Marquette University
e-Publications@Marquette

Master's Theses (2009 -) Dissertations, Theses, and Professional Projects

3D Visualization Architecture for Building Applications Leveraging an Existing Validated Toolkit

David A. Polyak
Marquette University

Recommended Citation
Polyak, David A., "3D Visualization Architecture for Building Applications Leveraging an Existing Validated Toolkit" (2014). *Master's Theses (2009 -)*. Paper 255.
http://epublications.marquette.edu/theses_open/255

3D VISUALIZATION ARCHITECTURE FOR BUILDING UBIQUITOUS APPLICATIONS LEVERAGING AN EXISTING VALIDATED TOOLKIT

by

David A. Polyak

A Thesis submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the requirements for
the Degree of Master of Computing

Milwaukee, Wisconsin

May 2014

ABSTRACT

3D VISUALIZATION ARCHITECTURE FOR BUILDING UBIQUITOUS APPLICATIONS LEVERAGING AN EXISTING VALIDATED TOOLKIT

David A. Polyak

Marquette University, 2014

The diagnostic radiology space and healthcare in general is a slow adopter of new software technologies and patterns. Despite the widespread embrace of mobile technology in recent years, altering the manner in which societies in developed countries live and communicate, diagnostic radiology has not unanimously adopted mobile technology for remote diagnostic review. Desktop applications in the diagnostic radiology space commonly leverage a validated toolkit. Such toolkits not only simplify desktop application development but minimize the scope of application validation. For these reasons, such a toolkit is an important piece of a company's software portfolio. This thesis investigated an approach for leveraging a Java validated toolkit for the purpose of creating numerous ubiquitous applications for 3D diagnostic radiology. Just as in the desktop application space, leveraging such a toolkit minimizes the scope of ubiquitous application validation. Today, the most standard execution environment in an electronic device is an Internet browser; therefore, a ubiquitous application is web application.

This thesis examines an approach where ubiquitous applications can be built using a viewport construct provided by a client-side ubiquitous toolkit that hides the client-server communication between the ubiquitous toolkit and the validated visualization toolkit. Supporting this communication is a Java RESTful web service wrapper around the validated visualization toolkit that essentially "webifies" the validated toolkit. Overall, this ubiquitous viewport is easily included in a ubiquitous application and supports remote visualization and manipulation of volumes on the widest range of electronic devices.

Overall, this thesis provided a flexible and scalable approach to developing ubiquitous applications that leverage an existing validated toolkit that utilizes industry standard technologies, patterns, and best practices. This approach is significant because it supports easy ubiquitous application development and minimizes the scope of application validation, and allows medical professionals easy anytime and anywhere access to diagnostic images.

ACKNOWLEDGMENTS

David A. Polyak

This work is the result of my MS studies at Marquette University and my work at General Electric Healthcare as a Software Engineer. During this time I have met many inspirational people that have helped to guide me, and I would like to take the time to acknowledge their contributions and thank them.

During my studies I have taken many software engineering courses from Dr. Sheikh Iqbal Ahamed and he has been influential to my studies since my senior year at Marquette University. With the help of Dr. Sheikh Iqbal Ahamed as a senior a Biocomputer Engineering student I decided to continue my education as a MS student at Marquette University in fall of 2010. Very important to me was his support in starting my professional career at GE Healthcare in spring of 2012. Despite losing me as a fulltime student and Teaching/Research Assistant, I have received his utmost support as my studies transitioned to part time as I developed my industry software development skills.

My switch from a fulltime student to full time employee was an enjoyable transformation and my many mentors at GE were instrumental in my success at GE and finishing my thesis. First and foremost, I'd like to thank my manager Gopal Avinash Ph.D. who has helped me grow as a software developer, help me define my thesis project, and has helped prioritize my work. I would also like to specially thank my coworkers: John Hoford, Fausto Espinal Ph.D., Evan Francis, Andrew Wong, Gabriel Fernandez, and Musodiq Bello Ph.D. All my coworkers, especially those in my Scrum team, have been influential to my success at GE and the success of this project. They have all in many ways helped me achieve my goals.

Last, but certainly not least, I would like to thank my mother and father for all their love and support throughout my life and especially during my studies at Marquette University. Without their love, support, and motivation this dream would have never come to fruition.

TABLE OF CONTENTS

ACKNOWLEDGMENTS i

LIST OF TABLES v

LIST OF FIGURES vi

Chapter 1: INTRODUCTION 1

Chapter 2: MOTIVATION 7

Chapter 3: RELATED WORKS 11

3.1 Existing Java Validated Visualization Toolkit 11

3.2 Web Services Layer 13

3.2.1 Client-Server Communication Protocols 14

3.2.2 Java Web Services 16

3.3 Ubiquitous Toolkit 19

3.4 Ubiquitous Applications 21

Chapter 4: SYSTEM CHARACTERISTICS 29

4.1 Existing Java Validated Visualization Toolkit 30

4.1.1 Dynamic Viewport Resizing 30

4.1.2 Rendering Modes 31

4.1.3 Preset Camera Views 31

4.1.4 Coordinate Transforms and Volume Geometry for Graphics 31

4.1.5 Camera Manipulation 33

4.1.6 Mouse Based Application Viewport Interaction 33

4.1.7 Render the Current Viewport Render Engine 34

4.1.8 Save and Restore Viewport State 34

4.2 Web Services Layer 34

4.3 Ubiquitous Toolkit 35

4.3.1 Viewport Interaction Performance (Image Return Size and Mime Type) .. 36

4.4 Ubiquitous Application 37

Chapter 5: APPROACH 38

5.1 Existing Java Validated Toolkit 39

5.2 Web Service Layer 40

5.2.1 "Stateful" Versus Stateless RESTful Web Services 43

5.2.2 Java Servlets Versus JAX-RS RESTful Web Services 48

5.2.3 JAX-RS RESTful Web Service Layer Architecture 50

5.3 Ubiquitous Toolkit 64

5.3.1 Mouse Interaction Architecture 72

5.3.2 Toolkit Error Handling Architecture 73

5.4 Ubiquitous Application 76

Chapter 6: EVALUATION 80

6.1 Proof of Characteristics 80

6.2 Performance 86

Chapter 7: CONCLUSION 91

7.1 Summary 91

7.2 Broader Impact....93

7.3 Future Work....94

BIBLIOGRAPHY....97

APPENDIX A....100

LIST OF TABLES

Table 3.1: Table of HTTP method to Java methods in the Java HttpServlet class. 17

Table 3.2: Table that summarizes the core JAX-RS annotations provided in the Java package javax.ws.rs annotations. 18

Table 3.3: Zero-Footprint Application Architecture Types and categorization based on business logic location. 23

Table 3.4: Benefits/Drawbacks of thin-Client versus Thick-Client architectures for web applications 25

LIST OF FIGURES

Figure 1.1: This figure depicts two architectures for desktop applications. 3

Figure 1.2: This figure depicts two architectures for web applications. 4

Figure 1.3: This figure depicts the architectural approach of this thesis. 5

Figure 3.1: Like a widget, a toolkit provides the building blocks for application development. This figure shows two viewports included in an application 13

Figure 4.1: Dynamic viewport resizing feature allows an application viewport to be resized to any positive, non-zero width and height 30

Figure 4.2: The platform supports from left to right: MPR, MIP, and Volume Rendering render styles. 31

Figure 4.3: The 3D reference cursor Visualization Component shares the same world coordinate across the four application viewports. 32

Figure 4.4: The color of the 3D reference cursor Visualization Component changes when the cursor's world coordinate does not exist within the confines of the volume (defined by the white bounding box). 33

Figure 4.5: The viewport on the left contains an image of equal width and height. The viewport on the right contains an image one forth the width and height. 37

Figure 5.1: This figure depicts the high-level architectural approach of this thesis. 38

Figure 5.2: This figure highlights the existing validated toolkit used in this architecture. 39

Figure 5.3: This figure highlights the web service layer architecturally wrapping an existing validated Java 3D visualization toolkit (viewed as a black box). 41

Figure 5.4: The image on the left shows an anterior view volume rendering. The image on the right shows a right view volume rendering. 43

Figure 5.5: This figure visualizes the three components of a 3D medical render engine's camera: Eye Point, Look Point and Up Vector. 45

Figure 5.6: This figure visualizes a render engine camera translation by 7 millimeters in the superior direction by updating the camera Eye Point and Look Point. 45

Figure 5.7: This figure describes the hierarchy of the web service layer REST interface through an interface class diagram. 51

Figure 5.8: This figure highlights the server-side web service layer architecture. 54

Figure 5.9: A web resource manager manages a collection of web resources. The web resource manager logically exists between the web services and a web resource. 58

Figure 5.10: This figure highlights the web service layer wrapping of the existing validated visualization toolkit by JAX-RS RESTful web services for the purposes of "webifying" the existing validated toolkit. 63

Figure 5.11: Screenshot of the Google Chrome developer console invoking the getViewHeight web service through the ubiquitous viewport's getViewHeight function. 70

Figure 5.12: This figure shows the available communication utility function provided by the WebUtils singleton JavaScript library. 71

Figure 5.13: This figure highlights the ubiquitous JavaScript toolkit that exposes a "plain vanilla" JavaScript ubiquitous viewport object and provides a web utilities library for client-server communication. 75

Figure 5.14: This figure shows the association between a HTML 5 canvas element and the ubiquitous toolkit yields a ubiquitous viewport that can be placed in a ubiquitous application. 78

Figure 5.15: This figure highlights the ubiquitous applications that leverage the ubiquitous toolkit for rapid application development. 79

Figure 6.1: A simple, single application viewport thin-Client and Zero-Footprint 3D visualization web application. 82

Figure 6.2: A four-port thin-Client and Zero-Footprint 3D visualization web application. 86

Figure 6.3: Chart of mouse interaction performance versus return image size and compression type for a 512 by 512 pixel viewport with a render style of multiplanar reformat. 88

Figure 6.4: Chart of mouse interaction performance versus return image size and compression type for a 512 by 512 pixel viewport with a render style of volume rendering. 88

Figure 6.5: Shows performance degradation as a function of render style, with varying image render sizes and image compression algorithms. 89

Figure 6.6: Shows performance degradation as a function of image compression algorithm, with varying render styles, and image render size. 90

Figure 7.1: This figure shows the generic architecture presented in this thesis. 94

CHAPTER 1: INTRODUCTION

Healthcare professionals are inherently mobile individuals. Clinicians and nurses frequently move from one room to another, and each room change generally signifies a change in context. Therefore, access to pertinent information needs to be effortless, and presented in a user-friendly form.

Significant advancement of mobile technology in recent years has altered the manner in which societies in developed countries live. The computational power and contextual awareness of mobile devices, in addition to the wide availability of wireless connectivity, has untethered people from the traditional static desktop. Mobile devices enable individuals to readily access and share information from virtually any location ubiquitously. In the context of this thesis, something that is ubiquitous is something that is accessible to the largest number of electronic devices. These advancements in technology have the capability to transform healthcare through ubiquitous applications and paradigms such as the Internet of Everything (Cisco) and Industrial Internet (GE). However, the medical world is a slow adopter of new technologies and best practices due to technical complexities and regulation.

Medical imaging is a huge component of healthcare. This domain continues to be transformed with advancements in software. Software allows these devices to safely acquire medical images and procure information previously unattainable. As in the past, advancements continue to push the envelope of what is possible. Today, imaging scanners and workstations are capable of advanced 3D image visualization, a process that visualizes a 3D volume out of a stack of acquired 2D images. Development of such applications is tedious, and technically difficult, and according to regulatory entities, each application needs

to be validated for diagnostic use (Food and Drug Administration, 2002). The inherent complexity and need for application validation limit the progress of technology in the visualization space.

Traditionally, desktop applications are built using a top down approach where algorithms are developed specifically to serve the application. This top down approach to application development leads to tightly coupled architectures. Instead of each application implementing the same core capabilities of 3D medical image visualization, these applications should leverage a central codebase (PC Magazine01), or a visualization toolkit (PC Magazine02). This bottom up approach focuses on the development of fundamental services needed for a wide range of applications. This architectural approach not only supports the rapid development of unique applications but results in a decoupled architecture.

In the medical imaging domain, the key to application development is a time-tested validated visualization toolkit. Time-tested validated visualization toolkits are invaluable in the medical software industry because they have already gone through the growing pains inherent in all software development, and they simplify application validation efforts. Specifically, these toolkits simplify application development, and minimize time to market through streamlined application development and regulatory validation. Although this approach is capable of supporting an infinite number of applications for diagnostic radiology, these are static desktop applications. These existing validated toolkits are not in themselves sufficient to support a multitude of ubiquitous applications. Figure 1.1 visualizes the two architectural approaches to static desktop applications, tightly coupled versus decoupled.

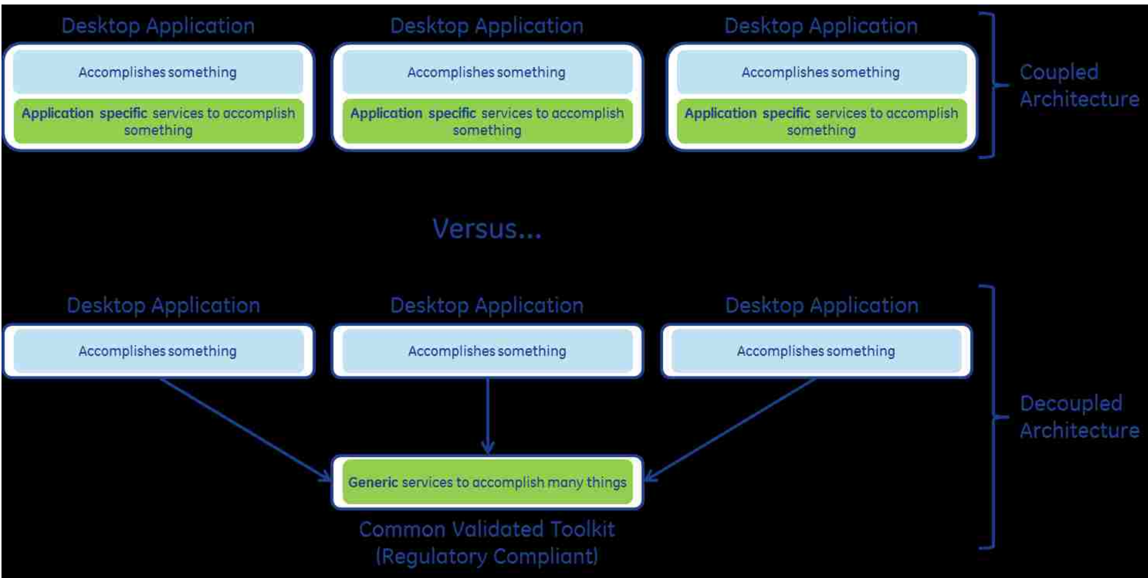

Figure 1.1: This figure depicts two architectures for desktop applications. A tightly coupled architecture includes application code and supporting services. A decoupled architecture includes application code that leverages a validated toolkit that provides generic services to a wide range of applications. Overall, applications leveraging an existing validated toolkit are much easier to develop and validate.

Today's connected world leverages the Internet and its technology stack to develop applications that allow users to view the same application on a multitude of devices. Such an application is commonly referred to as web application. Web applications commonly follow the client-server architecture i.e. lightweight client-side markup, scripts, styles, and supporting web services. This pattern coupled with core web technologies is perfect for developing applications capable of running on virtually any electronic device or platform. Unfortunately, with the web application technology stack, the development of web applications for diagnostic radiology is a difficult proposition. Just as it is prudent to leverage a toolkit for desktop diagnostic radiology applications, web applications for diagnostic radiology should leverage a robust supporting validated visualization toolkit. Without a tried and true supporting codebase, the web application technology stack is not sufficient for rapid validated diagnostic radiology application development. Figure 1.2 visualizes the two architectural approaches to web applications, tightly coupled versus decoupled.

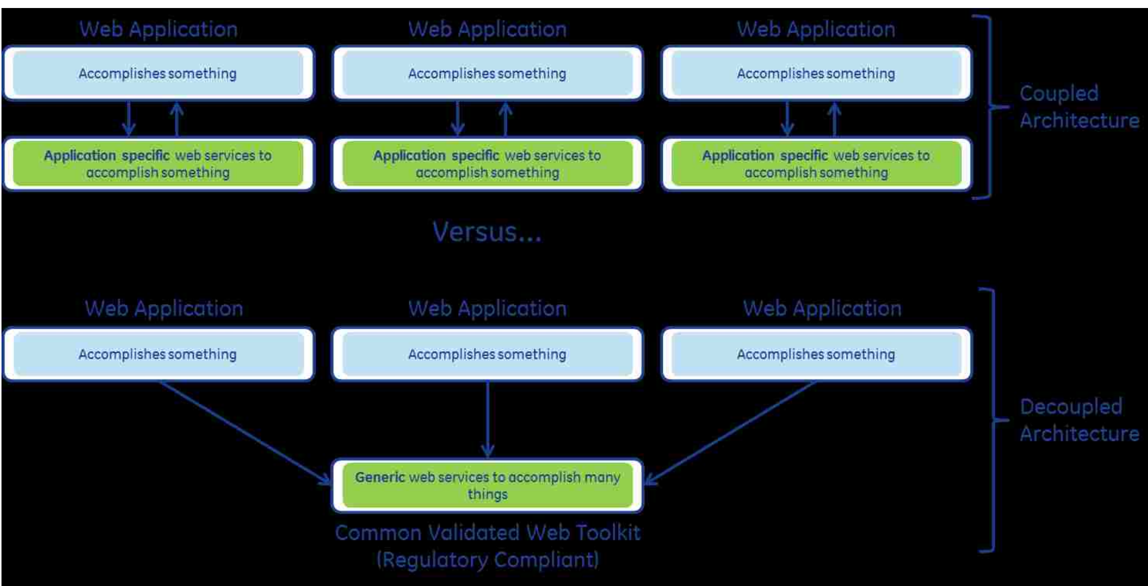

Figure 1.2: This figure depicts two architectures for web applications. A tightly coupled architecture includes client-side application code and supporting server-side web services. A decoupled architecture includes client-side application code that leverages a validated toolkit that provides generic server-side web services to a wide range of web applications. Overall, web applications leveraging an existing validated toolkit are much easier to develop and validate.

By reusing existing validated visualization toolkits and providing a client-server layer that interfaces with such toolkits (W3C, 2004), it is possible to enable the development of portable, web-based medical visualization applications for a wide range of computing platforms. Although web applications inherently run on a wide range of electronic devices, web applications may leverage Internet browser plugins that are not supported on all electronic devices. To reach the widest audience, a web application must be Zero-Footprint (Park, et al., 2010), meaning the web application does not require the installation of any custom software. Henceforth, web applications that are Zero-Footprint will be referred to as ubiquitous applications because they support the largest number of electronic devices.

This thesis shows how it is possible to leverage an existing validated 3D visualization toolkit, and add a client-server layer to connect the toolkit to any number of rich ubiquitous 3D visualization applications from multiple computing platforms. As seen in Figure 1.3, this approach provides an abstraction that supports the development of ubiquitous applications

that may run on virtually any electronic device or platform having many-to-one associations to common server-side services.

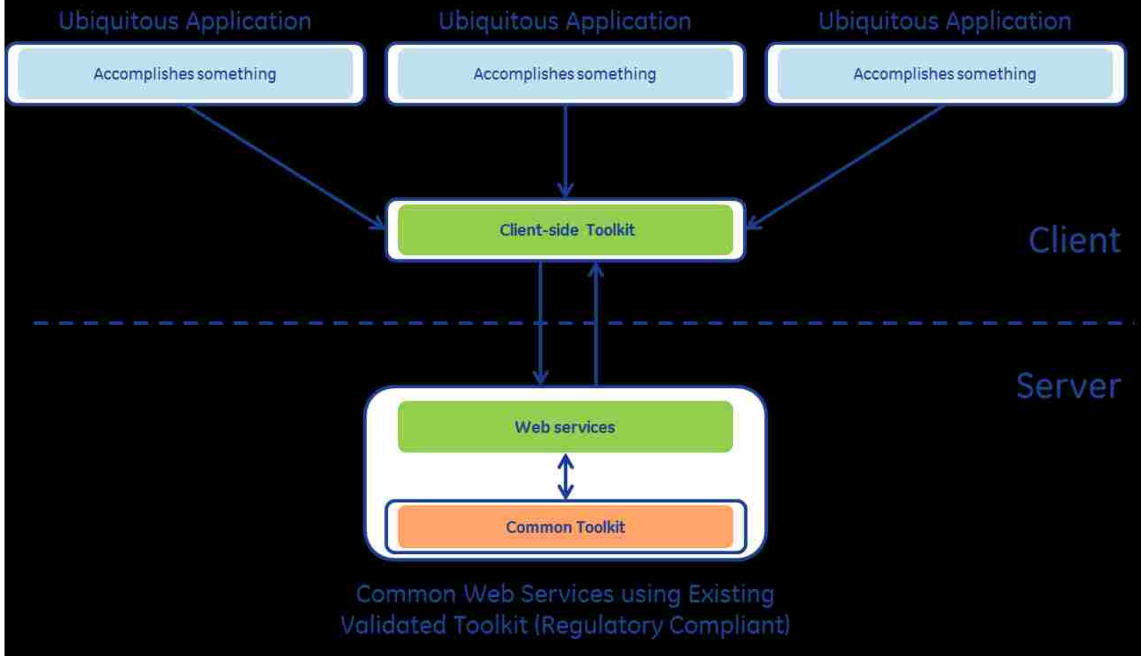

Figure 1.3: This figure depicts the architectural approach of this thesis. Using a verified visualization toolkit built for desktop applications, creating a web service layer and supporting client-side ubiquitous application development toolkit the existing verified toolkit supports an infinite number of ubiquitous applications.

In lieu of developing a platform capable of supporting the development of rich ubiquitous 3D visualization applications, this thesis provides an in-depth discussion of the benefits and capabilities of such an architecture for developing ubiquitous applications. The following chapters are organized in a way that mirrors the architecture presented in Figure 1.3, and follows a bottom-up discussion. Specifically, chapter 2 discusses the motivation of this thesis. Chapter 3 discusses the related works, current technologies, and best practices available. Chapter 4 discusses the necessary characteristics of this platform, specifically identifying the requirements of each layer. Chapter 5 is a detailed discussion of this platform, and an examination of each of the four

layers' architecture. Chapter 6 evaluates the platform based on the creation of two ubiquitous applications, from the platform fundamentals to the foundations of rich ubiquitous application. Continuing the evaluation of the platform, chapter 6 also includes a discussion of platform performance and capabilities. Lastly, the thesis ends with a discussion of future platform enhancements.

