in the compiler and the server.
		SOLO in DHTML. This release contains support for SOLO deployment with the DHTML runtime.
		Cascading Style Sheets (CSS). An initial implementation with limited functionality and very little integration into the LFC or the Components is included in this release.
		Audio – Video. Support for streaming audio and video media in applications compiled for the Flash Player is available in OpenLaszlo 4.
		Debugging Improvements.
		New Test Systems.
	4.0.5	Explicit Replication. This proposal is to expose data-driven replication as an explicit syntax tag in LZX. Currently a databound node is implicitly replicated if its datapath matches more than one node in the XML tree; with this change, the decision whether to replicate or not is pushed up into the source code and made explicit.
	4.1	User classes are no longer defined as part of the global namespace. This gives OpenLaszlo better interoperability with other frameworks.
	4.2	New SWF9 Compiler Switch
	4.3	Audio/Video Improvements
		Improved Debugger to deal with deep and circular objects when printing
	4.7	HTML 5 Shadow Across Runtimes
		Binary Library Support
	Google Chrome Plug-in	
	SWF10 Support	
4.8	Build System Changes	
4.9	Add support for CSS 'classes' to <node>	
	implement bidirectional text view for swf10	
	RTE component. Alpha release for DHTML	
	automatically generate exploded widget directory for developers to test SOLO widget deploy	

Table 4 | Important OpenLaszlo changes (OpenLaszlo Archive, 2011) (Changelog 3.3.1, 2011)

Version 2 Beta 1 introduced some changes that were incompatible with previous versions. Beta 2 introduced changes that were incompatible with Beta 1. Version 3.0 also introduced changes that were incompatible with previous versions. Also version 4.2 is incompatible with previous versions. This change log does not show versions that were only introduced to fix bugs. Table 5 shows OpenLaszlo’s competitors.

Framework	Software License	Build formats
Adobe Flex	MPL	Adobe AIR, SWF
Cappuccino	LGPL	JavaScript, .sj
QuickConnectFamilyFramework	MIT	Iphone, Android, BlackBerry, Mac, Linux
Curl	Proprietary	Curl
Appcelerator Titanium	Apache 2	Iphone, Ipad, Android
Google Web Toolkit	Apache 2	Javascript
iPFaces mobile framework	Open Source	Iphone, Blackberry, Java ME
Lively Kernel	MIT	Javascript
MotherApp	Proprietary	Iphone, Android, BlackBerry, Windows Mobile, Symbian
PhoneGap	Open Source	Iphone, Android, BlackBerry
Qooxdoo	LGPL / EPL	Javascript
Rhobile	Open Source	Iphone, Android, BlackBerry, Windows Mobile, Symbian
.Net Framework	Proprietary	Microsoft Silverlight
Expanz	Proprietary	WPF, Microsoft Silverlight, Adobe Air, Flash, Java FX, Windows Mobile
CaptainCasa	Proprietary	WPF, Microsoft Silverlight, Ajax, Flash, Java FX
Smart GWT & Smartclient	LGPL & Commercial	Javascript client, Java server
Sproutcore	MIT	Javascript
Tersus	Open Source	Javascript, Iphone, Android, Symbian planned, BlackBerry planned
Vaadin	Apache 2	Javascript
ZK	LGPL / GPL / Commercial	Javascript
JavaFX	Open Source	Jar, applet
Qt_Quick	LGPL / GPL / Commercial	QML
Echo3	LGPL / GPL / MPL	Javascript client, Java server

Table 5 | OpenLaszlo competitors (List of Rich Internet Application Frameworks, 2011)

## Choosing an comparative version and competitor

As the trade-off is about the client-server communication within a proxied or non-proxied environment a predecessor has to be found that has a different architecture in means of client-server communication. Table 4 shows the OpenLaszlo changes over time, version 3.0 introduced the SOLO deployment feature and Version 4.0 introduced multi-runtime architecture. Other important architectural changes in OpenLaszlo since Version 3.0 are: Version 4.0 SOLO in DHTML, Version 4.2 New SWF9 compiler switch and Version 4.7 SWF10 support. By choosing Version 3.0 a lot of architectural changes that could influence client-server communication are investigated as the investigated system earlier in this research was version 4.9.

Table 5 provides an overview of current OpenLaszlo competitors. Some facts for Table 4: 78% are licensed, 43% are built with Javascript and 30% are rich internet application frameworks for smartphones. These facts are given to illustrate the differences with OpenLaszlo, as OpenLaszlo has an Open Source license and is built in SWF/ DHTML. Another build format that isn't widely used is the Microsoft Silverlight format, Expanz is one Framework that uses Silverlight and is exactly the competitor chosen to compare with OpenLaszlo.