CHAPTER 2: MOTIVATION

Modern mobile technology is capable of providing continuous connectivity and the ability to work from anywhere, at any time. Today's technology lives in the Internet of Everything (IoE) which is defined by Cisco as "bringing together people, process, data, and things to make networked connections more relevant and valuable than ever before-turning information into actions that create new capabilities, richer experiences, and unprecedented economic opportunity for businesses, individuals, and countries" (Cisco). In the IoE, devices integrate seamlessly with infrastructure and supply users with data on-demand. However, not all individuals are able to reap the benefits of the IoE, particularly professionals in the healthcare space. It is common for a medical practitioner to be restricted to sitting in front of a static workstation to view image data acquired from a medical scanner. Ideally, they should be able to carry a device that is capable of delivering data independent of their location.

However, creating an environment for clinicians that utilizes the IoE is not trivial. In general, developing software in the medical domain is a complex endeavor because the healthcare space is slow to adopt the latest industry standard software practices, patterns, and technologies. The slow adoption can be largely attributed to the stringent regulations enforced on medical products. The Food and Drug Administration (FDA), which regulates medical products in the USA, requires that software running in a medical context be validated before it is sold. The FDA defines software validation to be the "confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled'" (Food and Drug Administration, 2002).

The burden of validating new medical software can be lessened by leveraging existing components which have already been validated. For medical software companies, having to validate multiple visualization applications individually is expensive in terms of time and resources. Rather, it is sensible to invest resources in the development of a core visualization toolkit. Once validated, this visualization toolkit can support an endless number of desktop applications, thereby minimizing the application validation effort. Additionally, these toolkits are dynamic as they adapt to new visualization standards to support the needs of applications.

Understanding the regulatory constraints in medical software, the complexities of medical visualization, and the importance of a validated toolkit for rapid application development and validation are key to understanding the problem facing the future of medical visualization applications. These factors make the transition of medical visualization applications from a static desktop environment to a distributed environment very complex. The ultimate goal is to support the development of medical applications that run on the largest possible number of electronic devices.

The widest commonly supported environment across electronic devices and platforms is the Internet browser. As mobile computing advances, electronic devices are increasingly interconnected. Depending on the connected network, devices are capable of local network or wide area network intercommunication. An Internet browser is a key standard application execution environment; standard Internet browsers without additional plugins, require minimal computational power. Developing applications for execution in Internet browsers allows these applications to be reached by the widest range of devices. Applications executing in Internet browsers are typically referred to as web applications.

Although the web space is a relatively new domain for diagnostic radiology, and contains its own unique complexities, the overall application architectures between the desktop and web space are analogous. As discussed, standalone medical visualization desktop application development is complicated. The addition of a general validated visualization toolkit simplifies application time to market by simplifying application development and validation efforts. Just as it is complex to develop static desktop visualization applications from scratch, it is equally, if not more, complex to develop standalone medical visualization web applications. The logical conclusion is to leverage the same validated toolkit for web applications and desktop applications. These are the motivating factors surrounding this thesis.

To support remote diagnostic radiology review, and make healthcare relevant in the IoE and Industrial Internet, existing validated visualization toolkits and supporting technologies can become the core building blocks for remote diagnostic radiology in the mobile computing domain. Leveraging these existing toolkits, technologies, and best practices has huge impacts in the field of medical software development and diagnostic radiology. The crux of the matter is regulation and the difficulties inherent in migrating toolkits from one technology stack to another are complex. However, the transition from a world of supporting static desktop applications, to a world of supporting lightweight applications running on the widest range of possible devices is important to the future of diagnostic radiology and healthcare.

This thesis explores several technologies that enable mobile applications to interface with server-side applications, thus removing the burden of implementing computationally intense operations. Specifically, this thesis proposes an architecture that “webifies” an existing validated Java 3D visualization toolkit, adding to the IoE paradigm. In this thesis,

"webification" is an architectural approach that wraps some code as web accessible resources. This is a client-server approach based on lightweight clients and fully featured servers that can be used to distribute server-side visualization technologies to a multitude of mobile clients. As a result, the existing validated toolkit is revived to support ubiquitous 3D visualization applications.

To summarize the motivation of this thesis, the overarching trend in computing today is mobile computing and the IoE. Unlike traditional static desktop applications, mobile computing provides anytime and anywhere access to data. However, due to regulatory concerns, professionals in the healthcare space are not able to reap the benefits of the IoE. The burden of validating new medical software in the healthcare space is very complex and expensive. Medical device and medical software manufacturers have figured out the key to lessening desktop application validation in the medical space is building and validating toolkits. In the world of diagnostic radiology ubiquitous applications are not the norm, and technically complex. In the same way a validated toolkit simplifies static desktop application development and validation, it can be used to simplify the development and validation of ubiquitous visualization applications. Although the problems mentioned are relevant to healthcare in general, this thesis is focused on adding ubiquitous 3D visualization to the IoE paradigm. For brevity, ubiquitous 3D visualization applications will be henceforth referred to as ubiquitous applications.

CHAPTER 3: RELATED WORKS

There are many examples of industry transformation and adoption of the Internet technology stack. Both Cisco and General Electric are defining and influencing the Internet of Everything (IoE), and Industrial Internet, respectfully (Cisco), (GE). Collectively, the IoE, and Industrial Internet are at work for a smarter world. Today, our world is data driven. Devices and machines create data, and push it to the cloud. From there, the data is analyzed and used by other devices. As we increase the amount of data collected, and introduce more machines in this world we realize the true value of interoperability, and smart machines living in a connected infrastructure.

Any successful software is built with an architecture that leverages common design patterns and best practices. The web technology stack is full of best practices and architectural philosophies for designing and creating web applications. These web applications rely on services to provide data, and allow for the creation of rich applications and user interfaces and single page applications.

This chapter examines the approaches available for a platform that supports the creation of ubiquitous applications that leverage an existing Java validated visualization toolkit. The layout of this chapter logically follows the platform starting from the existing validated toolkit all the way to the ubiquitous applications.

3.1 Existing Java Validated Visualization Toolkit

The foundation of this platform is a validated visualization toolkit. In the medical space, the creation of applications for diagnostic radiology requires each application to be validated. The United States Food and Drug Administration defines software validation to be

the “confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled’” (Food and Drug Administration, 2002). Overall, validation is a complex and tedious process all medical device software is required to perform prior to sale. For rapid application development in the medical space, it is advantageous to group common visualization code into a software component. Once validated, this software component allows applications to reuse common visualization routines or modules for the purposes of developing validated visualization applications (PC Magazine01). Leveraging these validated components simplifies the application validation process.

Although software components are a common architectural approach for code reusability in software development, they do not support rapid application development. Rather than visualization applications leveraging a validated visualization component, they should be built using a validated visualization toolkit. A software toolkit is “a set of software routines or a complete integrated set of software utilities that are used to develop and maintain applications” (PC Magazine02). In the space of diagnostic radiology, given a validated visualization toolkit, an infinite number of visualization applications can be designed, developed, and validated. The fundamental difference between a software component and a toolkit lies in its intent. Whereas a component is simply a modular building block of functionality for a larger system, a toolkit exposes an easy to use wrapper around these core blocks. For example, in the space of 3D diagnostic radiology, a component would bundle core 3D visualization application capabilities such as rendering a volume, and changing the orientation of the volume in 3D space. A toolkit however, would take these core capabilities and bundle them as a viewport. This viewport is a high level reusable construct that is built to support 3D visualization application development, like a widget. Like widgets,

viewports are simply "elements in a graphical user interface" that interact with user, see Figure 3.1 (PC Magazine03).

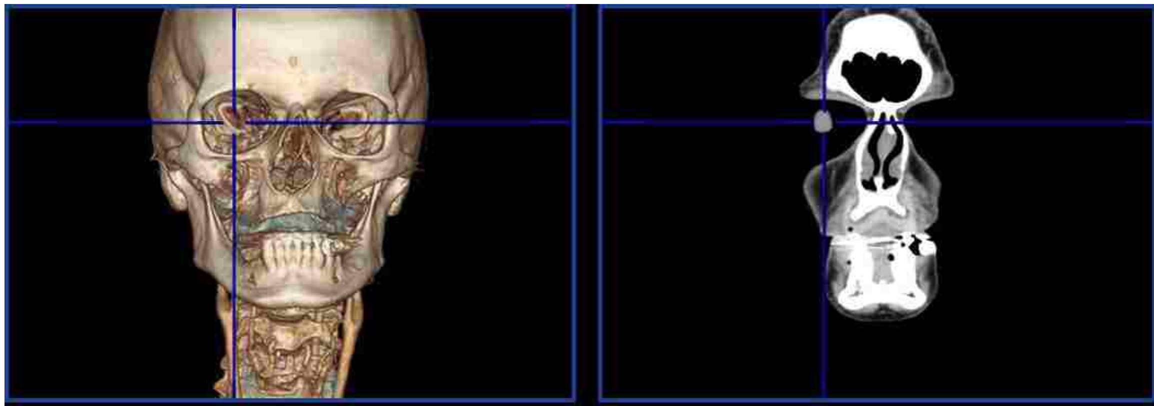

Figure 3.1: Like a widget, a toolkit provides the building blocks for application development. This figure shows two viewports included in an application. Although these two viewports look different they are instances of the same object provided by a validated 3D visualization toolkit.

To summarize, a validated toolkit is capable of streamlining application development and minimizing the scope of validation. A toolkit, unlike a component exposes a high level application construct that facilitates application development. In diagnostic radiology, applications commonly contain viewports that act as graphical user interface elements that interact with the user and serve as the foundation of an application. This thesis uses an existing proven validated Java 3D visualization toolkit as its foundation. The next sections will investigate web service technologies that can layer this validated toolkit for the purposes of creating ubiquitous applications.

3.2 Web Services Layer

Given an existing validated Java 3D visualization toolkit, and the goal of using this toolkit for the purpose of supporting ubiquitous applications, it is necessary to "webify" the high level 3D visualization application constructs. This process requires Java web services

that logically layer the toolkit, and expose the toolkit's API to local machine and external machine programs.

3.2.1 Client-Server Communication Protocols

This section surveys available client-server communication architectures that can be leveraged to expose the constructs of an existing validated Java 3D visualization toolkit for the purpose of creating ubiquitous applications. In general, client-server communication architectures can be used to distribute server-side visualization capabilities to a multitude of ubiquitous applications. The Simple Object Access Protocol, Representational State Transfer, and WebSockets are the three main client-server application protocols; each will be discussed in this section.

Both the Simple Object Access Protocol and Representational State Transfer are communication protocols that layer the Hypertext Transfer Protocol (Oracle01), while WebSockets supports machine to machine communication by layering the Transmission Control Protocol (Oracle, 2013a).

The Simple Object Access Protocol, or SOAP, is an application protocol that provides machine to machine communication across a local network or the Internet through the Hypertext Transport Protocol. Fundamental to SOAP communication is the transportation of Objects serialized and encoded in the Extensible Markup Language (Oracle01). This XML based web service communication protocol uses the standard XML schema of the World Wide Consortium to provide one-way messaging (W3C, 2007). The SOAP standard "provides the definition of the XML-based information which can be used for exchanging structured and typed information between peers in a decentralized, distributed environment" (W3C, 2007).

The Representational State Transfer protocol, or REST, is an application protocol that provides machine to machine communication across a local network or the Internet through the Hypertext Transport Protocol using a series of stateless operations. Specifically these stateless operations request the "transfer of representations of resources" through Uniform Resource Identifier parameters sent in the Uniform Resource Locator (Oracle, 2013b). The URI parameters in the URL specify a machine to machine communication contract for resource transfer. These URI parameters, known as query parameters, specify the necessary parameters for the operation the client machine is requesting. In general, a URL is broken into many components. Of note to this discussion are the web service URL, and the query portion of a URL. Given the following sample URL:

```
http://myserver:80/location/to/myservice?param=value
```

In the above URL, the full path to the web service is `http://myserver:80/location/to/myservice`, and the query is `param=value`. This query portion of a URL contains the mapping of REST query parameters to values. In this example the query parameter `param` has a value of `value`.

WebSockets provide bidirectional machine-to-machine communication through the Transmission Control Protocol, known as full-duplex communication. This full-duplex communication is the primary difference between WebSockets and Hypertext Transport Protocol communication protocols that follow the traditional request-response communication model, such as SOAP and REST (Oracle, 2013a). With the request-response workflow the exchange of data is always initiated by the client request, this does not allow the server to send data without the client first issuing a requesting (Oracle, 2013a). "This model worked well for the World Wide Web when clients made occasional requests for documents that changed infrequently, but the limitations of this approach are increasingly

relevant as content changes quickly and users expect a more interactive experience on the web" (Oracle, 2013a).

3.2.2 Java Web Services

Given the range of available client-server communication protocols, and the fact that the existing validated 3D visualization toolkit is implemented in Java, the next step is to investigate the available web service capabilities supported in Java.

Web services in Java are not new. Java 2 Platform Enterprise Edition, v 7 supports SOAP, REST, and WebSockets in the `javax.xml.soap`, `javax.ws.rs`, and `javax.websocket` packages respectfully (Oracle02). Historically, these web services supported only half-duplex communication between two machines via HTTP but have expanded to full-duplex communication over TCP with the advent of WebSockets. Although Java supports client-server communication through SOAP, REST, and WebSockets, the ease of RESTful communication makes it the top contender for new web services. Therefore, the following discussion will focus on RESTful web services in Java.

Half-duplex RESTful communication is a client-server architecture where a client requests or posts information to the server, and receives a response. These simple transactions are known as HTTP `GET` and `POST` respectfully. The HTTP `GET` method is "designed for getting information (a document, a chart, or the result from a database query), while the `POST` method is designed for posting information (a credit card number, some new chart data, or information that is to be stored in a database). To use a bulletin board analogy, `GET` is for reading and `POST` is for tacking up new material" (Hunter & Crawford, Java Servlet Programming, 2001a). Java has supported the creation of clients and servers that communicate via HTTP since the inception of Java 2 Platform Enterprise Edition. Today, in

Java 2 Platform Enterprise Edition, v 7 there are more options available for creating Java servers that communicate via HTTP. The two common frameworks for Java web services are Java Servlets and the Java API for RESTful Web Services.

Java has supported the creation of clients and servers since the inception of the `javax.servlet` package in Java 2 Platform Enterprise Edition, v 1.4 (Oracle, 2011a). Servlets communicate via the stateless HTTP, where "A client, such as a web browser, makes a request, the web server responds, and the transaction is done" (Hunter & Crawford, Java Servlet Programming, 2001b). In practice, a Java Class that provides HTTP methods as services will sub-class the `HttpServlet` Java Class using the `extends` keyword, thereby overriding the Java methods that correspond to the HTTP GET, POST, HEAD, DELETE, and OPTIONS, see Table 3.1.

HTTP Method	**Java HttpServlet Method**
`GET`	`doGet(HttpServletRequest req, HttpServletResponse resp)`
`POST`	`doPost(HttpServletRequest req, HttpServletResponse resp)`
`HEAD`	`doHead(HttpServletRequest req, HttpServletResponse resp)`
`DELETE`	`doDelete(HttpServletRequest req, HttpServletResponse resp)`
`OPTIONS`	`doOptions(HttpServletRequest req, HttpServletResponse resp)`

Table 3.1: Table of HTTP method to Java methods in the Java HttpServlet class.

In Java 2 Platform Enterprise Edition, v 6 the Java Specification Request number 311 was implemented as the Java API for RESTful Web Services provided in `javax.ws.rs` package (Oracle, 2011b). In 2008 the JAX-RS specification was defined to provide a core framework for writing RESTful web services that applies Java annotations to plain Java objects, this enhances the overall readability of service classes and methods, and aids in the overall development of RESTful web services (Burke, 2010).

The Java Enterprise Edition `javax.ws.rs` package defines the "high-level interfaces and annotations used to create RESTful service resources" (Oracle, 2011b). Table 3.2 illustrates core annotations for developing RESTful web services, and defines the annotation based on the published Java documentation.

JAX-RS Annotation	**Summary**
`ApplicationPath`	"Identifies the application path that serves as the base URI for all resource URIs provided by Path" (Oracle, 2011b).
`Path`	"Identifies the URI path that a resource class or class method will serve requests for" (Oracle, 2011b).
`Consumes`	"Defines the media types that the methods of a resource class or MessageBodyReader can accept" (Oracle, 2011b).
`Produces`	"Defines the media type(s) that the methods of a resource class or MessageBodyWriter can produce" (Oracle, 2011b).
`GET`	"Indicates that the annotated method responds to HTTP GET requests" (Oracle, 2011b).
`POST`	"Indicates that the annotated method responds to HTTP POST requests" (Oracle, 2011b).
`PathParam`	"Binds the value of a URI template parameter or a path segment containing the template parameter to a resource method parameter, resource class field, or resource class bean property" (Oracle, 2011b).
`QueryParam`	"Binds the value(s) of a HTTP query parameter to a resource method parameter, resource class field, or resource class bean property" (Oracle, 2011b).
`FormParam`	"Binds the value(s) of a form parameter contained within a request entity body to a resource method parameter" (Oracle, 2011b).

Table 3.2: Table that summarizes the core JAX-RS annotations provided in the Java package javax.ws.rs annotations.

Servlets and JAX-RS are the two Java frameworks provided in the latest Java 2 Platform Enterprise Edition, version 7, for web services. These approaches will be further evaluated in Section 5.2.2.

To summarize, Java supports three approaches for client-server communication in Java: WebSockets, Java Servlets, and JAX-RS. Due to the inherent complexities of WebSockets, and the non-standard support of the communication, only RESTful

communication technologies in Java will be further investigated. In general, web services can be viewed as providing software application remote procedure calls that communicate over a network and listen on the HTTP application layer, perform some operation, and send back a response. Due to the widespread adoption of HTTP communication, web services provide interoperability through a standard means of communication capable of communication between applications running on different platforms and machines connected by a network (W3C, 2004).

3.3 Ubiquitous Toolkit

Just as a Java desktop 3D visualization toolkit facilitates the development of validated 3D desktop applications in the desktop space, a toolkit for ubiquitous applications will facilitate the development of ubiquitous applications. Henceforth, a toolkit for ubiquitous applications is known as a ubiquitous toolkit. Specifically given the client-server architecture of this thesis, a ubiquitous toolkit will expose easy to use constructs for ubiquitous applications. Specifically, this toolkit needs to obfuscate the client-server communication, and expose a viewport construct, similar to the underlying validated toolkit. This viewport is essentially a web widget that can be placed in the graphical user interface of a ubiquitous application. Just as a desktop viewport provided by the existing validated 3D visualization toolkit, this ubiquitous toolkit supports the core features and capabilities needed by ubiquitous applications. This section will further explore the approaches surrounding ubiquitous applications in the healthcare space.

As ubiquitous applications are designed to reach the widest range of electronic devices they execute in the electronic device's Internet browser. Supporting the largest number of Internet browsers means the toolkit must leverage standard technologies. Fundamentally, applications that execute in an Internet browser are built using markups,

scripts and styles. Specifically, these are the HyperText Markup Language, JavaScript and Cascading Style Sheets. There are many versions of these standard markups, scripts and styles, and not all Internet browsers support the same versions or even contain the same Application Programmers Interface. For instance, the latest markup standard is HTML 5. Version 5 of this markup language is not supported by all Internet browsers, and interacting with the standard elements defined in the HTML 5 standard may differ between Internet browsers.

To obfuscate these subtle differences JavaScript libraries have been designed. Libraries like jQuery exist to provide an Internet browser agnostic solution to web development that supply an "easy-to-use API that works across a multitude of browsers" including Internet Explorer, Safari, Opera, and Chrome (The jQuery Foundation01).

Rich ubiquitous applications are single page applications where the elements in the HTML page, elements in the Document Object Model are modified through JavaScript. UI events in a rich application typically involve multiple DOM updates. Because DOM updates through JavaScript or even jQuery can get very complex other JavaScript libraries exist that simplify data binding between HTML and JavaScript. The Knockout JavaScript library "associates DOM elements with model data using a concise readable syntax" that includes automatic UI updates to the DOM when the state of the underlying data model changes (Knockout).

To summarize, a ubiquitous toolkit is designed to facilitate ubiquitous application development and to obfuscate client-server communication. Because an Internet browser is a standard execution environment for an application ubiquitous applications are built using markups, scripts, and styles including HTML, JavaScript, and CSS. Due to the differences in HTML support and the API differences among Internet browsers, libraries such as jQuery exit

to provide an API that supports all common browsers. In addition, because rich ubiquitous application UI interaction is complex and requires multiple updates to the DOM libraries such as Knockout exist. Such libraries support DOM to JavaScript data binding and automatic UI updates when the state of the underlying JavaScript data changes.

3.4 Ubiquitous Applications

The motivation of this thesis is the creation of ubiquitous applications that are easily developed and validated. The development of such applications is simplified by the ubiquitous toolkit, and the scope of application validation is lessened by the ubiquitous toolkit leveraging an existing validated visualization toolkit.

To support ubiquitous application execution on the largest range of electronic devices, execution must not require any non-standard plugins to a device's Internet browser. This means the application must not leverage browser plugins such as Adobe Flash and Java. This means ubiquitous applications are Zero-Footprint, requiring a minimal execution environment. Also, because these ubiquitous applications will be leveraging a ubiquitous toolkit that communicates with web services for 3D visualization, these applications are thin-Client.

Ubiquitous applications are built using markup, scripts and styles that run in a device's Internet browser. Due to the complexities surrounding rich, single page web application development JavaScript libraries exist that provide an application framework. Such frameworks include AngularJS (Google) by Google, and Durandal (DURANDAL). These frameworks encourage separation of concerns between web application UI logic and data through patterns like the Model-View-Controller.

In the software domain, thin-Client and Zero-Footprint are related but not mutually exclusive software architectural concepts. The term thin-Client is an architectural approach where the software application contains little to know business logic, and is focused on the application User Interface. On the other hand, Zero-Footprint is an architectural approach where applications require no special software to execute (Park, et al., 2010). Therefore, the thin-Client architecture specifies a distinct separation of concerns between application and UI logic, and a Zero-Footprint architecture limits the execution environment of an application. Because a thin-Client architecture does not restrict the execution environment of an application, a thin -Client architecture is a type of a Zero-Footprint architecture. In fact, a Zero-Footprint architecture can be categorized as thin or thick.