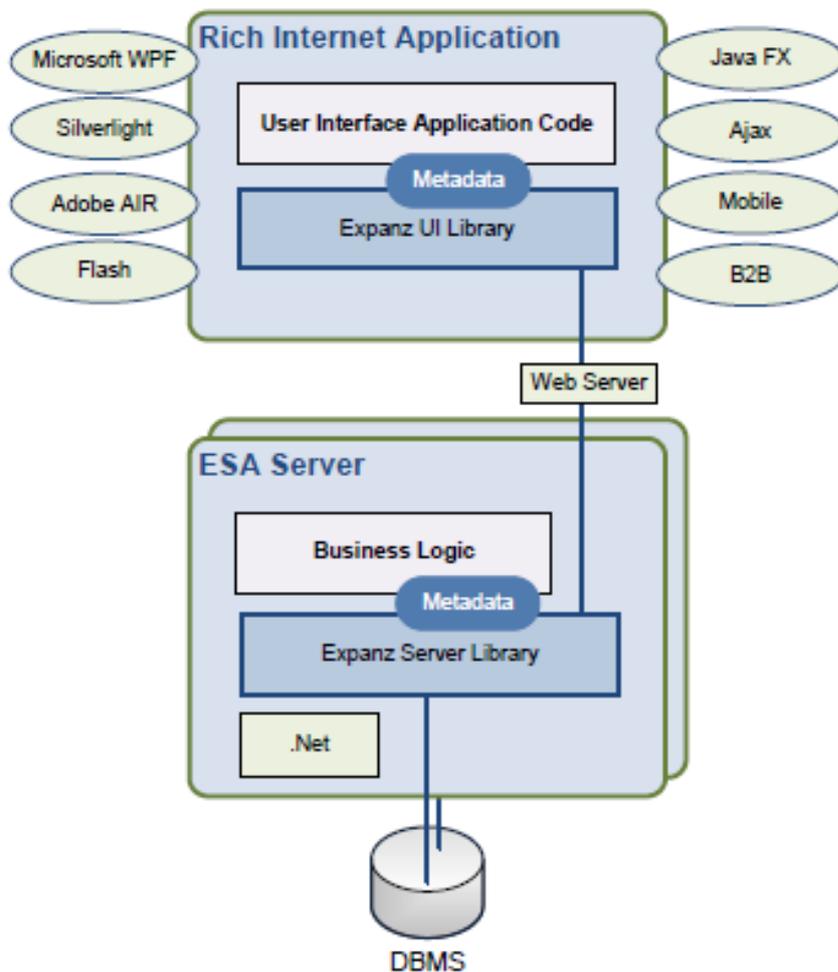


Figure 10 | Expanz Platform Server-Client

Expanz works with Microsoft WPF, Silverlight, Adobe AIR, Flash, Java FX, Ajax, Mobile and B2B on the client side and with .NET on the server side. Due to scope limitations only Silverlight (Client) and .NET (Server) will be examined for performance trade-offs on client-server communication. Expanz has the same kind of client-server deployment as OpenLaszlo (Figure 11). An EasyServerARCHITECTURE (ESA) (which is described by its creators as a service application platform) portal serves as a bridge between the client and server providing access through SOAP calls. This approach is the same as the proxied deployment of OpenLaszlo. Security on the client side is maintained by a sessionhandle which contain encoded server IP address and port. Difference between Expanz and OpenLaszlo is that Expanz can't be deployed SOLO. "ESA enforces total abstraction – It is impossible to declare a rule based on a presentation layer feature. Every client-side object is named/linked to a published server-side object, and it is the architectures job to maintain value equivalence between any server and client changes. All server activities (business process/use-case) publish a well-defined interface (schema) of fields and methods". (Expanz Platform Overview, 2011)