Zero-Footprint applications are not rare; we interact with them many times a day. Since the Zero-Footprint architecture means the application requires no special software to execute, a Zero-Footprint application must execute on a technology stack common to all devices. The only consistent environment across devices is an Internet browser. All Internet browsers support the execution of standard applications written as web pages. These web pages, often referred to as web applications, are all written on the same technology stack, and minimally include technologies such as: HyperText Markup Language, Cascading Stylesheets, and JavaScript that execute through a web browser application (Landgrave). These standard client-side markups, styles, and scripts allow for the creation of platform and device agnostic applications.

Many of today's web applications are by definition Zero-Footprint because they run on an Internet browser and do not require the installation of any extra software. However, not all web applications fall into the Zero-Footprint category. Just because an application executes in an Internet browser does not mean it is Zero-Footprint. A Zero-Footprint

application must not require the existence of any non-standard Internet browser plugins. Any web application that requires Adobe Flash or a Java Runtime Environment is not Zero-Footprint.

There are two subcategories of Zero-Footprint application architectures. Zero-Footprint application architectures can be categorized as either thin-Client, or Thick-Client. Although thin-Client and Thick-Client are generic architectures methodologies for any technology stack, this discussion will tied to the domain of Zero-Footprint web applications. The categorization of a Zero-Footprint web application depends only on the amount of application business logic contained in the application code.

In the thin-Client architecture, the client-side code is only responsible for User Interface controls, and the application business logic Application Programmer Interface is offloaded to another software program, typically executing on a server. This server-side application business logic API is usually exposed as web services that are available to the thin-Client application, typically through the HTTP protocol. Conversely, in the Thick-Client architecture the client-side code is responsible for UI and application business logic. The application business logic API is local to the Thick-Client application itself. Therefore, as shown in Table 3.3, the categorization of a Zero-Footprint application as thin-Client or Thick-Client can be made solely based on the location of the application business logic API relative to the UI-application logic.

Zero-Footprint Application Type	Application Business Logic API
thin-Client	External
Thick-Client	Internal

Table 3.3: Zero-Footprint Application Architecture Types and categorization based on business logic location.

As stated, a Zero-Footprint application is a web application that is built using only standard markup, styles and scripts executing in an Internet browser. This limits the technology stack and language choices to languages supported universally by Internet browsers, and must not require Internet browser plugins. For web applications written in HTML and JavaScript, the business logic of Thick-Client web applications are JavaScript functions whereas the business logic of thin-Client web applications are typically web services invoked via a specially formatted Hypertext Transfer Protocol request messages that often originate from the JavaScript `XMLHttpRequest` Object. The `XMLHttpRequest` Object is used to exchange data synchronously (SJAX) or asynchronously (AJAX) between the web application and a server. The significance of the `XMLHttpRequest` Object is that it is used to dynamically update the content of a web page with web service data without reloading the web page, supporting single page web applications (W3Schools01). It is common for both thin-Client and Thick-Client web applications to leverage web services; however, thick-Client applications are more reliant on web services and usually require constant client-server communication.

As with all software architectural approaches, thin-Client and Thick-Client architectures have their benefits and drawbacks. Table 3.4 lists some of the major pros and cons of thin-Client and Thick-Client architectures. To distinguish thin-Client and Thick-Client architectures for Zero-Footprint web applications each characteristic may not be a pro or con for both.

Architecture Characteristics	thin-Client	Thick-Client
Web Application Complexity	Pro	Con
Server-Side Complexity	Con	Pro
Code/Algorithm reusability	Pro	Con
Intellectual Properties Protection	Pro	Con
Network Connection	Con	Pro
Network Bandwidth	Con	Pro
Web Application Startup	Pro	Con
Web Application Responsiveness	Con	Pro
Security (Server-Side)	Con	Pro

Table 3.4: Benefits/Drawbacks of thin-Client versus Thick-Client architectures for web applications

One of the most compelling benefits of a thin-Client, Zero-Footprint architecture is implementation hiding. Proprietary algorithms are hidden from view because they can exist in the server-side API, and do not execute locally in the Internet browser. This provides ultimate algorithms protection in a way not possible with JavaScript obfuscation.

Another noteworthy benefit of thin-Client architecture not captured in Table 3.4 is multi-application language support. For a Zero-Footprint application restricted to standard web technologies this is not a primary concern; however, it is an additional benefit of a thin-Client approach. Since a thin-Client approach leverages a central software application, like a server, for application business logic, the overall language choices for the thin-Client applications are limitless. Specifically, for servers that expose web services, any language capable of communicating by the Hypertext Transfer Protocol is a candidate language for developing a thin-Client application. This fact makes the overall application business logic reusable across the software language space.

Park et al. (2010) expressed one of the major benefits of a Zero-Footprint viewer for medical imaging lies in the fact that such a technology provides remote image view while adhering to the Health Insurance Portability and Accountability Act Security Guidance for

Remote Use of and Access to Electronic Protected Health Information (Park, et al., 2010), (HHS, 2006). Specifically, any device that displays any patient information, text or images must protect said information. This is difficult in a medical imaging viewer because the images being displayed are sensitive. The approach of Park et al. is the creation of a Zero-Footprint mobile image display. This approach sends rendered images encoded as jpeg from the server to the client, and displays them. The authors' Zero-Footprint approach has the following characteristics:

a. "It is less restricted to the browser vendor used. Some browser incompatibilities issues might need to be resolved but it has the benefit of working in a variety of Operating Systems (OS) and browsers.

b. Quality of the images is not enough for Radiological readings; however it is suitable for review and training.

c. Standard web protocols are used, no extra knowledge is required for handling the communication between the server and the client.

d. The cache mechanisms are inherent from the browsers and the settings by the user. This could be an issue if a high number of images are needed to be downloaded.

e. The bandwidth usage can be considered low because there is no need to send the native DICOM images to the client, only the jpeg images.

f. Minimal requirements are set for the hardware of the client PC. Because the processing is done at the server side, the clients do not require having high-end components in order to display the images" (Park, et al., 2010).

Of the six characteristics: ***a***, ***c***, and ***f*** define a Zero-Footprint system. Characteristics ***b*** and ***e*** deal with Zero-Footprint medical image display, and require further examination.

A Zero-Footprint system cannot simply be considered a low-bandwidth because Zero-Footprint applications come in two flavors: Thick-Client, and thin-Client. A Thick-Client

over a thin-Client approach is less reliant on an external application, such as one running on another machine, and does not require constant communication to perform operations. On the other hand, a thin-Client Zero-Footprint architecture has a heavily dependency to an external application to perform operations, and requires almost constant communication. Thus, with a thin-Client Zero-Footprint architecture it is difficult to agree with characteristic ***e***.

The claims for why such a Zero-Footprint for timely evaluation of stroke patients should send images as jpeg is an understandable design characteristic (approach characteristics ***b***); however the rational is incomplete. A Zero-Footprint medical imaging viewer should not be limited to image review and training. An alternative Zero-Footprint medical imaging viewer framework will be explored in Section Chapter 5: APPROACH, where image quality does support radiological readings and large image traffic is not an issue (Park, et al., 2010).

For Park et al. the main purpose of a Zero-Footprint medical image viewer is securing patient privacy, and a Zero-Footprint approach is perfect because images are not stored on the viewing device, only currently relevant information is displayed on the viewing device via an Internet browser (Park, et al., 2010). This is a great solution for the issue of protecting patient information; however, the advantages of a Zero-Footprint viewer framework go much further, and do not have the limitations voiced by Park et al.

The major benefit of a thin-Client architectural approach is the reusability of business logic. However, this benefit is also a drawback. With all business handled by web services running in a remote application the architecture requires a stable and reliable connection to the machine hosting the web services when making HTTP requests. This often makes a thin-Client approach appear as “chatty”, and may require a high bandwidth connection to make the application usable.

Overall, ubiquitous applications are web applications that execute in an Internet browser. Because the Internet browser is a common execution environment across devices it is the sensible environment for applications to reach the widest range of devices. The overall application technology stack includes HTML markup, JavaScript scripts, and CSS styles. Although this stack is a standard execution environment, rich, single page application development is not simple. To simplify development of these applications there exist many application frameworks libraries, including: AngularJS, and Durandal. In general, ubiquitous applications are known as Zero-Footprint because they are built using the standard browser technology and do not require the installation of any browser plugins. Ubiquitous applications can be categorized as thin-Client, or Thick-Client based on the location of the underlying application business logic. For 3D visualization, a thin-Client ubiquitous application leverages a server for 3D visualization services; rather a Thick-Client ubiquitous application leverages a client-side library for 3D visualization services. Zero-Footprint applications have many advantages in the diagnostic radiology space; Park et al. chose a Zero-Footprint architecture for the timely evaluation of stroke patients for many reasons. To summarize, their Zero-Footprint approach is less sensitive to Internet browser inconsistencies, it limits bandwidth usage by minimizing the number of images needed to download, and this approach does not save patient information on the device (Park, et al., 2010).

CHAPTER 4: SYSTEM CHARACTERISTICS

The primary goal of this thesis is the creation of an architecture that leverages an existing validated visualization toolkit for the purpose of supporting the creation of ubiquitous applications for diagnostic radiology. Therefore, rather than developing such ubiquitous applications, the goal is to support their development by streamlining development and minimizing the validation processes via a ubiquitous toolkit. Only with the mindset of toolkit development in place can one truly develop the set of software routines and utilities that are the foundations of ubiquitous applications capable of supporting the greatest number of ubiquitous applications executing on the widest range of devices.

Proper software development must start with requirements. Only after collecting software requirements is the intent of the software understood. Using a Behavioral Driven Development approach "I focus on the goals of my users and the steps they take to achieve those goals" (Satrom, 2010). This ensures the software requirements are based on real customer uses cases. When developing the supporting architecture for ubiquitous applications for 3D diagnostic radiology, all system characteristics are focused on the ubiquitous toolkit.

This chapter examines the system characteristics for this supporting architecture. Specifically, this is a bottom-up requirements discussion that traverses the architecture layout from the existing validated Java toolkit to the ubiquitous applications. Each section discusses the requirements necessary for supporting ubiquitous application development.

4.1 Existing Java Validated Visualization Toolkit

This section is a discussion of the core platform features a validated 3D visualization toolkit must provide for the support of rich 3D visualization applications. As described in the overall layout of this ubiquitous application supporting architecture, this toolkit will be "webified"; therefore, it must provide the necessary features. Following is a discussion of eight core 3D visualization features needed by diagnostic radiology applications.

4.1.1 Dynamic Viewport Resizing

Dynamic viewport resizing allows the viewport to be resized to any positive, non-zero width and height, see Figure 4.1. This feature sets the size of the server-side viewport that map to one or more application viewports. Changing the size of the application viewport without notifying the server-side viewport will modify the image aspect ratio, thereby stretching the application viewport image.

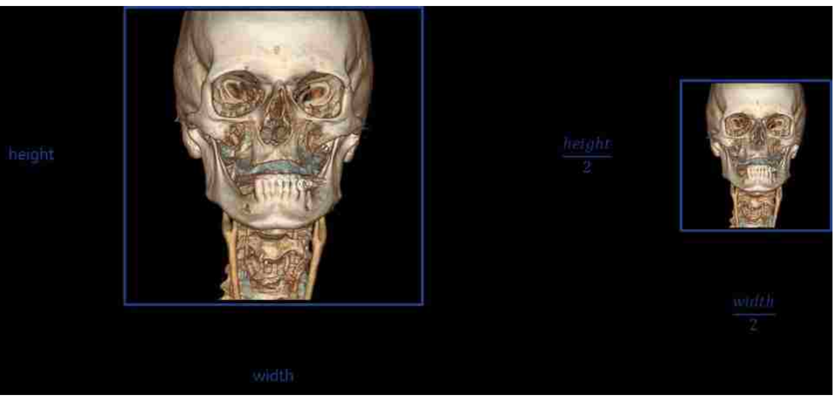

Figure 4.1: Dynamic viewport resizing feature allows an application viewport to be resized to any positive, non-zero width and height.

4.1.2 Rendering Modes

The platform supports different viewport rendering modes. These render modes are applied to the server-side render engine, and affect the style of the image rendered. The default render style is Multi-Planar Reformat; however the render style can be set to Maximum Intensity Projection and Volume Rendering before or after a dataset is loaded in the viewport, see Figure 4.2. The middle image is a MIP view that supports dynamically changing the view thickness in millimeters.

Figure 4.2: The platform supports from left to right: MPR, MIP, and Volume Rendering render styles.

4.1.3 Preset Camera Views

The platform supports the setting of the render engine's camera eye point, look point, and up vector components to predefined anatomical directions and views, including: anterior, posterior, superior, inferior, right, and left.

4.1.4 Coordinate Transforms and Volume Geometry for Graphics

Custom Visualization Components are client-side JavaScript graphics that are painted over the application viewport. These graphics need to update during application viewport interaction so that they appear to stick to their location. These graphics can be

integrated into the viewport paint cycle and require coordinate system transforms. Specifically, 3D Visualization Components require transforms that convert coordinates between display and world coordinates, between 2D and 3D. This allows Visualization Component to dynamically update their position in the application viewport during viewport interaction through the world to display and display to world coordinate transforms.

A visualization component's world coordinate does not change during viewport interaction. It only changes when it is interacted with. For example, a 3D reference cursor Visualization Component is used to show the same world point in all application viewports. Figure 4.3 shows a 3D reference cursor across four viewports. This 3D reference cursor is set to the anterior side of the right eye.

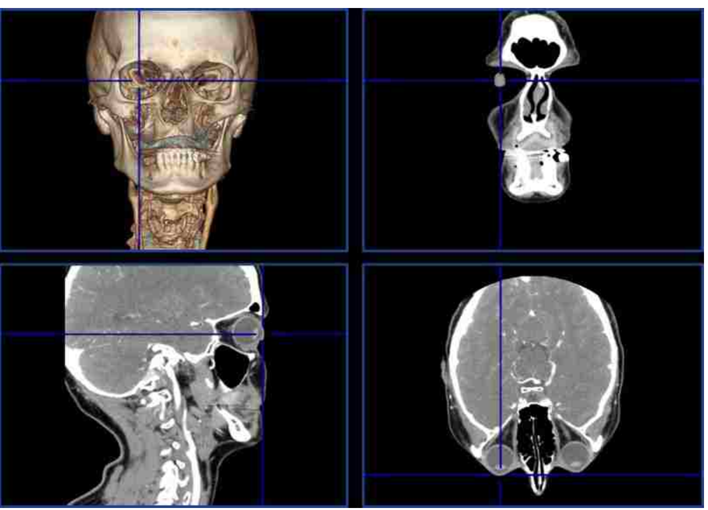

Figure 4.3: The 3D reference cursor Visualization Component shares the same world coordinate across the four application viewports. When a user interacts with any of the cursors, their association cause the position of the cursor in the other viewports to update.

In addition to coordinate transforms, rich 3D visualization applications require the volume geometry context. This enables smart Visualization Components that have an understanding of the boundaries of the volume loaded in the viewport. Specific to the 3D reference cursor, the cursor should only update its world position if it exists within the

confines of the volume. Figure 4.4 shows the 3D cursor Visualization Component's world coordinate outside the volume geometry, shown by the bounding box Visualization Component.

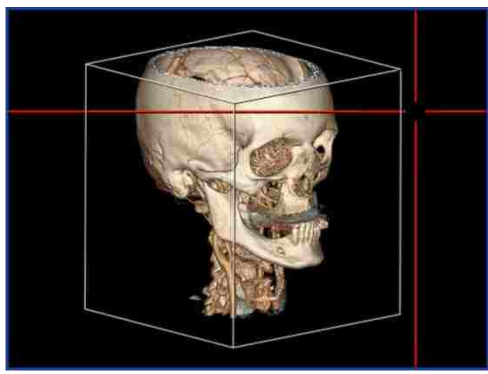

Figure 4.4: The color of the 3D reference cursor Visualization Component changes when the cursor's world coordinate does not exist within the confines of the volume (defined by the white bounding box). This bounding box is another Visualization Component that updates during viewport interaction.

4.1.5 Camera Manipulation

The platform supports receiving and setting the render engine's camera eye point, look point, and up vector together, or independently. This concept is further discussed in beginning of Section 5.2.1.

4.1.6 Mouse Based Application Viewport Interaction

Mouse based application viewport interaction supports viewport pan, zoom, window width and window level, trackball for changing the camera orientation, and paging through slices orthogonal to the cut-plane. The platform allows the mouse interaction to be set to the aforementioned. Pan, zoom, trackball, and paging involve manipulation of the render engine's camera eye point, look point and up vector components.

4.1.7 Render the Current Viewport Render Engine

Ensuring the quality of the image displayed on the application viewport is possible through a render image web service that renders a snapshot image of the current render engine state. This web service allows the application toolkit to request the appropriate full size rendering compressed using a lossy or lossless algorithm.

This feature allows the client-side thin-Client and Zero-Footprint application toolkit to display high quality images in the application viewport during periods of stationary mouse interaction.

4.1.8 Save and Restore Viewport State

The platform supports the saving of the current state of the application viewport by returning the state of the server-side viewport as XML. This state can be set on the server-side viewport to restore the state of the application viewport.

4.2 Web Services Layer

To support ubiquitous applications that leverage an existing validated Java visualization toolkit for the core 3D features described in Section 4.1, the existing validated toolkit must be "webified". This architectural approach adds a service layer that logically sits on top of an existing codebase. In this thesis, this is a web service layer that wraps the features of the existing validated toolkit, and exposes its features as web accessible resources.

Since ubiquitous applications are intended to execute on the largest range of devices possible the web service layer must not make any choices that limit the use of the validated

Java visualization toolkit. Specifically, this layer must use standard protocols and best practices to support ubiquitous applications.

Appendix A lists the core 3D features described in Section 4.1 along with categorizing the HTTP operation. The appendix also lists the web services that support each core feature, including the web service return type.

4.3 Ubiquitous Toolkit

3D visualization algorithms are complex, require adequate hardware resources, and medical datasets are not small. Therefore, in regards to Java 3D visualization algorithms, it is not simple to port these algorithms to a client-side web application language like JavaScript. Not only would this exercise be very complex this would limit the number of devices capable of preforming 3D visualization because of hardware requirements. Plus, because the image datasets being visualized are large, the transfer of images to the client device is inefficient. Therefore, due to the overall hardware resource limitations of these client side devices, and the complexities of the 3D visualization algorithms it is best to leverage an existing 3D validated toolkit for these web applications.

However, simply providing a set of web services for ubiquitous applications is only half the story. Just as porting a Java visualization toolkit to JavaScript is complex, building a ubiquitous application from web services is also complex. Therefore, to facilitate ubiquitous applications it makes logical sense to provide a ubiquitous toolkit.

The most fundamental 3D visualization construct is a viewport. A viewport is a graphical element placed in a user interface that simulates physical interaction with a volume. This construct can be thought of as a widget that is self-contained, and exposes an API for association with other user interface elements.

Therefore, this ubiquitous toolkit must expose a viewport construct for easy inclusion in ubiquitous applications. To be truly successful, a viewport construct must obfuscate its reliance on the web services, essentially by being a web service wrapper. Just as viewport constructs are the building blocks of any diagnostic radiology application, the exposed 3D viewports are the building blocks of ubiquitous applications for diagnostic radiology.

Because the ubiquitous toolkit provides core 3D visualization services by leveraging the existing validated visualization toolkit's features through web services, one of its unique traits lies in viewport interaction performance. Following is a discussion of the unique the viewport interaction performance feature the ubiquitous toolkit must provide.

4.3.1 Viewport Interaction Performance (Image Return Size and Mime Type)

Ubiquitous application viewport interaction performance is a balance between frames per second and image quality. In regards to viewport interaction performance, frame rate and image quality are inversely related. Therefore, to improve interaction performance, image quality must decrease. The ubiquitous toolkit must supports changing the interaction quality by decreasing the size of the image rendered, and changing the compression type of the image between `image/jpeg` and `image/png`. Both these performance features influence the image transportation time from the server to the client by decreasing the image render time, and the return image file size. Changing the size of the image rendered improves performance by also decreasing the render time. Figure 4.5 shows how the render size changes viewport quality. For the viewport on the right, the return image width and height are a forth the application viewport size. Because the image is stretched to fit the application viewport, the viewport appear pixelated.

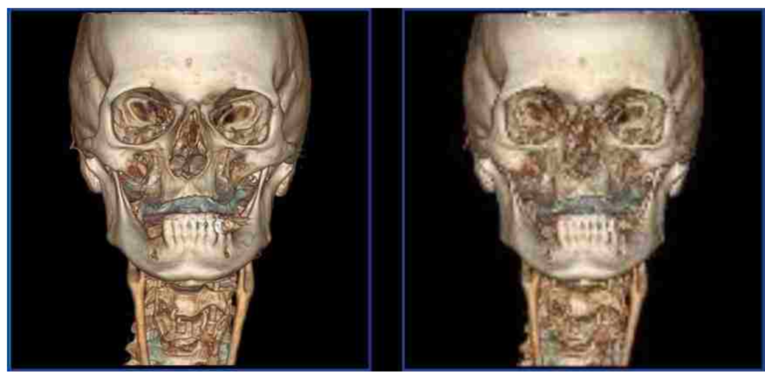

Figure 4.5: The viewport on the left contains an image of equal width and height. The viewport on the right contains an image one forth the width and height.

These two performance features mean that web services will not limit the range of applications based on software safety classification, and will be designed to support diagnostic review through ubiquitous applications

4.4 Ubiquitous Application

Ubiquitous applications are 3D applications capable of executing on the widest range of devices for the purpose of diagnostic radiology. Therefore, these applications must allow users to visualize and interact with volumes in a way that supports diagnostic review ubiquitously.

From the standpoint of end users, the expectation is these ubiquitous applications support diagnostic review from workstations, to their mobile devices. Therefore, ubiquitous applications must be validated applications that display a high quality representation of the current state of the volume being viewed and modified. These applications must be capable of advanced visualization and interaction, and include core features necessary for diagnostic radiology. Therefore, the core features necessary for ubiquitous applications mirror those described in Section 4.1.

CHAPTER 5: APPROACH

This chapter examines the details of this client-server architectural approach for supporting ubiquitous applications. Specifically, this is a bottom-up architecture discussion that traverses the architecture layout from the existing validated Java toolkit to the ubiquitous applications. Each section discusses the details surrounding the layer in the architecture necessary for supporting ubiquitous application development. As the discussion progresses through the architecture, the details of high-level architecture client-server diagram supporting an infinite number of unique ubiquitous applications shown in Figure 5.1 will be filled in.

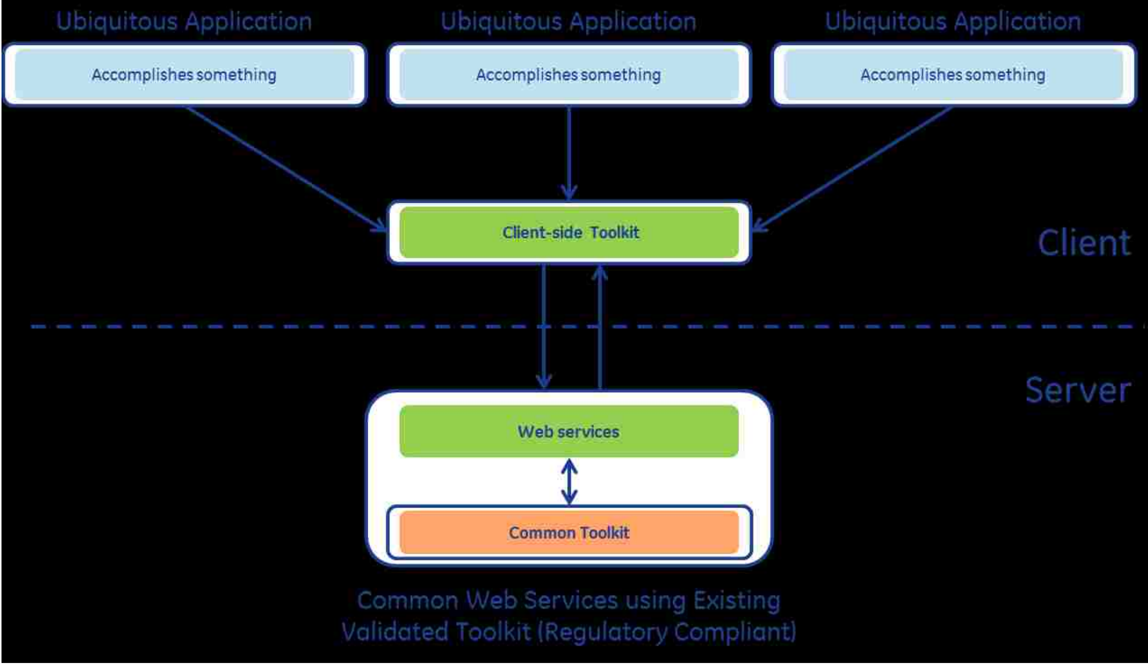

Figure 5.1: This figure depicts the high-level architectural approach of this thesis. Using a verified visualization toolkit built for desktop applications, creating a web service layer and supporting client-side ubiquitous application development toolkit the existing verified toolkit supports an infinite number of ubiquitous applications.

5.1 Existing Java Validated Toolkit

The foundation of this client-server architecture supporting ubiquitous application is an existing validated Java visualization toolkit. Importantly, this toolkit is not designed for supporting web applications, ubiquitous applications. This validated toolkit is designed to support rich Java desktop applications for diagnostic radiology. Being a validated toolkit, it provides the foundation for rapid creation of rich Java desktop applications that require less validation then traditional desktop applications not leveraging a validated toolkit.

In the context of this architecture, the existing Java validated visualization toolkit is simply considered a third party library providing 3D visualization features. Bundled as Java ARchive files, this toolkit can be leveraged by adding the JAR files to the server-side web services code, see architecture in Figure 5.2. This is the foundation of this thesis as it is a time tested validated toolkit proven to support many applications running on many platforms. Importantly, it provides all the necessary features.

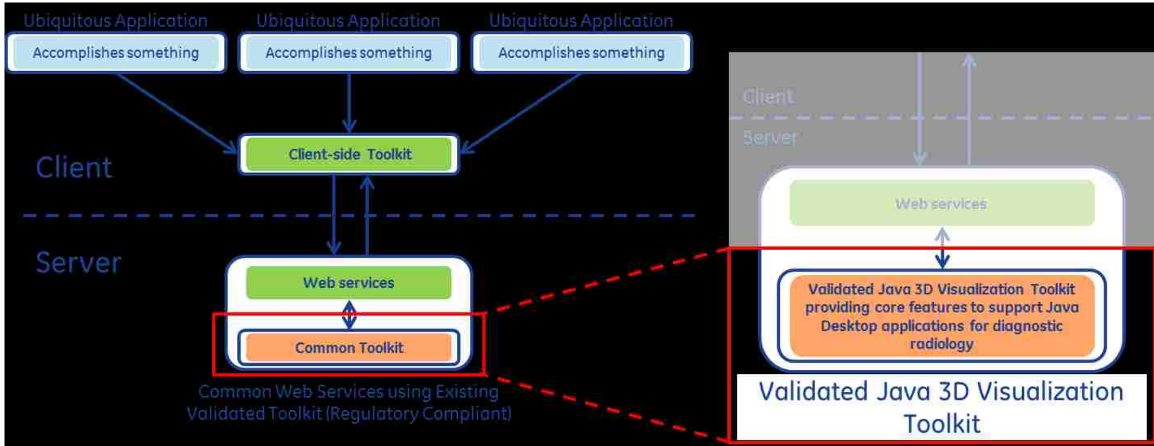

Figure 5.2: This figure highlights the existing validated toolkit used in this architecture. In the high level architecture, this toolkit will be layered by a web service layer for the purpose of "webifying" the existing validated toolkit.

5.2 Web Service Layer

The overall goal is to develop an architecture capable of supporting ubiquitous applications that support an existing Java validated visualization toolkit by exposing the features of the toolkit via a standard web communication protocol. Specifically, these web services are generic meaning they can be consumed by applications written in any language able to communicate via the chosen protocol. In this architecture, the collection of similar web services provides necessary application functionality in one place. In an enterprise setting, this allows all applications running on any device in the network to share the same core services ensuring consistent 3D volume visualization and interaction behavior.

For this client-server architecture the thin-Client Zero-Footprint pattern is the preferred choice over the Thick-Client for 3D Zero-Footprint applications for several reasons. The thin-Client architectural pattern minimizes re-implementation of existing algorithms. In a thin-Client architecture, existing algorithms can be wrapped by a service layer that exposes functionality as web services. Figure 5.3 shows the thin-Client pattern for the client-server architecture where the existing implementation is viewed as a black-box from the perspective of the client-side code. Another advantage is implementation hiding. A thin-Client pattern hides proprietary algorithms in the server-side black-box living under the service layer, shielding them from view. Contrary, the Thick-Client pattern does not hide algorithms from view because algorithms exist and execute locally on the client machine as JavaScript. Therefore, the algorithms are viewable by viewing the web page's source code through an Internet browser, even if the JavaScript is obfuscated. In this architecture for ubiquitous applications, the web-services are designed to be leveraged by a ubiquitous toolkit. As seen in Figure 5.3 the ubiquitous toolkit has direct access to the service layer but does not have access to the web validated Java visualization toolkit.

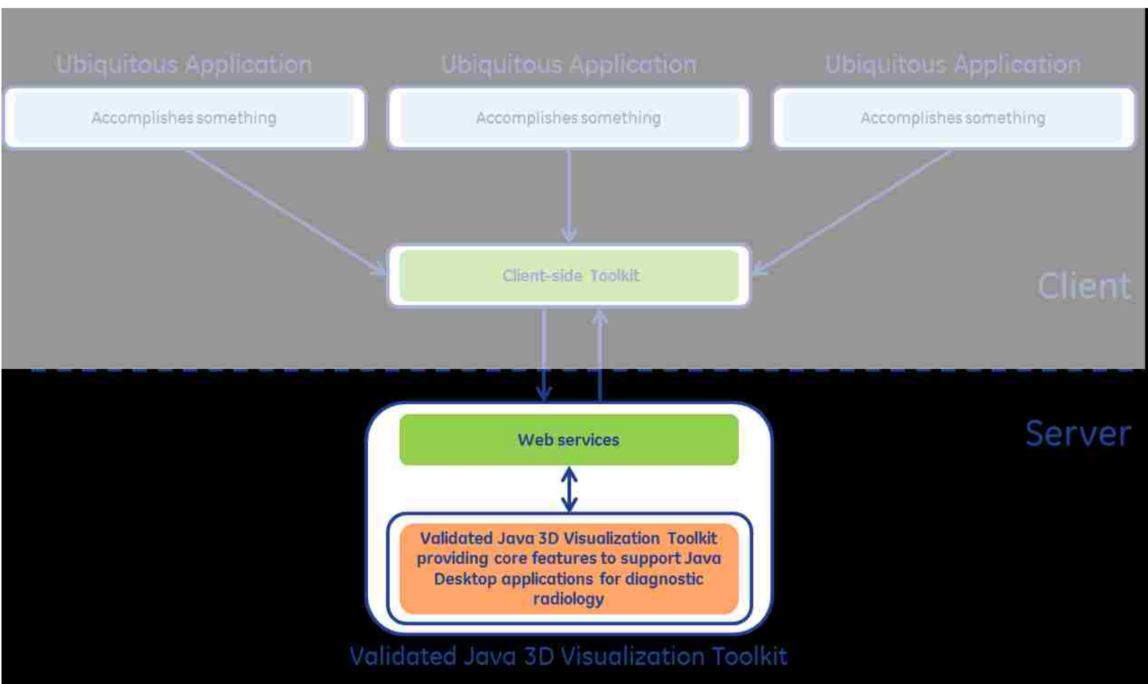

Figure 5.3: This figure highlights the web service layer architecturally wrapping an existing validated Java 3D visualization toolkit (viewed as a black box).

After choosing the thin-Client client-server architecture pattern, the next design choice is the web service client-server communication protocol. As noted in Section 3.2.2, there are three common client-server communication architectures: WebSockets, SOAP, and REST. These three communication standards for client-server communication have their advantages and disadvantages. In general, WebSockets provide higher throughput than REST and SOAP due to decreased overhead. Additionally, unlike REST and SOAP, Web Sockets provides full-duplex communication, allowing the server to push data to the client. However, WebSockets should not be thought of as the first and only solution for client-server communication; rather, WebSockets should be used when a use case exists that can be best solved through WebSockets. For these reasons, the use of WebSocket for this architecture is not a primary consideration until it is proven that REST or SOAP will not support ubiquitous applications.

Representational State Transfer and Simple Object Access Protocol both provide an interoperable means for client-server communication using the Hypertext Transfer Protocol. REST provides machine-to-machine communication that provides remote access to server-side resources through a set of stateless operations, often HTTP `GET` and `POST` (Laine). SOAP leverages the Extensible Markup Language for object communication between machines. Transport of objects serialized and transported as XML produces readability but increases the amount of data transported, and requires additional XML and object conversion overhead.

RESTful web services have several advantages over SOAP for client-server communication. RESTful communication does not require XML for data transfer. Marshaling and unmarshaling objects to and from XML requires client-side and server-side parsing of the XML document and therefore has added computation and data transportation overhead (Laine). RESTful web services support the transportation of objects using the JavaScript Object Notation. JSON provides a "lightweight, text-based, language-independent data exchange format that is easy for humans and machines to read and write" (Kotamraju, 2013). Another advantage of RESTful communication is inherent in Internet browsers. Internet browsers are well apt at image retrieval, and when the `src` attribute of an `image` HTML element is changed, the browser automatically issues a HTTP `GET` operation. Therefore, retrieving an image from a RESTful web service requires little client-side code.

As mentioned, RESTful web services are intended to be stateless. This means each web service call should be independent from prior web service calls. This is great in theory, but in practice this pattern is not ideal for a 3D imaging toolkit. The next section will investigate why a "stateful" RESTful web service approach is appropriate for ubiquitous applications by illustrating the advantages of 3D visualization web services that store the state of a ubiquitous application viewport during a mouse drag operation.

5.2.1 "Stateful" Versus Stateless RESTful Web Services

Given a working ubiquitous application, a clinician is viewing a volume rendering of a CT head study, and wants to change the view from an anterior view to a right view using the mouse on the computer, see Figure 5.4.

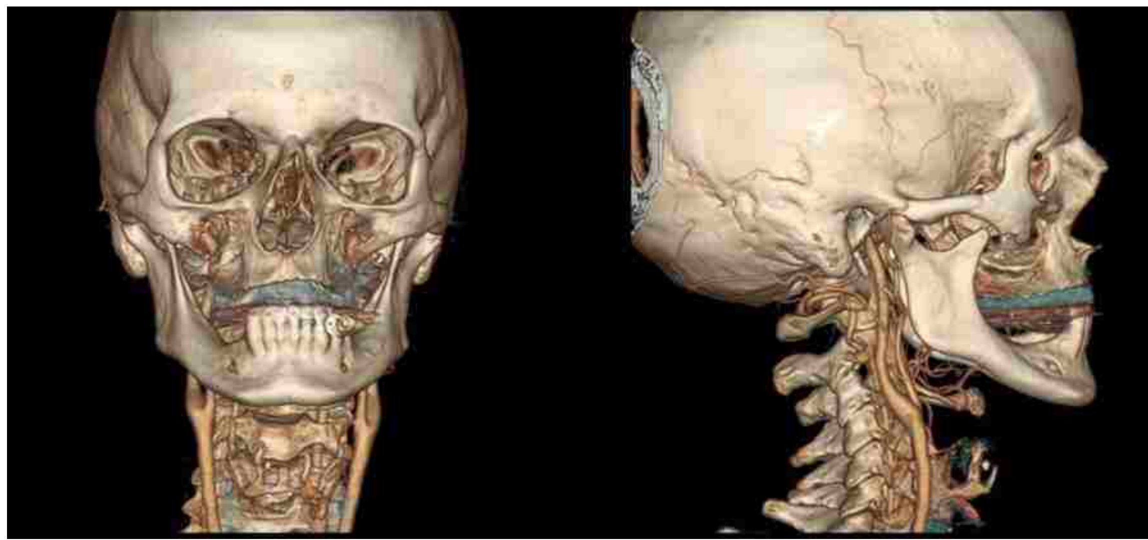

Figure 5.4: The image on the left shows an anterior view volume rendering. The image on the right shows a right view volume rendering.

Before discussing the underlying RESTful web service design several volume rendering terms must be defined.

- Render engine
- Render style
- Camera

A **render engine** is responsible for producing 3D views. In this context, a render engine is able to produce an output image based on its state. A render engine state is minimally defined by a loaded DICOM dataset(s), a render style, dimensions, and camera attributes.

A **render style** is an important component of any render engine. A render engine render style defines the type of volume being rendering. In this context, the output is an image that is a rendered with a render style such as multi-planar reformat, volume rendering, or maximum intensity projection.

In the context of volume rendering the **camera** defines specific attributes, including: eye point, look point, and up vector. Together these define the orientation of the volume, and simulate what the viewer would see. Eye point and look point are points defined as a 3D world point, and define the point that simulates the location of an eye that is looking at a specific point. The up vector defines the orientation of the camera in 3D world coordinates. These three camera attributes define the views of the render engine in a three dimensional world, see Figure 5.5. The camera attributes allow the render engine to render views that are zoomed, translated, rotated, or any combination of the aforementioned operations. For example, if in Figure 5.5 the look point and eye point both are moved 7 millimeters in the positive superior direction, the result is a translated or pan image. Figure 5.6 visualizes the 7 millimeter translation in the positive superior direction.

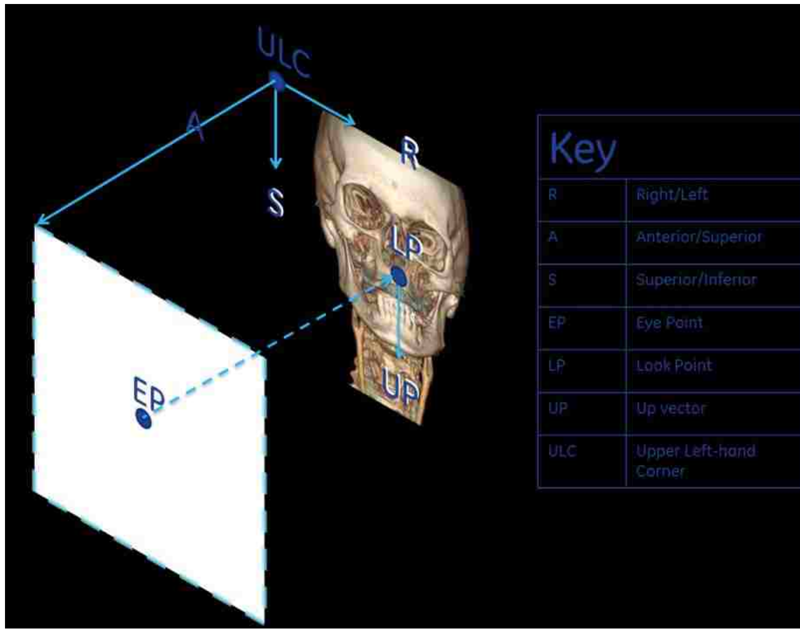

Figure 5.5: This figure visualizes the three components of a 3D medical render engine's camera: Eye Point, Look Point and Up Vector.

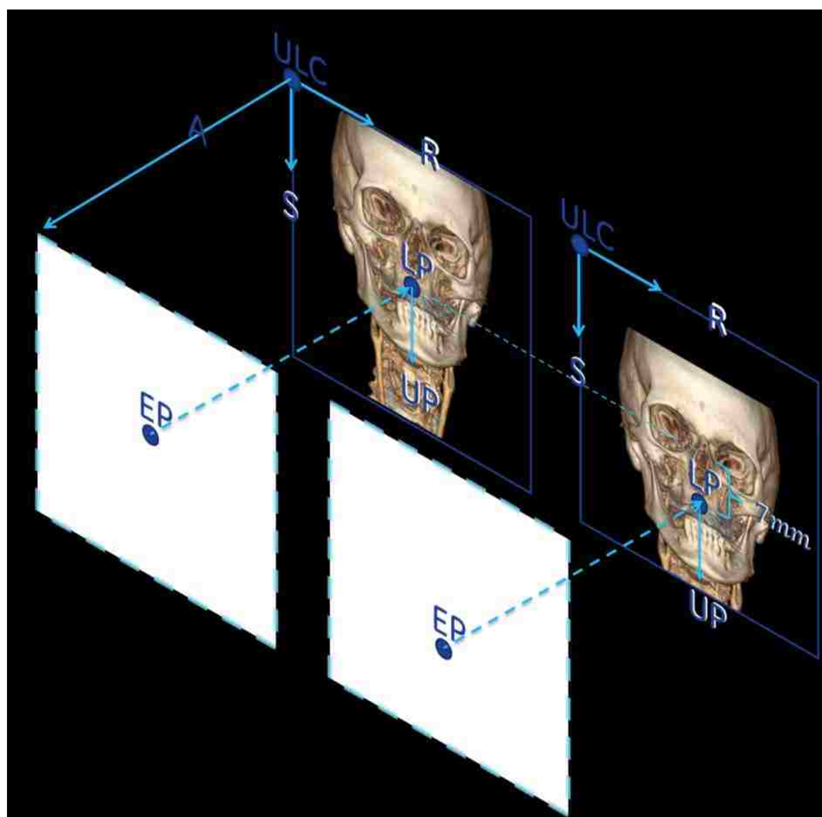

Figure 5.6: This figure visualizes a render engine camera translation by 7 millimeters in the superior direction by updating the camera Eye Point and Look Point.

Given the definitions of render engine, render style, and camera, the following is a discussion of "stateful" versus stateless RESTful web services. If the underlying web service layer utilizes RESTful web services that are stateless, each mouse event during the mouse drag would issue requests to the `mousemove` web service in the following URI form:

```
GET /mousemove?datasetID={ID}&renderStyle=VOLUME&x={mouse x}&y={mouse y}
```

This stateless RESTful `mousemove` web services requires URI query parameters for each property that specify the volume to be loaded and manipulated. Specifically, each call to the `mousemove` web service requires four URI query parameters:

1. `datasetID`- the DICOM data identifier
2. `renderStyle`- the volume render style
3. `x`- the x component of the mouse event relative to the HTML `canvas` upper left hand corner
4. `y`- the y component of the mouse event relative to the HTML `canvas` upper left hand corner

To be completely stateless the underlying RESTful web service needs to perform five operations in order to service each `mousemove` call:

1. Load the specified dataset,
2. Set the render style,
3. Apply the mouse move through render engine camera properties (eye point, look point, and up vector)
4. Perform a rendering, and send the response image
5. Cleanup all objects created during this process.

Because each JavaScript mouse move event on the HTML `canvas` element will result in these five server-side operations, this is a very inefficient design. A stateless `mousemove` web service invocation will likely spend most of the service time in the dataset load, especially if the data is not cached in memory and/or has to be retrieved over a network. Overall, the more operations that the web service performs, the longer time the HTTP issuing browser thread waits for the response.

This truly stateless `mousemove` RESTful web service design has a lot of room for improvement. A more efficient design is for the `mousemove` service requires stateful RESTful web services that cache and operate on the stored state of the ubiquitous application viewport. The viewport in this context is a one-to-one mapping between each HTML `canvas` element and a viewport representation in the web service. The major difference is that this architecture requires the client to request operations to be performed on a specific volume object, meaning the web services need to hold a collection of volume objects in data structure. In this pattern, this `mousemove` web service decomposes into four web services:

1. `create`- create a viewport object
2. `load`- perform the loading of a specific data set
3. `setRenderStyle`- set the volume render style
4. `mousemove`- perform a mouse move by updating the render engine camera attributes

This RESTful web service design requires the return of a unique identifier when the object is created through the `create` web service. Each additional web service requires the unique identifier to be passed as a URI query parameter that identifies the viewport object that will be modified. This requires a series of web service calls before the call to

mousemove, and eliminates the need for datasetID and renderStyle query parameters when invoking the mousemove web service:

```
GET /mousemove?id={object ID &x={mouse x}&y={mouse y}
```

A "stateful" RESTful web service that stores the state of the viewport does not require the loading of a dataset, and the setting the render style for every mousemove, and only needs to change the camera and return the resultant rendered image. The overall web services URI query is more concise. It also breaks the web services into separate logical operations. This approach is not perfect, storing the state requires synchronization between the client and server; however, since this is a thin-Client approach, the client leverages the server for all viewport updates. The only major drawback with a "stateful" web service approach is need for resource management. Web service resource management will be explored in Section 5.2.4.