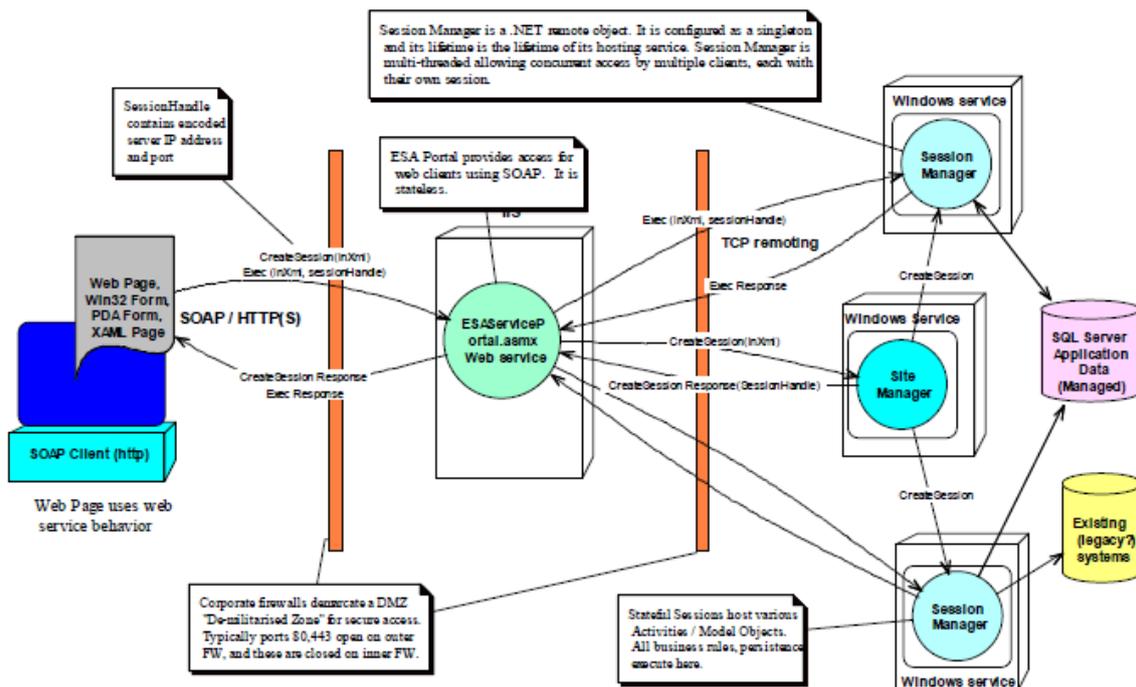


Figure 11 | Expanz deployment topology

### Compiling with .NET and Silverlight

As the .NET Framework works with the Code Access Security (CAS) model, Silverlight differs from .NET and has an attribute-based security model (MSDN The Silverlight Security Model, 2011) named the CoreCLR security model. Whereas CoreCLR is the acronym for Core Common Language Runtime. The CoreCLR is a stripped off version of the Common Language Runtime (CLR) which is the virtual machine from the Microsoft .NET initiative. CoreCLR differs from CLR in the fact that the JIT compiler is more focused on performance instead of complex optimizations. The garbage-collection mode tuned for multiple worker threads has been changed so now it only includes the standard workstation GC tuned for interactive applications (MSDN Program Silverlight with the CoreCLR, 2011). CoreCLR runs side-by-side in process with the CLR. Developers that use CLR write their code in a programming language such as C# or VB.NET. A .NET compiler translates the source code towards the Microsoft Intermediate Language (MSIL), this code is translated again by the just-in-time-compiler towards the code that can be executed on the system where the CLR is instanced on.

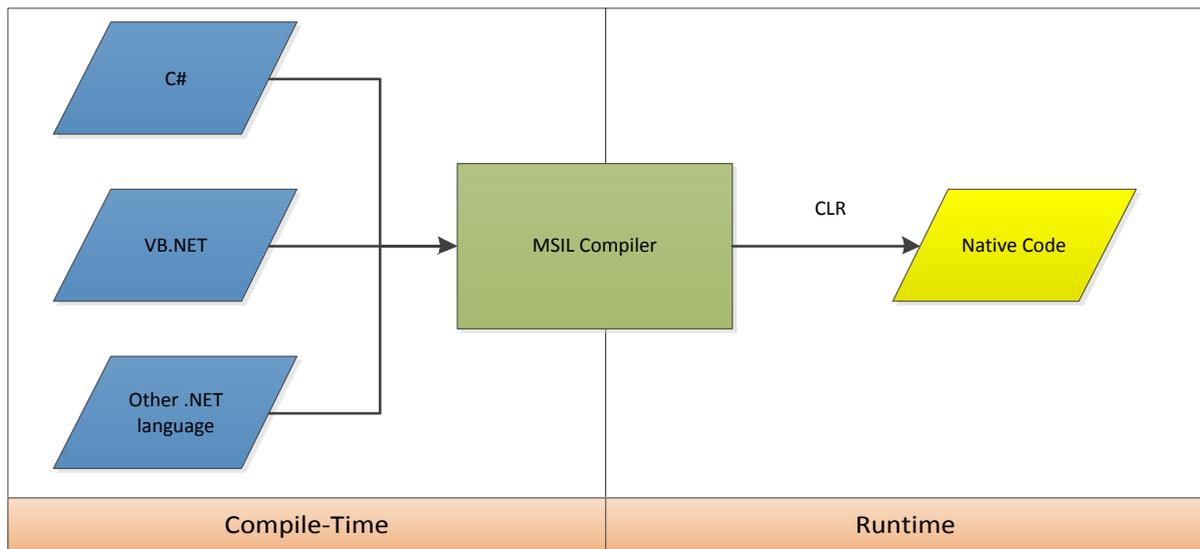


Figure 13 | Common Language Runtime (CLR)

### Laszlo Presentation Server Version 2.0

Laszlo Presentation Server (LPS) is a predecessor of OpenLaszlo, this predecessor was only available for a limited number of users while OpenLaszlo is open for anyone to use. The following table (table 6) gives a representation of all features from LPS to OpenLaszlo 4.9 that could improve performance in client-server communication.

Version 3	3.0	SOLO deployment
		Smaller file sizes (using internal gzip compression)
		Optimized for Flash Player versions 6 and 7
		Integration with browser JavaScript
		Performance enhancements
	3.1	Flash 8 file generation
	3.2	Since the 3.1 release, the Krank feature has been removed.
JavaScript compiler ported from Jython to Java		
	3.4	CSS2 support (limited runtime style support)
Version 4	4.0	Multi-runtime Architecture. The entire runtime architecture has been overhauled. The most important new developer-visible feature is the change in how runtime can be selected when an OpenLaszlo application is fetched from the server.
		New DHTML Runtime and new URL parameters.
		JavaScript language improvements.
		Wrapper pages and JavaScript include files.
		Resource Loading. OL4 includes a major rewrite of the resource loading system in the compiler and the server.
		SOLO in DHTML. This release contains support for SOLO deployment with the DHTML runtime.
		Cascading Style Sheets (CSS). An initial implementation with limited functionality and very little integration into the LFC or the Components is included in this release.
	4.2	New SWF9 Compiler Switch
4.7	HTML 5 Shadow Across Runtimes	
	Binary Library Support	
	SWF10 Support	

Table 6 | Performance changes since LPS 2.0 to OpenLaszlo 4.9

Performance differences between LPS 2.0 and OpenLaszlo 4.9 could differ when Flash player 8 is used in LPS 2.0 and when Flash player 10 is used in OpenLaszlo 4.9. But this measurement will fall out of scope as these performance differences are hard to measure. Another difference in performance could be found in the use of the Krank feature. “Laszlo Presentation Server 2.0 introduces a new feature, called Krank, for optimizing the startup time of an application. Krank allows you, without changing the source code, to construct an application that doesn't need to run any initialization code when it starts. Kranking an application thus can often result in a dramatic improvement in user-perceived startup time.” This feature was dropped in OpenLaszlo version 3.2. This suggests that the usage of Krank became obsolete as other feature implements promised the same start up times. One of those feature implements could be the use of smaller file sizes (using internal gzip compression). A lot of performance enhancements can be found during the development of OpenLaszlo within either Flash players, Javascript enhancements or the multi-running architecture.

### Expanz: Silverlight CoreCLR security model

Microsoft chose to move from the Code Access Security (CAS) towards CoreCLR security model due to complicated nature of CAS. CAS allows a user or administrator to define various sandboxes for code using permission sets and then map individual assemblies to those sandboxes (MSDN Program Silverlight with the CoreCLR, 2011). Rather than using various sandboxes CoreCLR only uses one sandbox that's equivalent to the sandbox that Internet Explorer uses for running script in a web page.

Another change in security is found in the simplification of the security enforcement model (Figure 13). CoreCLR is now based on security transparency. The security transparency model categorizes code into one of three buckets: Transparent, SafeCritical, or Critical code. Transparent is the lowest trust level for code and cannot elevate privilege or access sensitive resources or information on the computer. All application code in Silverlight 2 is transparent. Critical code is the highest trust level for code and can interact with the system through P/Invokes or even contain unverifiable code. Within Silverlight 2 all critical code must be part of the Silverlight platform. The bucket SafeCritical acts as a bridge between Transparent and Critical code. It allows Transparent code to access system resources by calling Critical code.