5.2.2 Java Servlets Versus JAX-RS RESTful Web Services

Having chosen a "stateful" RESTful web service client-server communication architecture, the last design decision surrounds the creation of Java RESTful web services. Section 3.2.2, introduced the Java Servlet and JAX-RS APIs for creating RESTful web services in Java. Overall, the Java Servlet approach to RESTful web services has many disadvantages. Java Servlets require a large amount of standard code to implement and dispatch HTTP requests appropriately from the doGet and doPost methods. Specifically, one of the greatest shortcomings of Java Servlets is the need to manually parse and cast URI query parameters from the URL. This URI query parameter decomposition and casting shortcoming is handled automatically with JAX-RS.

JAX-RS automatically performs the parsing and casting of the URL to extract the URI query parameters. This is accomplished through the use of Java annotations. These annotations specify the URI query parameters for each web service method, which improves Java code readability. Another advantage to JAX-RS is the declaration of web services in Java Interfaces, this provides modular separation between definition and implementation, and increases the readability and documentation of Java RESTful web services. Below is a code snippet of a JAX-RS `load` RESTful web service defined in a Java Interface. From the method annotations it is clear that this web service maps to a HTTP `GET` at the URL path `/load`, and requires three URI query parameters: `id`, `dataset`, and `mimeType` that map to Java parameters with the same name.

```
/**
 * Loads the data specified by the provided data ID into the viewport specified
 * by the provided web viewport ID
 * @param id The universally unique ID of the viewport in which the
 * data specified by {@code datasetID} will be loaded.
 * @param datasetID The ID of the data set to be loaded into the
 * viewport specified by {@code id}.
 * @param mimeType The MIME type of the image to be returned.
 * @return An HTTP 200 response containing graphical data in PNG format
 * if the load was successful; otherwise, a HTTP 500 response.
 */
@GET
@Path ( "/load" )
Response load( @QueryParam ( "id" ) String id,
    @QueryParam ( "datasetID" ) String datasetID,
    @QueryParam ( "mimeType" ) String mimeType );
```

Overall, JAX-RS is a very nice Java framework for mapping URI patterns and HTTP operations to Java methods based on annotating Java classes and methods (Burke, 2010). These annotations effectively inject URI parameters to Java methods. The JAX-RS framework aids in code readability, and facilitates RESTful web service development in Java. For these reasons JAX-RS is the application choice for RESTful web services.

5.2.3 JAX-RS RESTful Web Service Layer Architecture

The web service layer is fundamentally a service wrapper around an existing validated Java visualization toolkit. This service wrapper exposes the toolkit's API through "stateful" JAX-RS RESTful web services. Collectively these web services abstract and act as a proxy to the toolkit's features, and constitutes a virtual web 3D viewport object that is tightly coupled to a ubiquitous application viewport. Henceforth, a server-side web 3D viewport will be referred to as a web viewport. Fundamentally, this client-server architecture creates a one-to-one mapping between ubiquitous application viewports and web viewports where the web viewports expose the 3D visualization features of the leveraged validated toolkit as "stateful" JAX-RS RESTful web services.

Architecturally, when creating JAX-RS REST web services it is best to specify the JAX-RS RESTful annotations in a Java interface. A RESTful web service interface "is responsible for identifying how our service is to be exposed as a REST service" (IBM). A "REST interface is where we place our JAX-RS annotations that describe how our service is deployed and how the HTTP requests are mapped to our interface" (IBM). This REST interface ensures a clean separation between RESTful service definition and implementation. The interface not only ensures the existence of the web services, it clearly documents the capabilities of each web service through normal Java API documentation.

Following the practice of RESTful web service definition through a REST interface, the server-side toolkit defines three REST interfaces to define the full RESTful web service contract for 3D viewports. The core service contract is defined in the `Web3dViewportService` Java Interface. This REST service contract defines what it means to be a 3D web viewport resource through a collection of web services. This interface guarantees, and documents a set of HTTP accessible services that rely on two other RESTful

service contracts, see Figure 5.7. The Web3dViewportService REST interface extends WebViewportService. WebViewportService defines general viewport capabilities and RESTful services that are standard across all types of viewports, such as 2D and 3D. Finally, the WebResourceService REST interface defines the REST contract necessary for any web resource. This interface defines the necessary RESTful services that all persisting Java Object resources bound to a specific client-side application must follow. The key feature of a web resource is that it ensures proper memory management, and server stability. Without a resource management strategy the stability is in question. The web resource concept is further discussed in Section 5.2.4.

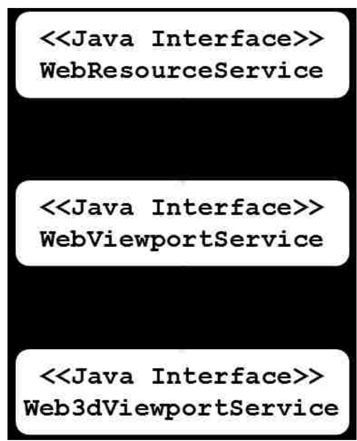

Figure 5.7: This figure describes the hierarchy of the web service layer REST interface through an interface class diagram.

The toolkit platform architecture is designed to be highly extendable in terms of future features. As the existing validated toolkit feature set is expanded with new features, or if they are replaced, this modular platform can easily adapt. This REST interface structure ensures platform extensibility, as changes are made the web service contract is updated by adding and implementing new methods in the REST interfaces.

With the concept of a 3D visualization viewport defined, and the full RESTful web service contract logically broken into three interfaces, the capabilities of the existing validated visualization toolkit can be extended to external programs through the web services. The implementation of the REST interfaces uses the proxy pattern to leverage the validation visualization toolkit's features. In this way, the true logic provided by the existing visualization toolkit is accessible to external programs via HTTP. By leveraging the JAX-RS framework and Java REST annotations, HTTP messages are routed to the appropriate service layer methods that proxy to the necessary methods in the existing toolkit. As an example, the load web service is defined in the `WebViewportService` REST interface. Through annotations, the interface indicates that the load web service responds to HTTP GET requests, and expects three query parameters.

```
@GET
@Path ( "/load" )
Response load( @QueryParam ( "id" )
String id, @QueryParam ( "datasetID" )
String datasetID, @QueryParam ( "mimeType" )
String mimeType );
```

The `load` method REST interface indicates that this REST method requires the identifier of the web viewport to use for loading the specified dataset, the identifier of the dataset to load, and the mime type of the return image. Because the return type of the image produced by this service is not static, this REST interface method does not identify a specific image mime type return using the Produces annotation. Lastly, the `Path` and `QueryParam` annotations together specify the URI to which this load web service maps. With a dataset identifier of `dataset1`, and a mime type of `png`, `dataset1` can be loaded on a viewport defined by `viewport1` given a HTTP GET message with the URI:

```
/load?id=viewport1&datasetID=dataset1&mimeType=png
```

By implementing the REST contract the JAX-RX framework invokes the implementing methods with the necessary parameters for every HTTP message that matches the defined Java REST annotations. Because any web architecture needs to be designed to support simultaneous users the architecture needs to define a resource management strategy. As a "stateful" REST architecture, persistent Java Objects used by specific clients are considered a web resource.

In the context of this web service layer, there exists a one-to-many relationship between a server-side web viewport, a web resource, and client-side ubiquitous application viewport. This relationship is defined as one-to-many because some 3D visualization applications may allow the sharing of a server-side viewport context between multiple application instances; for example, to support volume sharing between users for consultation purposes. To accomplish this, web viewports need to be managed by a management layer. This manager interfaces the REST web service implementation and the specific web viewport specified by the `id` query parameter in the HTTP URI. In this context, the resource manager manages a collection of web viewport objects. Each web viewport Object proxies the existing validated visualization toolkit. Figure 5.8 describes the communication flow from a HTTP request to the existing codebase via the JAX-RS RESTful service layer. The following server-side web service layer architecture discussion discusses the concept of resource management, and the use of web resources and shard resources in the server-side architecture.

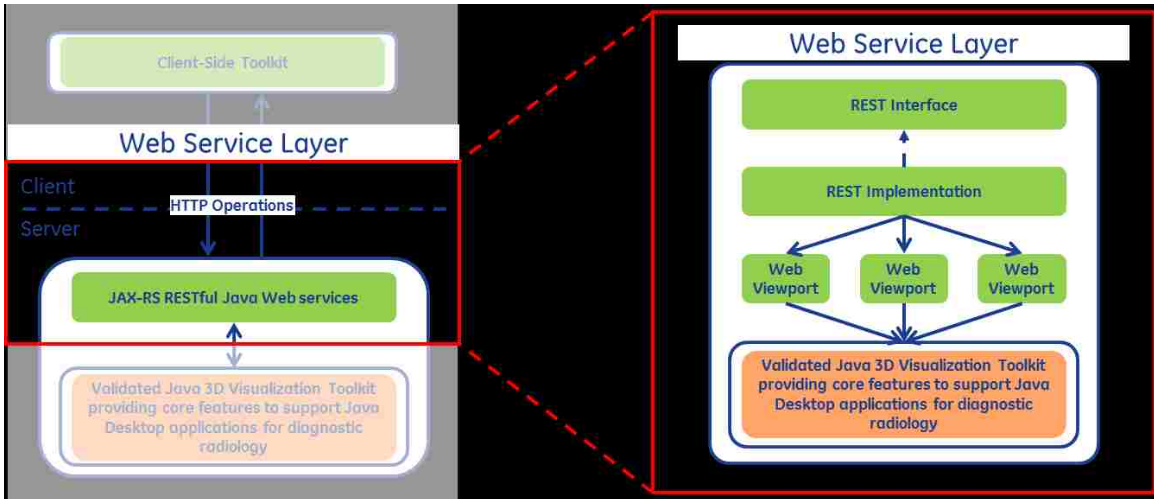

Figure 5.8: This figure highlights the server-side web service layer architecture.

5.2.4 Resource Management

Proper resource management is a vital design component of any application that allocates and stores resources. While "webifying" an existing implementation, it is not enough to simply create a one-to-one mapping between methods existing in an existing Java validated visualization toolkit and web services. Because the implementation is not designed for remote web application usage, the environment surrounding the implementation, expectations, and use cases are different. It is not sufficient for the web services to simply proxy the toolkit, a fundamental layer must be defined and added, a resource manager. In this discussion a resource is any Java object that is created and managed by an instance of a management class, a web viewport is a resource. This implies a one-to-many relationship between a resource manager and the resources it manages. Architecturally, a resource manager is tasked with managing a certain category of resources defined either by a Java Interface or a class.

For modularity and separation of concerns, it is advantageous to parameterize the resources that are managed by the resource manager. Parameterization allows instances of

a resource management class to constrain its managed resources to a particular resource type. Management through an interface guarantees the managed resource abide by a specific API contract.

In its true form, a resource manager is a wrapper around a map data structure. Construction of a resource manager instance requires the type of resource to be managed. Specifically, each manager contains a map that is a String to resource type, key value pair. This allows a resource manager user to specify a universally unique identifier that maps to a specific resource. Since the contents of the map needed to be guarded against inappropriate updates, the map is marked as private, and the resource manager provides a set of methods for access. The snippet below illustrates a basic resource manager with a parameterized resource type defined by `T`, where `T` is a class or interface.

```
public class ResourceManager<T> {
    private Map<String, T> resources;

    public ResourceManager( String managerName ) {
        resources = new ConcurrentHashMap<String, T>();
    }
    public void add( String key, T resource ) {
        resources.put( key, resource );
    }
    public int getNumberOfResources() {
        return resources.size();
    }
    public T get( String key ) {
        T resource = resources.get( key );
        return resource;
    }
    public T remove( String key ) {
        T resource = resources.remove( key );
        return resource;
    }
    public Map<String, T> getResources() {
        return new ConcurrentHashMap<String, T>( resources );
```

```
    }
}
```

Because this `ResourceManager` is parameterized, it requires the type of resource it manages during construction. The following snippet illustrates proper ResourceManager instantiation for a `ResourceManager` that manages `Viewport3D` resources

```
ResourceManager<Viewport3D> manager =
    new ResourceManager<Viewport3D>( "Viewport3D Manager");
```

The server-side web service layer requires two types of resource management: timed resource management, and shared resource management.

5.2.5 Managing Web Resources

In the web domain, client-side web applications often leverage a server that provides resources, like web viewports. Such resources are defined to be web resources. These web resources are managed by a resource manager, and are usually identified by some type of key. This key is used by the client-side application to identify the specific resource to use while performing the requested operation. Because there can be many client-side applications using server-side web resources, and only finite server hardware resources, there can be issues with memory management. As the number of applications and web resources increase, the overall memory usage on the sever increases. In this way it is too easy for memory usage to increase during the life of the server application. Without proper web resource management the server memory usage can continue to grow to the point where the server application runs out of memory. At this point there are two scenarios. Either the application can no longer provide web resources, or the Java Virtual Machine running the server application crashes and no longer communicates.

One of the largest draws on JVM memory is inappropriate web resource cleaning, specifically when a resource is no longer used. This situation occurs if a client application ends or permanently severs the connection to server. This lost association means the only reference to the web resource is the managing resource manager. Without proper notification to the server, these web resources will continue to exist in memory due to the resource manager containing a reference to the web resource object. This situation constitutes a memory leak. The JVM Garbage Collector cannot remove these web resources because they are referenced by the resource manager. Therefore, a resource manager in itself is not a complete solution to the problem of managing web resources.

To mitigate memory leaks, when applications are done using web resources they must notify the server. The resource manager can then remove the web resource from its collection. However, it is too easy for memory leaks to occur when the server still thinks a resource is being used. Forgetting, or purposefully not notifying the server produces a memory leak that cannot be mitigated until the server application is restarted and the OS reclaims this memory.

For this reason, the honor system is not a good pattern for web resource management; it relies on application developers following the rules. It is prudent for the server-side web service layer to be smarter. One approach is a special type of resource manager that defines a maximum time each web resource can remain unused before it is deemed “stale”, meaning it has been marked for removal. Such a resource manager is henceforth known as a web resource manager.

The server-side toolkit architecture utilizes a web resource manager for server-side web viewport management. What makes the web resource manager unique is that it only stores web resources. A web resource differs from a traditional resource in that it has a

property that stores the last time it was used. To work properly, each time a web resource is used in a web service the time stamp is updated with the current time. As with a normal resource manager, a web resource should only be managed by a single web resource manager. Because web resources are used by web services, and each web resource is held by a web resource manager, the web resource manager logically exists between web service and the web resource, Figure 5.9.

Figure 5.9: A web resource manager manages a collection of web resources. The web resource manager logically exists between the web services and a web resource.

Storing the last time a web resource has been used is not sufficient for minimizing memory leaks; the second step involves a proactive approach for removing "stale" web resources. The web resource manager requires a cleaner task that periodically scans all the web resources managed by the web resource manager and proactively removes any web resource that has been deemed "stale". A web resource is considered "stale" when the last time since the resource has been used exceeds a defined duration. Specifically, a specific web resource instance, `webResource`, is considered "stale" when:

```
System.currentTimeMillis() - webResource.getLastTimeUsed() > RESOURCETIMEOUT
```

When web resources are marked as "stale" they are removed from the web resource manager and their memory is freed. In this design, even if the client-side application does not tell the server about the web resource no longer being used it will eventually be freed. A web resource manager does not guarantee the JVM will not crash from a lack of available memory, it just helps. The web resource timeout needs to be chosen that maximizes server stability and minimizes removal of resources that are still in use.

The platform contains a many-to-one association between client-side application viewports and a web viewport, as shown in Figure 5.9. In this context, a client-side viewport is an application viewport, and a web viewport is a server-side viewport resource. This association between application viewports and viewport web resources is bound by a universally unique identifier. Since most RESTful web service operate on a specific viewport web resource, the service requires the universally unique identifier of the viewport web resource bound to the application viewport as a URI parameter. This allows web services to get a handle on the specific viewport web resource through the web resource manager.

All viewport web resources in the system are managed by a web resource manager. To ensure 3D viewports are not marked as “stale” and removed prematurely, each service call on a specific viewport web resource needs to update the resource’s last time used. Ideally, the process of updating a web resource’s last time used should happen automatically. The web resource manager is the perfect location to automatically update the web resource’s last time used, since all services that operate on a specific web resource must attain the web resource from the web resource manager. By automatically managing the update of a web resource’s last time used, and blocking restricting access to resources through the get method, the likelihood of a web resource’s last time used not getting updated is eliminated.

5.2.6 Managing Shared Resources

The second type of resource manager is one that manages shared resources. In a shared resource manager, the same resource is used by multiple objects. The manager keeps a count of the number of users for each shared resource, and continues to manage the resource while its count is greater than zero. A shared resource is typically a heavyweight resource that takes up a lot of hardware resources, and maybe a resource that

is considered slow to instantiate. In some cases, resources are designed to persist; these types of resources should not be instantiated and destroyed for every web service request. One solution is to create a pool of shared resources that wait until they are needed. When a web service request requires such a resource the web service grabs an available resource, performs the requested operation, and then returns the resource to the pool for reuse. If a web service requires the use of a shared resource but the pool is empty, the service must block until a resource is returned to the pool. More complex shared resource managers can extend this pattern, and may dynamically instantiate more resources depending on the load.

In the web services layer the server manages volumes instances with a shard resource manager. Each volume is a sharable resource that is used by at least one viewport web resource. When an application calls the `load` RESTful web service it provides a unique dataset ID that identifies the volume to be loaded. The loading of the dataset in a volume may take seconds, especially if the data describing this volume is stored remotely. Once a volume is created, it is stored in the resource manager, and loaded into the specified viewport web resource. Subsequent calls to load the same dataset are faster because the volume defined by the dataset ID already exists in the shared resource manager.

To adequately manage memory, these volumes must be removed from the shared resource manager when all viewport web resources referencing a volume cease to exist, or if another volume is loaded in the viewport. This requires a counter for each volume that increases as more viewport web resources load the volume, and decrease as web resources goes away or a change to the underlying volume is made. In this way, only when the reference count of a volume reaches zero is the volume removed from the containing data structure and memory. Specifically, this shared resource manager contains a mapping of volume dataset identifier to `VolumeRecord`. A `VolumeRecord` is a wrapper around a

Volume object that stores the number of resources using a specific Volume. The below VolumeManager singleton class snippet illustrates the concept of resource manager that manages shared Volume objects. All updates to the shared resource map are controlled through addToCahce and removeFromCache.

```
public class VolumeManager {
    private static class VolumeRecord {
        private Volume volume = null;
        private int count = 0;

        public VolumeRecord( Volume volume, int count ) {
            this.volume = volume;
            this.count = count;
        }

        public Volume getVolume() {
            return volume;
        }

        public void setCount( int count ) {
            this.count = count;
        }

        public int getCount() {
            return count;
        }
    }

    /**
     * The singleton instance of {@code VolumeManager}.
     */
    private static VolumeManager volumeManager = null;

    /**
     * Cache that stores a {@link VolumeRecord} to a {@link String}
     */
    private static final Map<String, VolumeRecord> cache =
        new HashMap<String, VolumeManager.VolumeRecord>();

    private VolumeManager() {}
```

```
/**
 * Returns a singleton instance of {@code VolumeManager}. This method is
 * thread safe.
 *
 * @return A singleton instance of {@code VolumeManager}.
 */
public synchronized static VolumeManager getVolumeManager() {
    if (volumeManager == null ) {
        volumeManager = new VolumeManager();
    }
    return volumeManager;
}

/**
 * Adds a new Volume to the cache. Increments the number of users.
 *
 * @param id The unique identifier for the Volume to add to the cache.
 * @param volume The Volume object to add to the cache.
 */
public void addToCache( String id, Volume volume ) {
    if ( cache.containsKey( id ) ) {
        VolumeRecord vr = cache.get( id );
        int newVolumeCount = vr.getCount() + 1;
        vr.setCount( newVolumeCount );
        cache.put( id, vr );
    } else {
        if ( null != volume ) {
            VolumeRecord vr = new VolumeRecord( volume, 1 );
            cache.put( id, vr );
        } else {
        }
    }
}

/**
 * Decrements the number of objects using the Volume. When the number is
 * less than or equal to 0 the Volume is removed from the cache.
 *
 * @param id The unique identifier for the Volume to remove for the cache.
 */
public void removeFromCache( String id ) {
```

```
        synchronized ( this ) {
            if ( cache.containsKey( id ) ) {
                VolumeRecord vr = cache.get( id );
                int newVolumeCount = vr.getCount() - 1;
                if ( newVolumeCount <= 0 ) {
                    cache.remove( id );
                } else {
                    vr.setCount( newVolumeCount );
                    cache.put( id, vr );
                }
            }
        }
    }
}
```

The overall result of the server-side web service layer architecture is shown in Figure 5.10. The figure identifies the web service layer as JAX-RS RESTful Java web services that communicating via HTTP. This toolkit exposes the existing validated Java visualization toolkit as web services that can be utilized by ubiquitous applications.

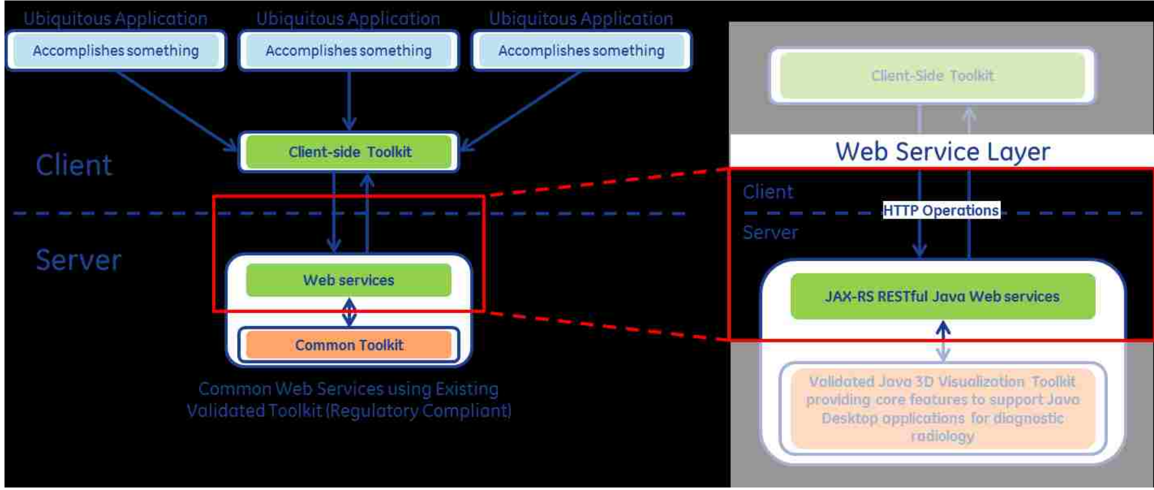

Figure 5.10: This figure highlights the web service layer wrapping of the existing validated visualization toolkit by JAX-RS RESTful web services for the purposes of "webifying" the existing validated toolkit.

5.3 Ubiquitous Toolkit

The exposed RESTful web services provided by the web service layer are sufficient for developing rich ubiquitous 3D visualization web applications; however, using JavaScript to consume them is not trivial. The common web service invocation pattern involves calling the web service via an HTTP operation, and then processing the response. In most cases, the response needs to update the DOM of the application HTML. Instead of each application having to develop the base constructs for interaction with the RESTful web services, a client-side JavaScript ubiquitous application toolkit provides a viewport construct. This viewport construct abstracts the client-server RESTful communication through a rich API for inclusion in ubiquitous applications. Henceforth, this toolkit will be referred to as a ubiquitous toolkit.

The ubiquitous toolkit simplifies ubiquitous application development. It exposes an API that surrounds a HTML 5 `canvas` object. Specifically, this toolkit takes ownership of the `canvas` object for the creation of an application viewport. The ubiquitous toolkit viewport handles all interaction and updates to the `canvas` object. Henceforth, the client-side toolkit wrapping of a HTML 5 `canvas` object constitutes a ubiquitous viewport. This ubiquitous viewport simplifies the creation of rich ubiquitous applications, and ensures standard ubiquitous viewport behavior and look-and-feel.

Since the application toolkit abstracts the RESTful web services, and because the web services are "stateful", each ubiquitous viewport is tightly bound to a server-side web viewport via a universally unique identifier. The specifics of this association are hidden from the ubiquitous application developer by the ubiquitous toolkit. However, the ubiquitous toolkit does provide functionality to a get and set the web viewport identifier for certain ubiquitous application workflows. This, among other things, allows the sharing of a viewport web resource context across ubiquitous viewports in the same or separate applications.

Despite the architecture supporting any software language capable of consuming RESTful web services, the idea is to create ubiquitous applications written in HTML and JavaScript. There are two goals of the JavaScript ubiquitous viewport: hide the client-server communication, and expose a ubiquitous viewport construct that simplifies the development of JavaScript. Essentially the ubiquitous viewport is a convenience API that removes some of the complexities inherent with JavaScript development. Collectively, the features of the ubiquitous viewport ensure standard behavior, standard ubiquitous viewport and look and feel, and obfuscate any changes to the web service layer.

In general, a toolkit should not force certain behaviors, and should carefully make decisions related to the usage of third party libraries. The JavaScript ubiquitous viewport exposes a viewport object, called `Ubiquitous3dViewport` that is continually referred to as ubiquitous viewport. This object is designed to be leveraged by ubiquitous applications, and provides core capabilities that each application would need to implement manually. The ubiquitous viewport does not rely on third party JavaScript libraries such as jQuery (The jQuery Foundation01) or Knockout (Knockout) as JavaScript convenience libraries. Although these libraries provide core capabilities to standard JavaScript that simplify JavaScript development, using only "plain vanilla" JavaScript controls the overall footprint of the toolkit, and eliminates the possibility of third party library version conflict. These ubiquitous toolkit design choice allows the ubiquitous application developer the flexibility to utilize these third party application libraries for the web application.

For 3D volume visualization and interaction the JavaScript ubiquitous viewport relies on the HTML 5 `canvas` element, specifically its 2D `context` JavaScript object (W3Schools02). The 2D `context` object supports drawing images and graphics, and text drawing useful for drawing medical image annotation on the HTML 5 `canvas` element. This

is standard across HTML 5 compliant Internet browsers such as Microsoft Internet Explorer 9+, Mozilla Firefox, Opera, Google Chrome, and Apple Safari (W3Schools02). Being a toolkit, the intent is to make the architectural design of the ubiquitous toolkit simple and easily to use, allowing for rapid ubiquitous application development.

A simple application requires a HTML page containing only a HTML 5 `canvas` element, a reference to the ubiquitous toolkit JavaScript file, and the creation of a JavaScript `Ubiquitous3dViewport` object. Upon construction of a ubiquitous viewport, the wrapped `Canvas` element has registered mouse and touch events, used for volume manipulation. In addition, during construction the application toolkit initializes a corresponding server-side web viewport object that will be leveraged by the ubiquitous viewport throughout its life. The ubiquitous viewport object exposes a local JavaScript API that leverages the web viewport through the server-side web services when necessary.

The ubiquitous toolkit architecture is intended to expedite rich ubiquitous application development by obfuscating interaction with the server-side RESTful web services by providing convenience APIs that performs the necessary HTTP client-server communication. The below HTML and JavaScript code segment shows how to instantiate the application toolkit `Ubiquitous3dViewport` ubiquitous viewport object with a HTML `canvas` object. The resultant `viewport` JavaScript variable has is a `Ubiquitous3dViewport` ubiquitous viewport object with a height and width of 512 pixels.

```
<canvas id="viewportCanvas" width="512" height="512"></canvas>
```

```
var canvas = document.getElementById('viewportCanvas');
var viewport = Ubiquitous3dViewport(canvas);
```

To convey the advantage of the ubiquitous viewport, following is a discussion using the RESTful web services by hand and through the ubiquitous viewport.

Without using the `Ubiquitous3dViewport` object, loading a volume onto the viewport requires invoking the `load` web service, and painting the returned image on the HTML `canvas`.

```
var image = new Image();
image.src =
http://localhost:8181/load?id=vp.id&datasetID=C:\path\to\dataset&mimeType=png;
var context = canvas.getContext('2d');
context.drawImage(image, 0, 0, canvas.width, canvas.height);
```

Although the code footprint for loading a dataset onto a HTML `canvas` using the web services is small, with the ubiquitous viewport the same operation is one line. The ubiquitous viewport `load` function takes care of performing the HTTP GET and handles painting the resultant image from the load web service to the HTML 5 canvas's context.

```
viewport.load("C:\path\to\dataset", "png");
```

While consuming web services that produce an image requires only setting the JavaScript `image` object source to the RESTful web service URL, consuming RESTful web services that return any other type of data requires the use of the JavaScript `XMLHttpRequest` object. Additionally, synchronous and asynchronous consumption of a web service that returns a non-image use the `XMLHttpRequest` object differently. The following snippets represent consumption of the `getViewHeight` web service synchronously and asynchronously.

```
var xmlhttp = new XMLHttpRequest();
// Perform synchronous web service consumption.
var performSynch = true;
// The web service URL including query parameters. Assume the server-side
// viewport web resource id is stored in id
var query = http://localhost:8181/getViewHeight?id=" + id;

xmlhttp.open("GET", query, performSynch);
```

```
var response = xmlhttp.onreadystatechange = responseHandler;
xmlhttp.send(null);

// Because the HTTP GET occurred synchronously the execution thread blocks
// until we have a response.
alert(response);
```

```
var xmlhttp = new XMLHttpRequest();
// Perform synchronous web service consumption.
var performSynch = false;
// The web service URL including query parameters. Assume the server-side
// viewport web resource id is stored in id
var query = http://localhost:8181/getViewHeight?id=” + id;

xmlhttp.open("GET", query, performSynch);

var response = null;
function responseHandler() {
	if (xmlhttp.readyState == 4) {
		if (xmlhttp.status == 200) {
			response = xmlhttp.responseText;
			// we performed consumption asynchronously
			alert(response);
		}
		// Request ERROR
		else {
			// handle the error
		}
	}
}
```

The JavaScript ubiquitous viewport uses both the synchronous and asynchronous code segments together, wrapped in a web utility function to support blocking and non-blocking HTTP web service consumption. The result is a one line call to the `Ubiquitous3dViewport` `getViewHeight` function.

```
var xmlState = viewport.getViewHeight();
```

The JavaScript ubiquitous viewport's functions all except a callback function as the last argument. The presence of a callback function results in asynchronous communication with the necessary server-side web service, while the lack of a callback function results in synchronous communication. For example, the following two JavaScript code segments exercise the `getViewHeight` function of the ubiquitous viewport with and without a callback function. The lack of a callback function forces synchronous communication, causing the code execution to block until the HTTP GET operation returns. Contrary, the presence of a callback function forces asynchronous communication, causing the code execution to continue. For asynchronous communication, when the HTTP GET operation returns the callback function is called. The below segments invoke the `getViewHeight` function, and display the response view height to the Internet browser console. Figure 5.11 shows the synchronous and asynchronous execution of the `getViewHeight` ubiquitous viewport function through the Google Chrome developer console.

```
console.log("Synchronous HTTP GET:\n\n" + viewport.getViewHeight());
```

```
viewport.getViewHeight(function(viewHeight) {
	console.log("Asynchronous HTTP GET:\n\n" + viewHeight);
});
```

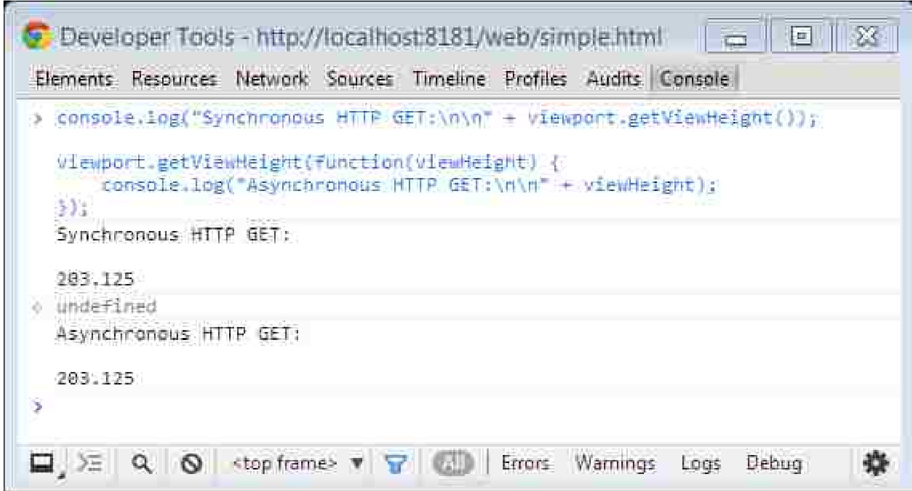

Figure 5.11: Screenshot of the Google Chrome developer console invoking the getViewHeight web service through the ubiquitous viewport's getViewHeight function. The function is called with and without a callback function to invoke the web service synchronously and asynchronously.

All ubiquitous viewport functions that proxy a web service that does not return an image support execution synchronously or asynchronously through the lack or presence of a callback function. For each of these functions, providing a callback function means the function will not return with the response. Rather the response will be passed as the argument to the callback function. Being a toolkit, the ubiquitous viewport does not restrict asynchronous or synchronous Hypertext Transport Protocol client-server communication between the ubiquitous toolkit and the supporting RESTful web services. Although the preferred and standard practice is to perform HTTP operations asynchronous, so that the HTML web application User Interface is responsive rather than blocked, the toolkit supports blocking and non-blocking communication. This flexibility is provided to allow the toolkit to provide its functionality to a larger collection of applications.

The common format for returning non-primitive and non-image data from web services is the JavaScript Object Notation. JavaScript provides a simple operation for converting JSON into its representative JavaScript object using the `JSON.parse()`

function. Each ubiquitous viewport function uses the `JSON.parse()` core JavaScript function to appropriately convert the response into its representative JavaScript object before returning, if synchronous, or calling the callback function, if asynchronous. This simplifies the use of the ubiquitous toolkit's API, and does not require a ubiquitous application developer to parse the HTTP response manually for each function call. To simplify the ubiquitous toolkit architecture, all web service calls are performed using a common web utilities library leveraged by each ubiquitous viewport function communicating with a web service. Specifically, the web utilities library provides a singleton object that exposes functions for retrieving and posting data of different types. These functions include the retrieval of text, JSON, and images, and posting of text, as seen in Figure 5.12. This singleton web utility object ensures consistent client-server communication and allows the communication protocol to be updated in one location, instead of requiring an update each ubiquitous viewport function.

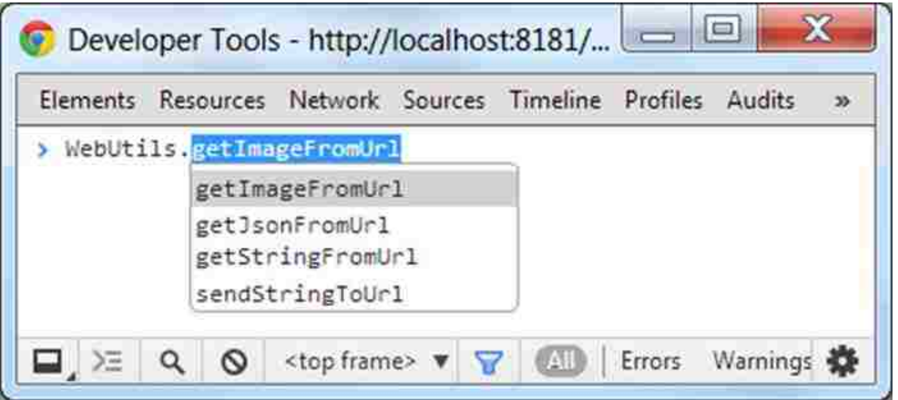

Figure 5.12: This figure shows the available communication utility function provided by the WebUtils singleton JavaScript library. This library handles asynchronous and synchronous HTTP communication with a web service.

The next two sections further discuss the ubiquitous toolkit. The next section discusses the ubiquitous viewport mouse interaction architecture, and the later discusses the error handling architecture of the RESTful web services and the ubiquitous toolkit.

5.3.1 Mouse Interaction Architecture

Mouse interactions over HTTP and ubiquitous toolkit allow the user to perform a mouse or touch drag, or use the mouse scroll wheel to interact with the volume displayed in the ubiquitous viewport. In this discussion a mouse drag is defined as a sequence of three steps:

1. A left mouse click on the HTML `Canvas` object, application viewport,
2. A series of mouse moves with the left mouse button remaining pressed,
3. Left mouse button release.

Users interacting with the volume in the ubiquitous viewport, specifically the `Canvas` object, may see this interaction as a complex series of operations; however, the pattern for volume interaction is handled by the RESTful web services.

When the ubiquitous viewport object in a ubiquitous application takes ownership of the HTML 5 `Canvas` object it installs on it mouse and touch events, specifically: `mousedown`/`touchstart`, `mousemove`/`touchmove`, and `mouseup`/`touchend`. Each of these events accepts a JavaScript `Event` object that contains specific information about the event. For mouse events, this object includes the screen coordinates (`x`, `y`) of the mouse event, and the index of mouse button pressed, assuming a mouse button was pressed. For touch events the `Event` object contains screen coordinates for each touch position, allowing for multi-touch gestures.

Using the `Event` object the `mousedown`/`touchdown` event starts the mouse drag process. The `mousemove`/`touchmove` and `mouseup`/`touchend` events are where the volume interactions occur. Each of these HTML `Canvas` object mouse events sends the (`x`, `y`) coordinate of the mouse event to their respective RESTful web service. For simplicity, touch

events leverage the mouse event web services. Unlike the web service for the mousedown/touchdown event that returns the (x, y) coordinates as an array, the web services supporting mousemove/touchmove and mouseup/touchup RESTful web services return images encoded as either image/png or image/jpeg. To complete the viewport mouse or touch interaction, the application toolkit paints the return image on the HTML Canvas object.

To improve user application viewport interaction, the ubiquitous toolkit supports an interaction timeout that is used to display a high quality image during stagnant mouse movement during ubiquitous viewport interaction. By default, during mouse interaction, if there are no mousemove/touchmove or mouseup/touchend HTML canvas events fired after 200 milliseconds, the ubiquitous toolkit requests a full sized PNG image from the server. This behavior does not require a mouseup/touchend HTML canvas event to display a high quality image on the application viewport.

Together the client and server work to efficiently support volume interaction through the ubiquitous viewport. Although the client-server communication is abundant during mouse interaction, the platform minimizes network bandwidth by minimizing the size of the images returned. The platform supports the return of images that are dimensionally smaller than the viewport, and are compressed using JPEG or PNG image compression algorithms.

5.3.2 Toolkit Error Handling Architecture

When designing a toolkit, it is necessary to notify the application of any errors. In the web domain this is a more complex problem than in the desktop application toolkit space. In the web domain a successful HTTP operation is denoted by a 200 response code that means the operation performed without error. RESTful web service consuming may result in any

number of errors defined by the HTTP response code standards; however for web service layer errors are denoted by an error code of `500 Internal Server Error`.

Simply returning a `500` response code to the ubiquitous toolkit is not very specific so the server-side web services incorporate unique error codes that are returned as the payload of any `500` response packet. The specific error codes have a publicly documented definition allowing the ubiquitous application to easily decipher and determine what error occurred. In this way the toolkit has no responsibility other than telling the application that an error has occurred. In ubiquitous application architecture it is up to the application to handle the error appropriately based on the `onerrorCallback` function provided to every ubiquitous toolkit API.

One added complication arises in the ability to extract the error payload of HTTP requests that are made expecting an image to be returned. Because these RESTful web services are invoked by setting the `image.src` equal to a URL there is no easy way to extract the error payload to determine the cause of the error. To mitigate this issue the platform contains a `getLastError` RESTful web service that will return the last error code to the client. As seen below in the JavaScript ubiquitous toolkit code snippet, a function is bound to the `image` object's `onerror` property. If an error occurs while requesting an image from a RESTful web service, the error code will be automatically requested and returned through the supplied `onerrorCallback` function. The `onerrorCallback` function is provided by the application, and is responsible for properly handling the error.

```
image.onerror = function() {
// if errorcallback is provided get error status code from the server
if (onerrorCallback && null != onerrorCallback) {
    // using XMLHttpRequest object get last error code by calling
    // getLastError web service
    onerrorCallback(errorCode);
```

```
}
image.src = query;
```

The overall result of the ubiquitous toolkit is an exposed ubiquitous viewport that, like a widget, is intended to be integrated into the graphical user interface of a ubiquitous application. Not only does this toolkit expose a ubiquitous viewport but it provides an error handling architecture as well as providing a full API for interacting with the ubiquitous viewport. As seen in Figure 5.13, this ubiquitous toolkit exposes a ubiquitous viewport that provides a rich JavaScript API leveraging the RESTful server-side web services for the purpose of supporting ubiquitous applications. This toolkit expedites and simplifies ubiquitous application development and also minimizes the scope of application validation because it leverages an existing validated visualization toolkit..

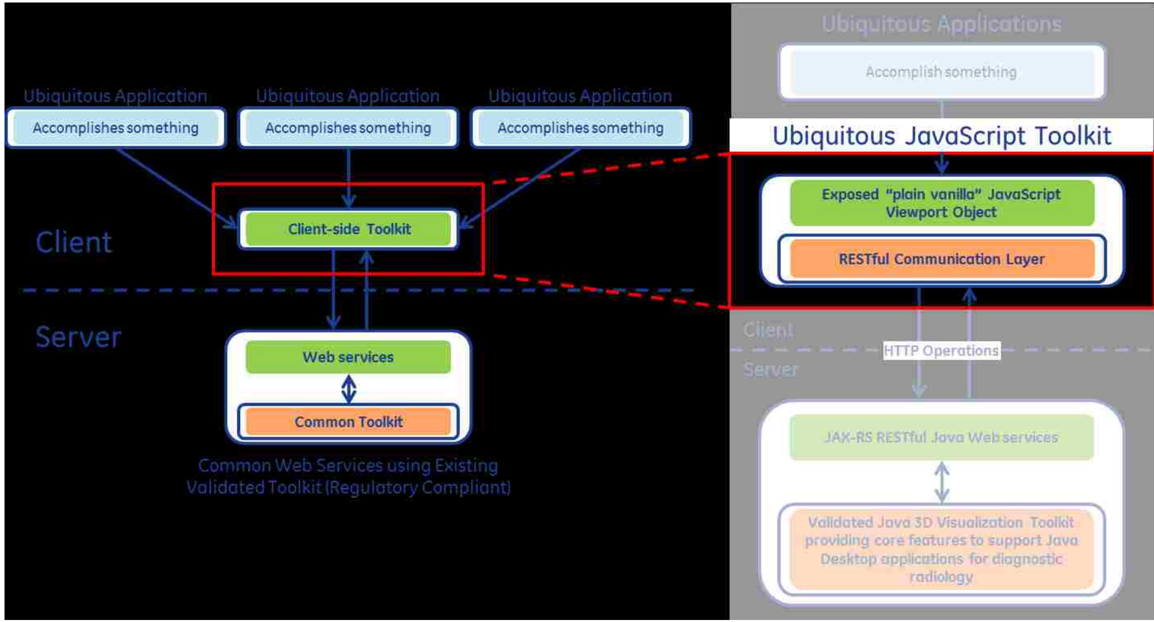

Figure 5.13: This figure highlights the ubiquitous JavaScript toolkit that exposes a "plain vanilla" JavaScript ubiquitous viewport object and provides a web utilities library for client-server communication.

5.4 Ubiquitous Application

The motivation of this thesis is the development of ubiquitous applications that leverage an existing validated Java visualization toolkit. Ubiquitous applications are thin-Client and Zero-Footprint web applications that are lightweight clients leveraging the ubiquitous toolkit, and indirectly a fully featured validated visualization toolkit for 3D volume rendering and manipulation.

Prior to today's web application technologies it was possible to build a ubiquitous application using markups, scripts, and styles. Given a similar set of web services leveraging a validated visualization toolkit a ubiquitous viewport like construct could be constructed using the HTML `image` element. With this foundation, ubiquitous applications can be built that simulate volume interaction by registering mouse events on the `image` element and leveraging the web services for changing the image displayed in the application simply by updating the source of the `image` element to the URL of the corresponding web service. Although this approach is capable of volume visualization and manipulation, it is not easy to display medical annotation and other graphic overlays that a true ubiquitous viewport needs. Due to this limitation, this approach is not ideal for ubiquitous applications, and is not appropriate for a ubiquitous toolkit.

With the arrival of version 5 of the HyperText Markup Language standard, and supporting updates to the JavaScript standard, it is now possible to use the HTML 5 `canvas` element and JavaScript technologies to build rich production level ubiquitous applications suitable for diagnostic image review. Unlike the HTML `image` element that is built to display images, the `canvas` element is an object that defines an area in the HTML that supports drawing images and graphics. Not only does the `canvas` element support drawing

graphics, it is capable of supporting interactive graphics that can interact with a user. This HTML element is the perfect construct for a ubiquitous viewport.