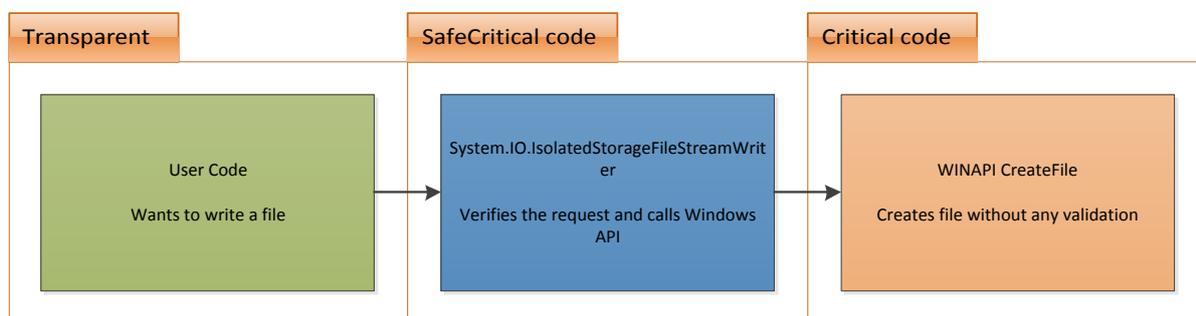


Figure 14 | Security Enforcement Model

**Performance against security trade-off between OpenLaszlo and Expanz**

A comparison between proxied mode and non-proxied mode can't be given for Microsoft Silverlight due to time limitations and technical reasons within this paper. But a clear distinction of performance in the security of both frameworks is present. There is one clear trade-off in Silverlight that decreases performances for compatibility due to the usage of the CLR method. The CLR method in Silverlight has the ability to run on any platform (depending on an available CLR compiler for that platform), but to support this performance is traded due to the compiling of executing code. Another reason for the lack of a comparison in proxied and non-proxied mode between OpenLaszlo and Expanz is the missing SOLO mode in Expanz.

**Learnability against compatibility trade-off between OpenLaszlo and Expanz**

Another trade-off comparison could be made in learnability against compatibility. As OpenLaszlo works with SWF and DHTML the learnability is low as the usage of these build formats looks a lot like HTML and XML programming. Expanz uses the build format .NET which supports better compatibility as it could run on any platform. But as the Expanz framework has many possibilities the learning curve is around 1 to 2 weeks for developers and 2 to 3 months to become expert.

## Conclusion and discussion

This paper addresses some concerns found in the architecture of the OpenLaszlo framework. Architecture deals with design decisions and making compromises between different software quality aspects. These decisions result in a tradeoff between two or more quality aspects in a software system. In many cases a best of both world decisions is made when difficult choices arise. The responsibility for making these choices is moved from the OpenLaszlo framework architect to the OpenLaszlo application architect.

This paper addresses a tradeoff found in different types of deployment of OpenLaszlo client applications that use information from external sources. Information from external sources can be accessed by using different types of protocols like XML-RPC, JavaRPC, and SOAP. OpenLaszlo supports two different strategies for dealing with external connections. Connections can be made to an external datasource directly (e.g. the Amazon webservice), this is called the non-proxied connection. It is also possible to use the OpenLaszlo server as a proxy. In this case all webservice calls are made to the OpenLaszlo server who deals with the requests and return the right information. The choice between deployment methods results in a tradeoff between functionality of security and efficiency of resource utilization according to the ISO/IEC 9126 quality model (Jung, Kim, 2004).

The tradeoff is exposed in an experiment using a modified version of an OpenLaszlo reference application for retrieving book detail through the Amazon webservice. In a test environment six different set-ups of the same applications were tested with the test automation tool Apache JMeter with a 1000 requests for every set-up. The six platforms are a combination of the two connection method: proxied, and non-proxied and three different deployment platforms: SWF8, SWF10, and DHTML. The result of the test showed that the non-proxied set-up was performing significantly faster in comparison to the proxied connection. The difference in performance can be addressed to the overhead of using centralized communication through the OpenLaszlo server.

### Discussion

This research provides an overview of the architecture of OpenLaszlo and exposes the details of a single tradeoff in the architecture of the OpenLaszlo server. The conditions that expose this tradeoff are based on an OpenLaszlo showcase application and therefore reflect a real problem that software architects have to deal with. Still the tests performed are scoped to enable a comparable situation. An important concession made in this research is the influence of the performance of the client platform because of its unpredictable nature.

It is important to acknowledge that the tradeoff exposed in this paper is probably not the only one. Further research has to be performed to expose more of these tradeoffs between quality aspects to ensure better understanding about the OpenLaszlo framework and to contribute to the body of knowledge of the software architecture domain.

## References

- Behr*. (2011). Retrieved Januari 12, 2011, from Behr: <http://b7101.behr.com/Behr/home#>
- Changelog 3.3.1*. (2011). Retrieved Januari 12, 2011, from OpenLaszlo: <http://download.openlaszlo.org/3.3.1/changelog.html>
- Expanz Platform Overview*. (2011). Retrieved Januari 12, 2011, from Expanz: [http://www.expanz.com/LinkClick.aspx?fileticket=8\\_w-GVASg-o%3d&tabid=258](http://www.expanz.com/LinkClick.aspx?fileticket=8_w-GVASg-o%3d&tabid=258)
- List of Rich Internet Application Frameworks*. (2011). Retrieved Januari 12, 2011, from Wikipedia: [http://en.wikipedia.org/wiki/List\\_of\\_rich\\_internet\\_application\\_frameworks](http://en.wikipedia.org/wiki/List_of_rich_internet_application_frameworks)
- MSDN Program Silverlight with the CoreCLR*. (2011). Retrieved Januari 12, 2011, from MSDN: <http://msdn.microsoft.com/en-us/magazine/cc721609.aspx>
- MSDN The Silverlight Security Model*. (2011). Retrieved Januari 12, 2011, from MSDN: <http://blogs.msdn.com/b/shawnfa/archive/2007/05/09/the-silverlight-security-model.aspx>
- OpenLaszlo Architecture*. (2011). Retrieved January 12, 2011, from OpenLaszlo: <http://www.openlaszlo.org/lps4.9/docs/developers/architecture.html>
- OpenLaszlo Archive*. (2011). Retrieved Januari 12, 2011, from OpenLaszlo: <http://www.openlaszlo.org/archive>
- Bevan, N. (1999). Quality in use: meeting user needs for quality. *Journal of Systems and Software*, 49(1), 89--96.
- Clemetns, P. (1995). Understanding architectural influences and decisions in large-system projects. *Proceedings of the first international workshop on architectures for software systems*. Seattle: citeseer.
- Jung, H. W., Kim, S. G., & Chung, C. S. (2005). Measuring software product quality: a survey of ISO/IEC 9126. *IEEE Software*, 21(5), 88--92.
- Marco, C. Q. (2004). ISO/IEC 9126 in Practice: What Do We Need to Know? *In Proceedings of the 1st Software Measurement European Forum (SMEF)*.

## Appendix A

### Answers to questions after the literature assignment

1. It seems that the LZX language mixes representation code with the code associated to events. It seems that it goes against the separation of languages that is for example found when separating HTML, CSS and JavaScript in different files. How does this affect the maintainability of the code versus efficiency and easiness of implementation of the compilation phase? (Alejandro Serrano Mena)  
*OpenLaszlo incorporates the logic of Javascript into the LZX presentation code. This is a drawback for the maintainability of the code, on the other hand the Javascript does not need compilation but is interpreted by the browser or the flashplayer. So the maintainability of the code decreases, in return does not need compilation so the efficiency increases.*
2. Isn't there a very big trade-off between the compatibility among the various platforms that OpenLaszlo can run on, and the functionality and performance it provides? You would expect that, as the architectures in which the system runs are quite different (technologically) and they have very different implementations. How does OpenLaszlo accommodate this trade-off? (Geert Wirken)  
*OpenLaszlo uses the smallest of common features available on their deployment platforms. So in theory each supported platform limits the capabilities of OpenLaszlo applications. Therefore the available functionality decreases by adding a new deployment platform.*
3. In section 3.3 they describe that OpenLaszlo generates DHTML and allows the developer using JavaScript with the mentioned LZX tags. But then the paper says, that the actual executed JavaScript code in the browser is an optimized "mixture" of the developer's code and the interpretation of the tags. What is that optimization and why does OpenLaszlo do that? (Matthias Lossek)  
*OpenLaszlo supports a range of settings or deployment options for their applications. An example of this is the proxied versus non-proxied communication method discussed in this paper. OpenLaszlo makes small changes in the Javascript on the compilation level to incorporate these design decisions like changing the request URL for a SOAP request.*
4. OpenLaszlo certainly seems a promising system. However, it seems that this piece of software is very enterprising, that is to say: it appears to be very bloated and complicated. How does this affect performance? It seems to be a Swiss army knife, a jack of all trades.  
  