Using the HTML `canvas` element a similar approach to using the `image` element can be used to create a ubiquitous viewport. Unlike the `image` element that will update the displayed image whenever the source attribute of the element is changed, drawing an image on the `canvas` element is a little more complex. Specifically, drawing an image on a `canvas` element requires a JavaScript image object to draw on the2D `context` of the `canvas`. The code snippet below shows the necessary JavaScript code for drawing an image returned from a server HTTP request on a HTML `canvas` with an identifier attribute of "`viewport`".

```
var canvas = document.getElementById('viewport');
var image = new Image();
image.src = 'http://localhost/image.png';
var context = canvas.getContext('2d');
context.drawImage(image, 0, 0, canvas.width, canvas.height);
```

The 2D `context` of an HTML `canvas` element contains a rich API that supports the display of a JavaScript image, and contains a set of complex display of graphics functions that can be used to draw graphics on a HTML `canvas` element. Therefore, the foundation of ubiquitous applications is the HTML 5 `canvas` element.

Just as it is advantageous to leverage an existing validated visualization toolkit for Java desktop applications for diagnostic radiology, it is advantageous to leverage a client-side ubiquitous toolkit for ubiquitous applications. Such a toolkit expedites and simplifies ubiquitous application development and also minimizes the scope of application validation. As this thesis has defined such a ubiquitous toolkit that leverages an existing Java validated

visualization toolkit, this discussion will commence with the architecture discussion of ubiquitous applications that leverage this ubiquitous toolkit.

Leveraging the ubiquitous toolkit requires creating an association between HTML 5 canvas objects and the ubiquitous toolkit. As seen in Figure 5.14 the association between a HTML 5 canvas element and the ubiquitous toolkit makes it easy to add ubiquitous viewports to a ubiquitous application.

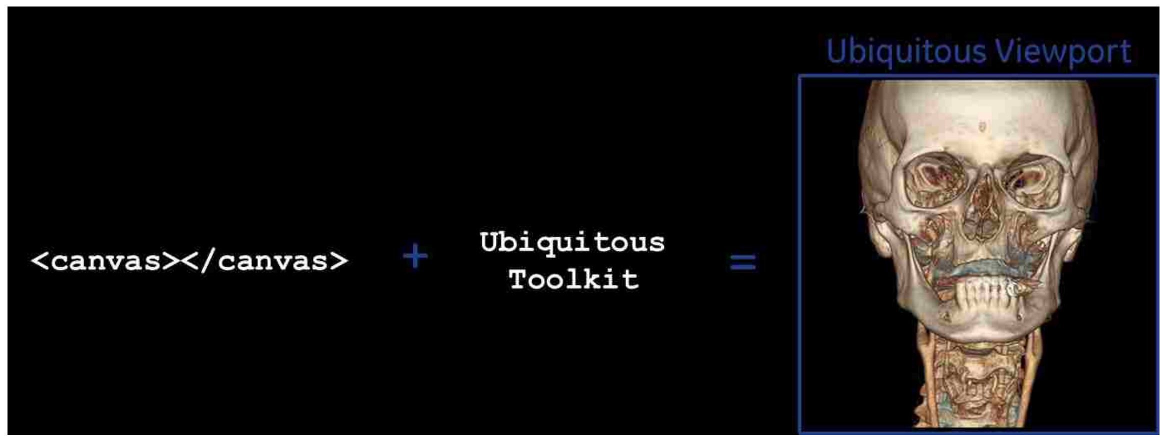

Figure 5.14: This figure shows the association between a HTML 5 canvas element and the ubiquitous toolkit yields a ubiquitous viewport that can be placed in a ubiquitous application.

The most basic of ubiquitous applications is written with HTML 5, JavaScript, and CSS, and contains several ubiquitous viewports. As JavaScript development is foundational, not many applications are built using “plain vanilla” JavaScript. Rather, to simplify application development, many choose to use a JavaScript library such as jQuery (The jQuery Foundation01). In addition, rich ubiquitous applications may leverage application frameworks such as AngularJS (Google) or Durandal (DURANDAL) for the creation of single page ubiquitous applications.

Creating ubiquitous applications is the end goal of this thesis. These applications are built leveraging the ubiquitous toolkit that exposes ubiquitous viewports that can be easily

incorporated into applications. Because these applications leverage the ubiquitous toolkit they therefore leverage the existing Java validated visualization toolkit. This means leveraging the ubiquitous toolkit not only simplifies application development, but it minimizes the scope of application validation for diagnostic radiology. The ubiquitous application layer architecture is shown in Figure 5.15. The next section goes builds two ubiquitous applications leveraging the ubiquitous toolkit.

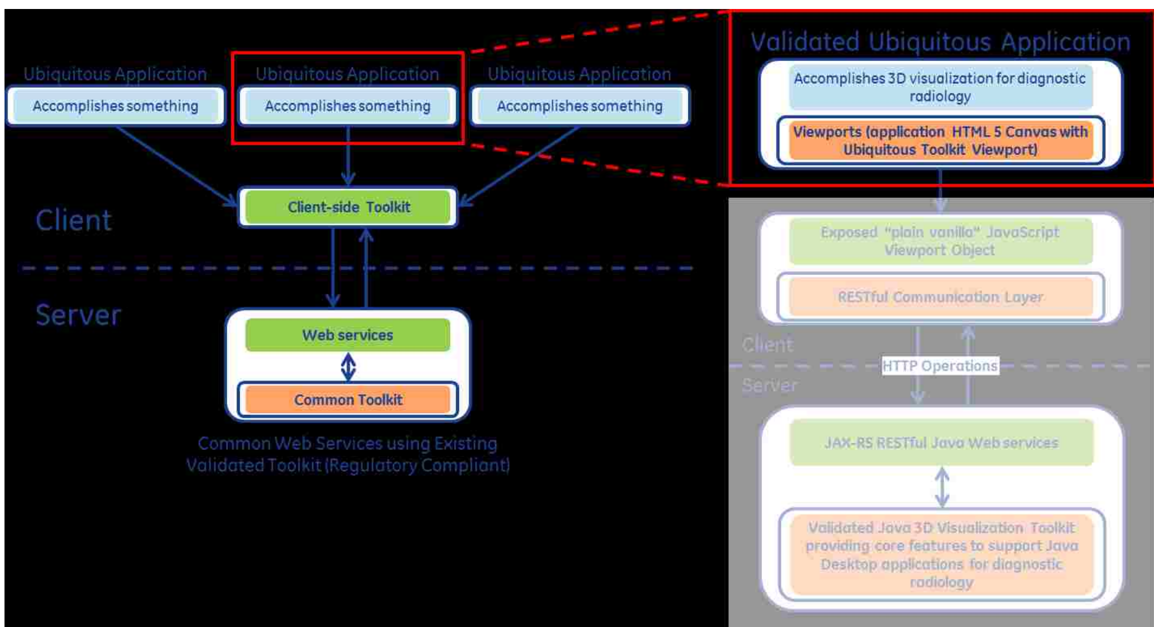

Figure 5.15: This figure highlights the ubiquitous applications that leverage the ubiquitous toolkit for rapid application development. These ubiquitous applications incorporate ubiquitous viewport object(s) that are provided by the ubiquitous toolkit. By leveraging the ubiquitous toolkit ubiquitous applications require less validation than applications that do not leverage such a toolkit.

CHAPTER 6: EVALUATION

This chapter evaluates the platform as a whole, from ubiquitous application development to platform performance. As a proof of characteristic of this platform, the platform, specifically the ubiquitous toolkit, is used to develop two sample ubiquitous applications that together showcase the platform. The two ubiquitous applications evaluate the overall platform fundamentals to the foundations of rich 3D visualization application.

6.1 Proof of Characteristics

The ubiquitous application platform for developing ubiquitous applications requires an execution environment that exposes the RESTful web services. This platform provides ubiquitous applications the necessary web services and ubiquitous toolkit constructs for building rich ubiquitous applications for diagnostic radiology.

For proof of characteristics of this platform, the RESTful web services will be packaged as OSGi bundles. OSGi bundles are Open Services Gateway initiative Java ARchive, JAR, files. What makes OSGi bundles special is a manifest file that specifies the packages the bundle requires, and the packages it exposes for other bundles. The special manifest file allows the OSGi container, the Java execution environment, to manage all the dependencies. Providing the necessary Java execution environment for the RESTful web services is Apache karaf (The Apache Software Foundation).

Once the web services are active, applications can be developed. The most basic of ubiquitous applications is simply a 3D viewport. The basic HTML skeleton code below serves as the foundation for any application.

```
<!DOCTYPE HTML>
<html>
    <head>
        <!-- Include the ubiquitous application toolkit -->
        <script src=" UbiquitousAppToolkit.js" type="text/javascript"></script>
            <!-- Use the application viewport -->
            <script type="text/javascript">
                function initialize() {
                    // Bind the application viewport to the canvas
                    var canvas = document.getElementById('viewportCanvas');
                    var viewport = new Ubiquitous3dViewport(canvas);
                    // Load a dataset
                    var datasetId = "C:/path/to/dataset";
                    viewport.loadVolume(datasetId);
                }
            </script>
        </head>
    <body onload="initialize()">
        <canvas id="viewportCanvas" width="512" height="512"></canvas>
    </body>
</html>
```

This web application contains only one active ubiquitous viewport, see Figure 6.1. All ubiquitous viewports by default register mouse events. By default the active mouse interaction is the trackball tool which rotates the volume in three dimensional space.

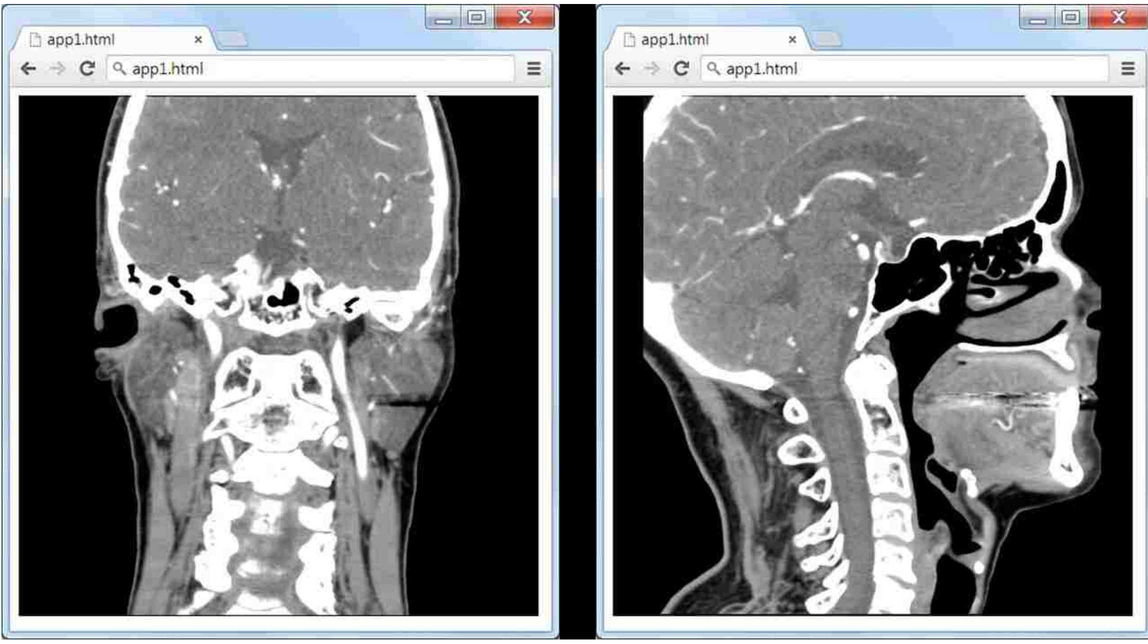

Figure 6.1: A simple, single application viewport thin-Client and Zero-Footprint 3D visualization web application. The figure on the left shows the application viewport as the volume is loaded. The figure on the right shows the same application viewport after mouse interaction.

The last example of using the ubiquitous toolkit and platform was the most basic web application. Following will be a more thorough examination of the platform capabilities for building a typical four-port application. A four-port application consists of four viewports in a 2x2 viewport layout with the following viewport configuration:

- Upper-left: anterior volume rendering
- Upper-right: anterior reformat
- Lower-left: right reformat
- Lower-right: superior reformat

The described four-port application will only require updates to the body and script tags of the previous application, and will display a bounding box and 3D reference cursor graphics on each viewport. First, the body of the HTML application must be updated to include four canvases.

```
<canvas id="viewportCanvas1" width="256" height="256"></canvas>
<canvas id="viewportCanvas2" width="256" height="256"></canvas>
<br>
<canvas id="viewportCanvas3" width="256" height="256"></canvas>
<canvas id="viewportCanvas4" width="256" height="256"></canvas>
```

Next, an array of JavaScript objects describes each of the four ubiquitous viewports is used to simplify the ubiquitous application JavaScript. Specifically, each JavaScript object is a key-value pair defining the ubiquitous viewport canvas id, the ubiquitous viewport object, and the necessary viewport configuration properties.

```
var viewports = [
    {
        canvasId   : 'viewportCanvas1',
        viewport   : null,
        properties : {
            renderStyle : 'VOLUME',
            view        : 'ANTERIOR',
            viewHeight  : 300
        }
    },
    {
        canvasId   : 'viewportCanvas2',
        viewport   : null,
        properties : {
            renderStyle : 'REFORMAT',
            view        : 'ANTERIOR',
            viewHeight  : 300
        }
    },
    {
        canvasId   : 'viewportCanvas3',
        viewport   : null,
        properties : {
            renderStyle : 'REFORMAT',
            view        : 'RIGHT',
            viewHeight  : 300
        }
    },
```

```
    {
        canvasId   : 'viewportCanvas4',
        viewport   : null,
        properties : {
            renderStyle : 'REFORMAT',
            view        : 'SUPERIOR',
            viewHeight  : 300
        }
    }
];
```

The last step involves using this array to instantiate ubiquitous viewport objects, configure them, and add the 3D reference cursor and bounding box visualization component graphics. The following code replaces the initialize function in the basic application.

```
function initialize() {
    var numberOfViewportsComplete = 0;
    // Setup each viewport relative to its properties in the viewports array.
    viewports.forEach(function(object) {
        var canvas = document.getElementById(object.canvasId);
        var viewport = new Ubiquitous3dViewport(canvas);
        object.viewport = viewport;
        var renderStyle = object.properties.renderStyle;
        var view = object.properties.view;
        var viewHeight = object.properties.viewHeight;
        function load() {
            viewport.loadVolume(datasetId, function() {
                viewport.setViewHeight(viewHeight, "png");
                registerViewportComplete();
            });
        }
        // Chain the configuration calls before loading the volume.
        viewport.setInitialRenderStyle(renderStyle, function() {
            viewport.setInitialView(view, load);
        });
    });

    // Calls createOverlays once all the viewports are complete.
    function registerViewportComplete() {
        if (++numberOfViewportsComplete == viewports.length) {
            createOverlays();
```

```
        }
    }

    // Create the overlay objects: BoundingBox and 3D Cursor.
    function createOverlays() {
        var cm = new Cursor3DModel();
        viewports.forEach(function(object) {
            var viewport = object.viewport;
            var canvas = viewport.getCanvas();
            var boundingBox = new BoundingBox(viewport, canvas, function() {
                viewport.addOverlayObject(boundingBox);
            });
            viewport.createAndAddOverlay(Cursor, [viewport, cm]);
        });
        // Define the location of the 3D cursor relative to the VR viewport.
        var disp = [128,64];
        var vrViewport = viewports[0].viewport;
        cm.updateRAS(vrViewport.getRASCoords(disp, null), null);
    }
}
```

The initialize function leverages the JavaScript `forEach` function to iterate through each JavaScript object in the application viewports configuration array. Each loop iteration uses the JavaScript object to instantiate and then configure a ubiquitous viewport, `Ubiquitous3dViewport`. After ubiquitous viewport instantiation, the viewport is configured through a series of chained ubiquitous viewport function calls to configure the render style, and view before loading the dataset, and setting the view height. Function chaining in JavaScript is a very common pattern for sequentially performing asynchronous operations. Since these four functions invoke their respective web service asynchronously, chaining ensures the functions are executed in the appropriate order. Once the same volume has been loaded to each of the viewports, the ubiquitous viewports are ready for the bounding box and 3D reference cursor graphics. The instantiation and addition of these graphics to each ubiquitous viewport is for demonstration of a typical four-port ubiquitous application. The result, as seen in Figure 6.2, is a four-port ubiquitous application with

graphics leveraging the ubiquitous toolkit, and indirectly the existing validated Java visualization toolkit. Such applications for diagnostic radiology require less validation than similar applications that do not leverage such a toolkit.

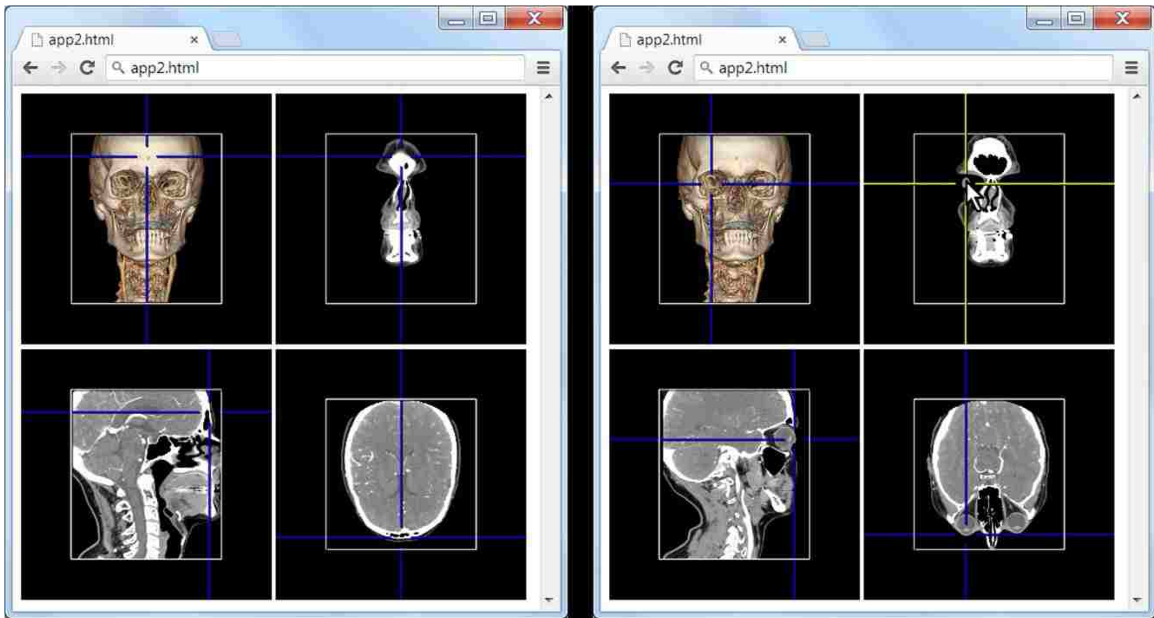

Figure 6.2: A four-port thin-Client and Zero-Footprint 3D visualization web application. The figure on the left shows the ubiquitous viewports configured with different render styles and views. Each viewport shows a bound box graphic and 3D reference cursor used to show the same coordinate in each view. The figure on the right shows the same application after 3D reference cursor is moved to highlight the right eye.

6.2 Performance

Performance is a key requirement with any platform, especially a web platform. Using QUnit, by jQuery, (The jQuery Foundation02) a test suite was developed to test the performance, in frames per second, of ubiquitous viewports rendering reformat and volume rendering views. The specific goal of this test suite is to show the impact image quality has on ubiquitous viewport mouse interaction performance. What is important is the overall effect certain configurations have on volume interaction performance, the trend is important, not the raw frames per second. This test suite examines reformat and volume rendering

ubiquitous viewports interaction performance during 1000 mouse interactions over localhost, and varies the size of the image being rendered, and the return type of the image. The size of the ubiquitous viewports remains fixed throughout these tests, at 512 pixels by 512 pixels; however, the return image sizes tested are:

1. 512 pixels by 512 pixels (application viewport size)
2. 256 pixels by 256 pixels (one quarter application viewport size)
3. 128 pixels by 128 pixels (one sixteenth application viewport size)

Figure 6.3, charts mouse interaction performance versus image render size and compression algorithm on a 512 pixel by 512 pixel reformat application viewport. Figure 6.4, charts mouse interaction performance versus image render size and compression algorithm on a 512 pixel by 512 pixel volume rendering application viewport. In these tests, the independent variables are the size of the render image, and the compression algorithm used, JPEG or PNG. The dependent variable is average frames per second across three executions of the test given each independent variable combination. Each test execution involved performing a mouse down, and then 1000 sequential mouse movements.

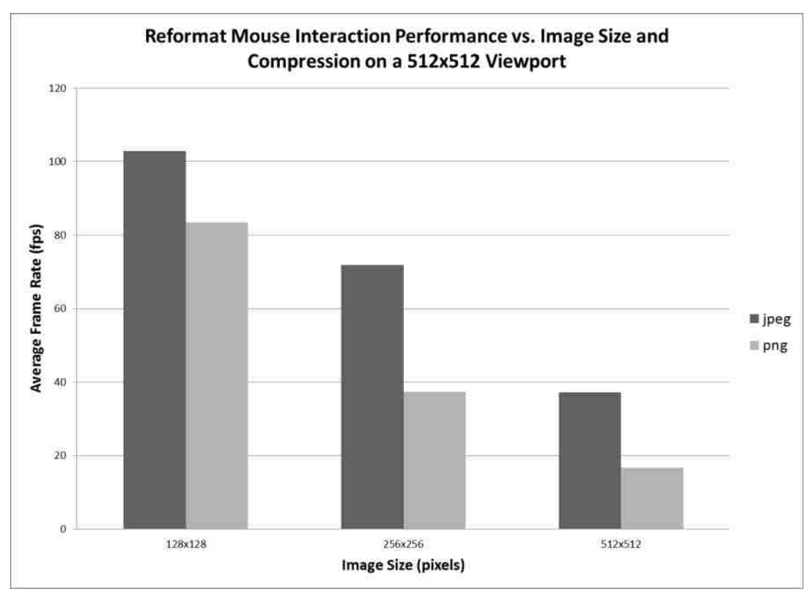

Figure 6.3: Chart of mouse interaction performance versus return image size and compression type for a 512 by 512 pixel viewport with a render style of multiplanar reformat.

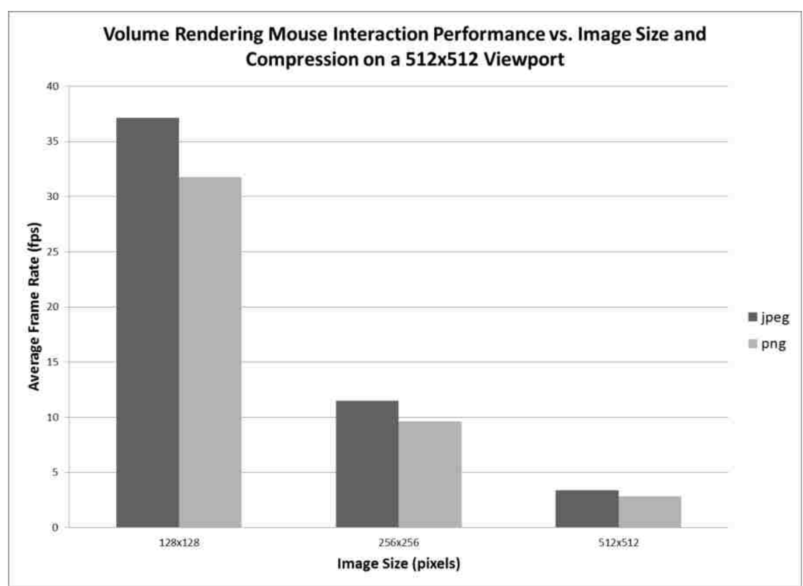

Figure 6.4: Chart of mouse interaction performance versus return image size and compression type for a 512 by 512 pixel viewport with a render style of volume rendering.

From Figure 6.3 and Figure 6.4 it is clear that the render style of a volume in a ubiquitous viewport has the largest effect on mouse interaction performance. This is to be expected, because the computational requirements between rendering a reformat image versus a volume rendering image are very different. Figure 6.5 echoes this finding. The

mouse interaction performance difference between reformat and volume rendering ubiquitous viewports is at least 62%, and performance degrades to 91% as the rendered image size increases and using JPEG for image compression. What is clear from Figure 6.5 is that the render style of the ubiquitous viewport has a much greater impact on performance than the compression algorithm with local web services.

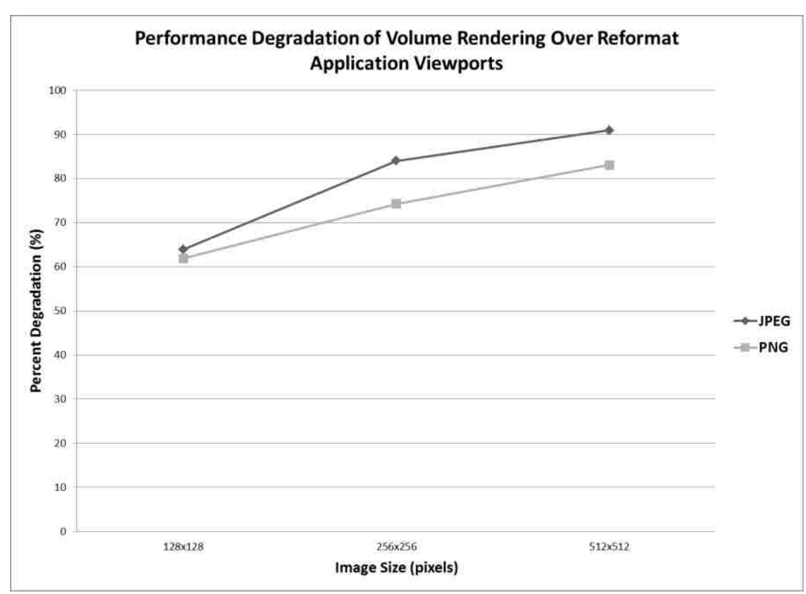

Figure 6.5: Shows performance degradation as a function of render style, with varying image render sizes and image compression algorithms. The render style of the viewport has a much greater impact on performance than the compression algorithm with local web services. The performance degradation is similar between compression algorithms.

Figure 6.6 shows how the image compression algorithm used to transport the rendered image from the client to the server impacts the mouse interaction performance of reformat and volume rendering render styles on ubiquitous viewports. As seen in the chart, PNG compression over JPEG compression has a relatively small impact on overall performance of a volume rendering application viewport, at around 15% for each image size. However, PNG compression over JPEG compression has a greater overall impact on performance for a reformat application viewport, anywhere from to 19% to 55%, based on image size.

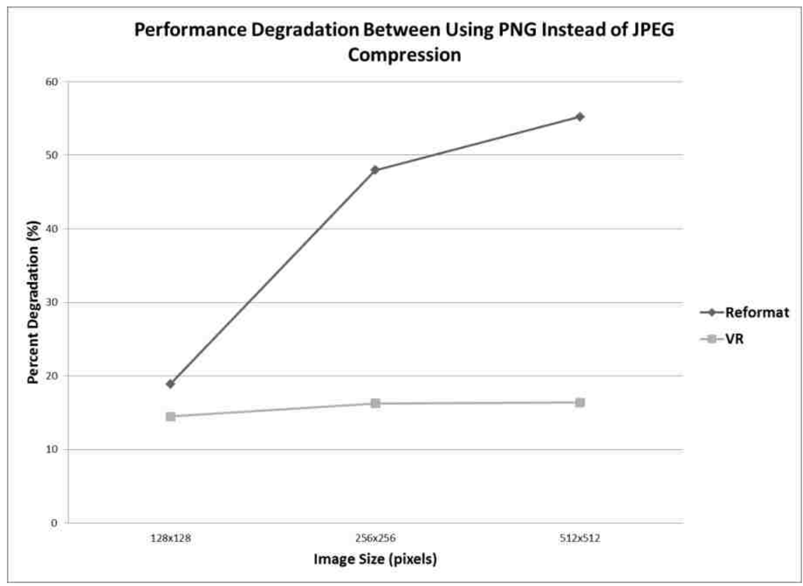

Figure 6.6: Shows performance degradation as a function of image compression algorithm, with varying render styles, and image render size. Image compression algorithm has a relatively small impact on interaction performance with a volume rendering render style, but has a very large impact on interaction performance with a reformat render style.

Based on these results and trends from localhost volume interaction performance, reformat ubiquitous viewports should request JPEG compression during mouse interaction, and should decrease the image size to further increase mouse interaction performance. On the other hand, volume rendering application viewport will see the largest mouse interaction performance gains by requesting image renderings that are a fraction of the size of the application viewport. Although the jpeg over png image compression has a relatively small impact on interaction performance, volume rendering images should also be transmitted as jpeg.

CHAPTER 7: CONCLUSION

This chapter concludes this thesis by summarizing the ubiquitous application supporting platform, discusses the broader impact of this work, and future directions.

7.1 Summary

In today's software world, new applications should be designed using a mobile first approach because of society's adoption of mobile connectivity. As the IoE expands to incorporate new devices, technologies, and applications, this paradigm will become even more relevant. The diagnostic radiology space and healthcare in general is a slow adopter of new software technologies and patterns. Desktop applications in the diagnostic radiology space commonly leverage a validated toolkit. Such toolkits not only simplify desktop application development but minimize the scope of application validation. For these reasons, such a toolkit is an important piece of a company's software portfolio. This thesis investigated an approach for the leveraging of such a Java validated toolkit for the purpose of creating numerous ubiquitous applications for diagnostic radiology. Just as in the desktop application space, leveraging such a toolkit minimizes the scope of application validation.

In this thesis, a ubiquitous application is an application that can be executed by the widest range of electronic devices, providing true anytime and anywhere access to for volume view and manipulation in the space of diagnostic radiology. Specifically, this pattern leverages the Internet browser of electronic devices for ubiquitous application development, adds these applications to the IoE.

This thesis provided a solution to simplify ubiquitous application development focused on 3D volume visualization and manipulation using a ubiquitous toolkit. Specifically,

the ubiquitous toolkit exposes a ubiquitous viewport that can be added to an application's graphical user interface. This ubiquitous viewport is a self-contained entity that can be thought of as a widget,. In this architecture, each ubiquitous viewport leverages an existing validated Java visualization toolkit for rendering and manipulation of volumes through a client-server communication layer. This ubiquitous toolkit and ubiquitous viewport exposes an easy-to-use local JavaScript API for the purpose of supporting the development of rich single page ubiquitous applications. The ubiquitous toolkit's client-server communication layer hides all the necessary communication with the server-side web services. Specifically, the server-side web service layer is a Java JAX-RS RESTful web service wrapper around the validated visualization toolkit. This wrapper fundamentally acts as a proxy between the ubiquitous toolkit and the validated visualization toolkit.

With the end goal of supporting the development of ubiquitous applications leveraging an existing validated toolkit discussed in depth, this thesis ended with an evaluation of the overall architecture in terms of application development and performance.

The evaluation began with building the most basic of ubiquitous applications, containing only a single ubiquitous viewport which was expanded to a four port application. The four port application showed how to build a ubiquitous application containing four ubiquitous viewports. This example showed how to include graphic overlays to the viewports, including the bounding box graphic and the 3D cross-reference cursor. The bounding box graphic is used to visually identify the boundary of the volume, and a 3D cross-reference graphic is used to identify the same 3D point in each ubiquitous viewport.

The evaluation section ended with a discussion of ubiquitous viewport mouse interaction performance in terms of viewport render style, and rendered image size and compression algorithm. To summarize the results, reformat ubiquitous viewports should

request jpeg compression during mouse interaction, and should decrease the image size to further increase mouse interaction performance. On the other hand, volume rendering application viewport will see the largest mouse interaction performance gains by requesting image renderings that are a fraction of the size of the application viewport.

Overall, this thesis provided a flexible and scalable approach to developing ubiquitous applications that leverage an existing validated toolkit through industry standard technologies, patterns, and best practices. The overall body of work includes a client-server architecture based on lightweight clients and fully featured servers used to distribute server-side 3D visualization algorithms and components to a multitude of mobile clients. The resultant work supports easy to build ubiquitous applications that minimize the scope of validation for diagnostic radiology.

7.2 Broader Impact

The overall platform presented in this thesis is generic. This non-healthcare specific design pattern can be used to revive any existing toolkit or modular implementation to support ubiquitous applications in IoE and Industrial Internet domain. This approach can be used to wrap a toolkit or implementation in a web service wrapper that “webifies” the API through web services that can be consumed by toolkits or web applications. In this case, if the underlying toolkit or implementation exists in a regulated space and is validated, this approach can be used to create a ubiquitous toolkit that supports easy ubiquitous applications development that require minimal validation. Importantly, this layered architecture can be used to add any toolkit or implementation into the IoE and Industrial Internet paradigm, see Figure 7.1.

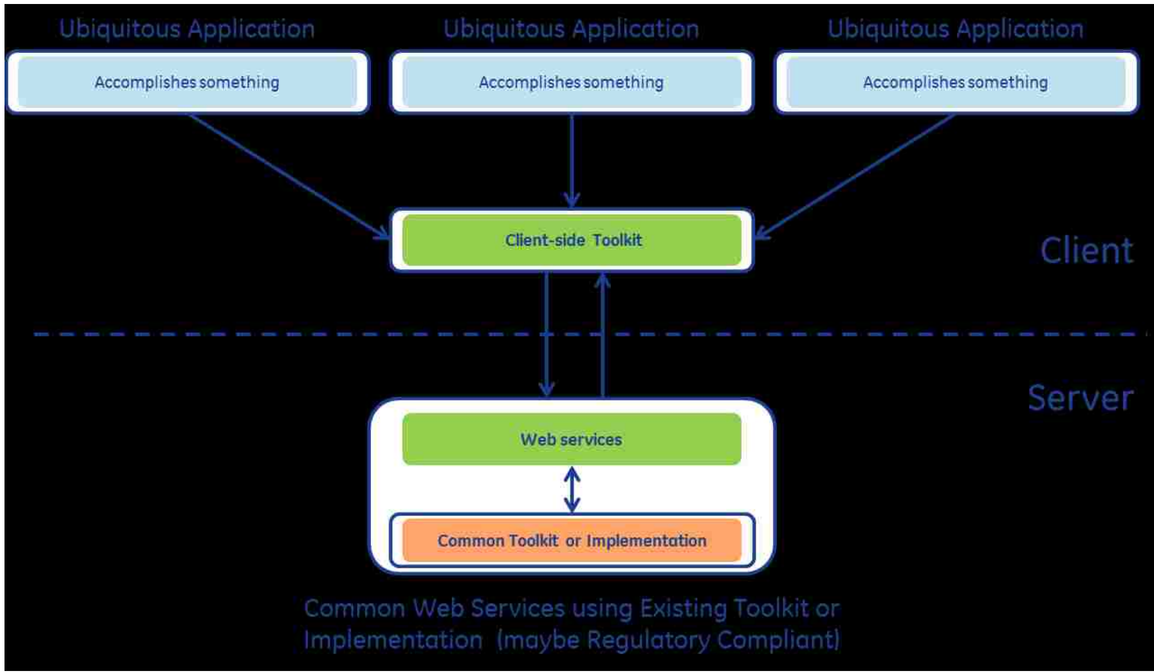

Figure 7.1: This figure shows the generic architecture presented in this thesis. Although this thesis focuses on diagnostic radiology and the ubiquitous applications, this approach is generic. This architecture can be used to "webify" an implementation.

7.3 Future Work

Given the decisions made for the creation of a ubiquitous application supporting platform leveraging an existing validated Java visualization toolkit, there are numerous future improvements that can enhance ubiquitous applications through changes to the ubiquitous toolkit.

The most obvious future enhancement for the platform is the support for bidirectional client-server communication through WebSockets. Although WebSocket communication was out of scope of this thesis, for several reasons ranging from the standard still evolving, and overall complexity, performance improvements transporting images through WebSockets is a point of interest. Of specific interest is a comparison of volume integration frame rate through HTTP and WebSocket communication. Despite the inherent implementation complexities surrounding bidirectional client-server communication through

WebSockets, ubiquitous applications are impervious to any such updates because of the ubiquitous toolkit's transparent and modular client-server communication layer.

Another future enhancement for this ubiquitous application supporting platform involves modularizing the server web services. Architecturally, the entire collection of web service can be broken into groups of common services that accomplish a common goal. Each of these collections can be built in isolation and can execute in isolation, and concurrently in different processes running in the same or different machines. In regards to the ubiquitous toolkit, this means the toolkit is built as a "mashup" of service existing at different URL bases. In this architecture, the server-side web services would still leverage the same existing validated Java visualization toolkit but instead of one web service layer existing in one program, many web service programs can be created, each exposing related web services. The key to this approach is this way the backend web services can be parallelized to increase ubiquitous viewport responsiveness.

Lastly, the overall platform presented in this thesis is generic; staying in the space of diagnostic radiology, the next logical expansion of this ubiquitous supporting platform is supporting 2D visualization for diagnostic radiology. Again, if the underlying 2D visualization toolkit is validated, this architecture can be mirrored to support 2D ubiquitous applications that can be easily developed and validated.

To summarize, although this platform supports the necessary constructs for easily building ubiquitous applications that are easily validated, there are some enhancements. The first is implementing WebSocket bidirectional client-server communication in the web service layer and the ubiquitous toolkit. This update will likely improve the observed mouse interaction performance on ubiquitous viewports, and will allow for bidirectional communication between the client and server. Because this approach is generic, it can be

used to support 2D ubiquitous applications in the diagnostic radiology space, and this approach can be used to “webify” an existing implementation in any space.

BIBLIOGRAPHY

(2010). In B. Burke, *RESTful Java with JAX-RS* (p. 27). Sebastopol: O'Reilly Media, Inc.

Cisco. (n.d.). *Internet of Everything*. Retrieved from Cisco: http://www.cisco.com/web/about/ac79/innov/IoE.html?_sm_au_=iVVDL7tKFHWnQpSR

DURANDAL. (n.d.). *DURANDAL Single Page Apps Done Right*. Retrieved from DURANDAL: http://durandaljs.com/

Food and Drug Administration. (2002, January 11). *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*. Retrieved from FDA: http://www.fda.gov/medicaldevices/deviceregulationandguidance/guidancedocuments/ucm085281.htm#_Toc517237969

GE. (n.d.). *Introducing the Industrial Internet*. Retrieved from GE imagination at work: http://www.ge.com/stories/industrial-internet

Google. (n.d.). *ANGULARJS*. Retrieved from ANGULARJS by Google, HTML enhanced for web apps!: http://angularjs.org/

HHS. (2006). *HIPAA Security Guidance*. HHS.

(2001a). In J. Hunter, & W. Crawford, *Java Servlet Programming* (p. 16). Sebastopol: O'Reilly Media, Inc.

(2001b). In J. Hunter, & W. Crawford, *Java Servlet Programming* (p. 15). Sebastopol: O'Reilly Media, Inc.

IBM. (n.d.). *Defining the REST interface*. Retrieved from IBM: http://pic.dhe.ibm.com/infocenter/initiate/v9r5/index.jsp?topic=%2Fcom.ibm.composer.doc%2Ftopics%2Fr_composer_extending_services_creating_rest_service_rest_interface.html&_sm_au_=iVVMQjnVLDPQk3MR

Knockout. (n.d.). *Knockout*. Retrieved from Knockout: http://knockoutjs.com/

Kotamraju, J. (2013, July). *Java API for JSON Processing: An Introduction to JSON*. Retrieved from ORACLE: http://www.oracle.com/technetwork/articles/java/json-1973242.html

Laine, M. (n.d.). *RESTful Web Services for the Internet of Things*. Retrieved from http://media.tkk.fi/webservices/personnel/markku_laine/restful_web_services_for_the_internet_of_things.pdf

Landgrave, T. (n.d.). *Thin Client Applications: End of the Road for Microsoft? Not Likely!* Retrieved from Microsoft TechNet: http://technet.microsoft.com/en-us/library/cc751249.aspx

Oracle. (2011a, February 10). *"Package javax.servlet" section of the Java 6 EE API documentation*. Retrieved from http://docs.oracle.com/javaee/6/api/javax/servlet/package-summary.html

Oracle. (2011b, February 10). *"Package javax.ws.rs" section of the Java 6 EE API documentation*. Retrieved from http://docs.oracle.com/javaee/6/api/javax/ws/rs/package-summary.html

Oracle. (2013a, September). *"Java API for WebSocket" section of the Java 7 Enterprise Edition tutorial*. Retrieved from ORACLE: http://docs.oracle.com/javaee/7/tutorial/doc/websocket.htm

Oracle. (2013b, September). *"What Are RESTful Web Services?" section of the Java 7 Enterprise Edition tutorial*. Retrieved from ORACLE: http://docs.oracle.com/javaee/7/tutorial/doc/jaxrs001.htm#GIJQY

Oracle01. (n.d.). *Simple Object Access Protocol (SOAP) for Java*. Retrieved from ORACLE: http://docs.oracle.com/cd/A97630_01/appdev.920/a96616/arxml11.htm

Oracle02. (n.d.). *Java(TM) EE 7 Specification APIs*. Retrieved from http://docs.oracle.com/javaee/7/api/overview-summary.html

Park, Y. W., Guo, B., Mogensen, M., Wang, K., Law, M., & Liu, B. (2010). A Zero-Footprint 3D Visualization System Utilizing Mobile Display Technology for Timely Evaluation of Stroke Patients. *SPIE, 7628*.

PC Magazine01. (n.d.). *Encyclopedia, Definition of: component*. Retrieved from PCMAG.COM: http://www.pcmag.com/encyclopedia/term/40111/component

PC Magazine02. (n.d.). *Encyclopedia, Definition of: toolkit*. Retrieved from http://www.pcmag.com/encyclopedia/term/52987/toolkit

PC Magazine03. (n.d.). *Encyclopedia, Definition of: widget*. Retrieved from http://www.pcmag.com/encyclopedia/term/54456/widget

Satrom, B. (2010, December). *MSDN Magazine December 2010 Issue*. Retrieved from Behavior-Driven Development with SpecFlow and WatiN: http://msdn.microsoft.com/en-us/magazine/gg490346.aspx

The Apache Software Foundation. (n.d.). *karaf*. Retrieved from karaf: http://karaf.apache.org/

The jQuery Foundation01. (n.d.). *jQuery*. Retrieved from jQuery: http://jquery.com/

The jQuery Foundation02. (n.d.). *QUnit: A JavaScript Unit Testing Framework*. Retrieved from QUnit: http://qunitjs.com/

W3C. (2004, February 11). *Web Services Architecture*. Retrieved from W3C Working Group Note: http://www.w3.org/TR/ws-arch/

W3C. (2007, April 27). *SOAP Version 1.2 Part 0: Primer (Second Edition)*. Retrieved from W3C Recommendation: http://www.w3.org/TR/2007/REC-soap12-part0-20070427/?_sm_au_=iVVDL7tKFHWnQpSR

W3Schools01. (n.d.). *AJAX - Create an XMLHttpRequest Object*. Retrieved from w3schools.com: http://www.w3schools.com/ajax/ajax_xmlhttprequest_create.asp

W3Schools02. (n.d.). *HTML5 Canvas*. Retrieved from w3schools.com: http://www.w3schools.com/html/html5_canvas.asp

APPENDIX A

Core RESTful Web Services

Feature	Web API	HTTP Operation	Return Type
Viewport resizing	`setSize`	GET	image/jpeg or image/png
Set render style	`setRenderStyle`	GET	image/jpeg or image/png
Set view to preset	`setView`	GET	image/jpeg or image/png
Coordinate transforms	`getRasCoords`	GET	application/json
	`getDisplayCoords`		
Volume geometry	`getVolumeGeometry`	GET	application/json
Render engine camera	`setEye`	GET	image/jpeg or image/png
	`setLook`		
	`Setup`		
	`setCamera`		
	`getEye`		application/json
	`getLook`		
	`getUp`		
	`getCamera`		
Setting mouse tool	`setMouseTool`	GET	text/plain
Mouse drag	`mouseDown` `mousemove` `mouseUp`	GET	image/jpeg or image/png
Mouse drag image size	`setDownsampleFactor`	GET	text/plain
Mouse drag lossy/lossless	Each web API that returns an image allows requires the mime type of the return image.		

Save state	saveState	GET	application/xml
Restore state	loadState	POST	text/plain
	renderImage	GET	image/jpeg or image/png