How can you prevent that all applications generated by OpenLaszlo look and feel the same? (Alessandro Vermeulen)  
*Performance wise OpenLaszlo server is a jack of all trades indeed. But, the clients compiled by OpenLaszlo only contain the functionality and libraries invoked in the application. Just like regular websites, OpenLaszlo supports custom designs.*
5. When I compile my OpenLaszlo application, I have to specify the target supported runtime in which it will execute in. Is it possible to write an OpenLaszlo application which will run in an environment which is not supported yet? (Jacek Marek)  
*No, it would be supported then. Of course you can be creative with the applications that are supported by building interpreters or mediators for them.*

6. OpenLaszlo includes a size and speed profiling. How does this actually work? (Vladimir Smatanik)

*The OpenLaszlo server offers a range of tools to optimize and debug OpenLaszlo source code including:*

- *Debugger client which connects to the J2EE compilation component of OpenLaszlo server. The debugger offers statistics of compilation time, memory footprint, size of the application ,etc. More information about this is available in the debugging section of the developers manual.*

7. What kind of project is OpenLaszlo best suited for? A small personal project or a corporate-wide project? For the latter case, what is an estimate overhead required for the company to train its developers into efficiently using OpenLaszlo? (Richard Derer)

*Running the OpenLaszlo server for development or production purposes requires knowledge and experience with J2EE compatible servers. This can be seen as a highly specialized skill. Creating and running an OpenLaszlo application requires substantially less specific knowledge. In this situation OpenLaszlo is suited for small corporate projects.*

8. Can you create your own custom components in LZX to use in your OpenLaszlo application? (Alexandru Dimitriu)

*OpenLaszlo applications always consist of one or more components. Components can be used in a hierarchical manor. OpenLaszlo development is done by extending the basic set of components available into new custom components.*

9. You have the choice to compile your application in several versions of Flash or DHTML. Is it possible to support all versions of flash, for example checking the flash version of the client and then sending the client the right swf? Do you really have to choose between flash and dhtml? I mean is there a big difference in code which makes the choice necessary? (Renato Hijlaard)

*The user can request a specific version of the client (SWF8, SWF10, DHTML) of the OpenLaszlo clients through an URL parameter. If no parameter is supplied the OpenLaszlo server serves the default platform declared in the application. If no default deployment platform is declared in the application the system default (usually SWF10) is used.*

10. Can you give examples of situations where a proxied implementation is preferred above the SOLO implementation and vice versa? (Hans Peersman)

*The comparison of proxied implementation with SOLO implementation in this question is not correct. Proxy and SOLO implementations are not real counterparts of each other. A SOLO application just contain all logic necessary and does not need an OpenLaszlo server to run. A SOLO application can be put on CD-rom or e-mailed to a friend. A proxied version implicated that traffic from a client to an external server run through the OpenLaszlo server. Proxied communication can be used when a company want to exert control over client-server communication.*

11. OpenLaszlo seems very useful for making online applications. What I'm curious about is if it is used mainly by companies (i.e. company websites) or more for hobby sites? Is there a reason for this distinction of usage? And what can you tell about the usage growth/decline of the past years? (Tim de Boer)

*OpenLaszlo has a learning curve which makes it unsuitable for hobby-sites. Companies that use OpenLaszlo are: IBM, FNAC, Wal-Mart, Pandora, and Gliffy.*

12. Compared to other rich internet applications frameworks (Adobe Flex, .Net), what do you think about the future of Laszlo? Has Laszlo reached the end of life in terms of development? Or does it bode well for the future? Assume a start-up company wants to choose/invest in this technology, would you recommend it? (Sander van der Rijst)

*OpenLaszlo is an opensource application and therefore it is difficult to say something about the future of it. The Internet is always changing, what means that OpenLaszlo needs to change with it. The future of the Internet is very positive so the future of OpenLaszlo can be positive too. The vision of the OpenLaszlo foundation indicates that they want to expand the amount of supported platforms in the years to come. Examples for expansion are mobile devices and home media products.*

13. Everyone who has programmed web applications must have felt the frustrations of the several browsers interpreting your code different. The power of OpenLaszlo is to write your code once and run it everywhere. If you choose to do a SOLO deployment instead of using a proxy approach; how is this principle preserved? Do you have to compile your application several times and write your own code to choose what compilation to serve to the client? If so, don't you lose the only strongpoint of OpenLaszlo? (Lambert Veerman)

*A SOLO deployment in DHTML is platform independent because it contains logic to deal with differences between browsers. A SOLO deployment in Flash is less platform independent, it needs a flashplayer which is not supported on all platforms (cq. operating systems). It is not the only strongpoint of OpenLaszlo (see section Quality Aspects) but it does have negative impact.*

14. Should OpenLaszlo really support Flash in the future? What are the expectations regarding new HTML standards, can they be integrated? (Matthijs Neppelenbroek)

*The features offered by the new HTML5 standard offer similar possibilities in comparison with Flash. There is no or little reason to stop supporting Flash in the future. One of the primary factors for the future developments of OpenLaszlo is the level of support for both Flash and HTML.*

15. The Laszlo systems allows developers to write Rich GUI applications without the burden of client specific anomalies (Things like incompatibilities between browsers etc.). In the development life cycle the developers writes the program code in LZX, and then compiles the source code against a selected target runtime. In essence the compiler handles all the incompatibilities between the runtimes. This is where my question arises.

Suppose we want to develop a rich gui application consisting of specific gui elements (e.g. well designed in Photoshop elements). In former web application development methods we had a fine grained control how the gui looked like. Unlike with Laszlo (correct me if I'm wrong), where you are programming against a library. How does the abstract way of programming with Laszlo interferes with the ability of fine grained gui/layout specification? And, does Laszlo have ways to overcome this problem? (Kevin van Blokland)

*OpenLaszlo has similar capabilities for constructing a custom lay-out. The default OpenLaszlo lay-out can be extended and customized to the wishes of a client.*

16. OpenLaszlo is intended for writing Web Applications. Are there any advantages when using OpenLaszlo for fully static websites? How complicated should your website be for OpenLaszlo to be advantageous? (Thijs Alkemade)

*There is no real split-off decision point where OpenLaszlo is better suitable than plain old HTML or Flash. OpenLaszlo deals with the differences between browsers which resolves a lot of classic problems which also arise building fully static websites. The applicability of OpenLaszlo depends for a large part on non-functional requirements like platform independence.*

17. Most of the web toolkits that provide ajax or simple html/javascript and generally rich web components have many browser compatibility issues. They all say that they follow the standards, and that they are always up to date, but in reality there are problems. How OpenLaszlo deals with this problem in case of serving applications in DHTML? (Theodoros Polychniatis)

*OpenLaszlo compiles application along with multiple Javascript libraries which deal with differences between browsers for a large part just like the other web toolkits do. The big difference between OpenLaszlo and web toolkits is that OpenLaszlo uses an intermediary format for declaring the UI with LZX code. LZX code is converted into standardized HTML.*

18. OpenLaszlo can be run in two modes: *Proxied mode* and *SOLO*. Is there any difference in performance between these modes? And if yes, what kind of applications (give some examples) can be best benefitted in performance by each of these modes? (Nikos Mytilinos)  
*A SOLO application is not the counterpart of a Proxied mode application. SOLO applications can be proxied as well. Performance details of proxied versus non-proxied are discussed in the sections above. Proxied applications offer more possibilities for applying security restrictions.*

19. To be able to "compile" to different platforms an OpenLaszlo application developer can only use features, which are offered by all platforms. Is it still a good idea or even possible to use OpenLaszlo if a needed feature (like e.g. DRM) is only offered by one platform? (Leupolz, J.S.)  
*This is a good point. In this case you can still use OpenLaszlo but you will lose some of it featured advantages like platform independence.*
20. Does OpenLaszlo leave any room to create custom user interface components, or compose them from existing ones? (Stijn van Drongelen)  
*Yes. OpenLaszlo is component based and every application has at least one component. Components can be extended and are implemented in a hierarchical manor.*
21. There are two ways in which you can use OpenLaszlo: proxied or SOLO. Do you know which is the most widespread use of OpenLaszlo? Do you think that choice is because it is better-performing? or because simplicity? If not, why is the reason? (Maria Hernandez)  
*SOLO and proxied are the least used types of deployment. The most widely spread way of using OpenLaszlo is non-proxied communication.*
22. OpenLaszlo is used to develop rich internet applications, which offers the same experience as native desktop applications. Therefore the OpenLaszlo architecture should have some differences in their architecture compared to traditional web applications. This brings me to my question; "which architectural differences are needed to offer these kind of experiences compared to traditional web applications". (Ruben van Vliet)  
*OpenLaszlo applications offer asynchronous interaction for a seamless experience just like native desktop applications. Traditional web applications offer synchronous interaction which requires refreshing pages.*
23. When reading this there are not much advantages compared to a SOLO application. Do you think this ability to build a proxied application negatively influences the aspects of understandability and maintainability? And what would have been the trade-off when they would have removed this 'proxied application building feature' out of OpenLaszlo? (Rik Janssen)  
*Proxied application are more complex to maintain and require experts for server maintenance. Still, some features offered by proxied connection are not existent in other deployment setups like: the ability to monitor outgoing traffic, debug options, etc..*
24. Is it possible to use the Data Manager to also extract the data into a format suitable for other programs instead of just functioning as an interface? (Jeroen van der Velden)  
*The data manager compiles the data to a compressed binary format. The data manager can only be access using a SOAP request.*