INSERTION OF PRIVACY SERVICES IN PRIVACY ARCHITECTURE FOR WEB SERVICES (PAWS)

by

Ajith Winston Bryn

Submitted in partial fulfilment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
March 2014

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Huge growth of the Internet is due to the large number of websites and web services through which information is easily accessible. E-commerce and e-services obtain much private data from users for various reasons such as advertising, marketing, etc. Collection, storage, and usage of private data are subject to various standards, privacy laws, and regulations. To adhere to these legal requirements, many privacy services, such as secure data transmission, authentication, notice, and consent, are required. Inclusion of these required privacy services early in the life cycle of the software development is preferred and advocated, but not fully adhered to. Inclusion of privacy services in legacy software and currently developed software is required. We describe software architecture and a system for automatic inclusion of privacy services, under the supervision of privacy expert, into web pages after the development phase of the Software Development Life Cycle. This will help organizations to adhere to standards, privacy laws, and regulations when collecting private data online from its clients. We also describe a prototype that we have developed as a proof-of-concept to demonstrate the feasibility of our approach.

LIST OF ABBREVIATIONS USED

FIPP	Fair Information Practices Principles
FTC	Federal Trade Commission
HTTP	Hypertext Transfer Protocol
KB	Knowledge Base
PAWS	Privacy Architecture for Web Services
PbA	Privacy by Architecture
PbD	Privacy by Design
PE	Privacy Engineer
PI	Private Information
PMRM	Privacy Management Model and Methodology
ROA	Resource Oriented Architecture
SDLC	Software Development Life Cycle
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer
UDDI	Unique Description Discovery and Integration
URL	Unique Resource Locator
WWW	World Wide Web
XACML	eXtensible Access Control Mark-up Language

ACKNOWLEDGEMENTS

I would like to thank The Lord Jesus for giving me the strength, wisdom, and knowledge to complete my thesis successfully.

I wish to express my sincere gratitude to my supervisor Dr. Peter Bodorik, whose encouragement, dedication, and expert guidance from the beginning of my course tenure to the final level has enabled me to develop, understand and make progress in the research area. I would like to thank Dr. Dawn Jutla, whose encouragement and care has provided great confidence in me to make progress in the research area.

I would like to thank my Father A. Bryn Green Anand, and my Mother B. Jothy Bryn for providing the opportunity to study and, love of God they showed that has helped me achieve great things in my life. I would like to thank the prayers, love and, time they have invested in me.

I would like to thank, Satish Samuel (Anna), Catherine Samuel (Akka), Sheryl Samuel, and Serena Samuel for their prayers, guidance, motivation and the love of God they showed throughout my time in Halifax. Without their support, constant care and, love I would not have been able to stay in Halifax and complete my degree.

I would like to thank my church family for providing continuous support and love. Church Pastors, brothers, sisters, fellowship friends in India and friends for being there and believing in me.

CHAPTER 1 : INTRODUCTION

1.1 Motivation

Internet is defined as the network of networks that is able to connect millions of private, public, academic, business and government networks. Internet has become much more vital in the last few decades and has changed the way people view and do things. This huge growth of the Internet is due to the large number of websites and web services through which information is easily accessible. Huge amount of information and data are stored on the organizations' public, private or hybrid database(s). Online shopping or services have become popular over the last few decades and this has led to a tremendous growth of many companies. It is necessary for people to have secure mechanisms to conduct online transactions in e-commerce, e-business, e-government, e-health, and other online activities facilitated by Internet. This leads to the need for subjects, users who provide private information, and organizations, who collect and store subjects' private information, to take into account protection of Personal Information (**PI**) (Olivier & Oberholzer, 2005).

There has been a huge growth in forms on web pages collecting personal information from subjects. These forms mostly use web services to store private data. Subjects knowingly and unknowingly reveal much private information through the Internet and this necessitates standards and laws to address this issue. FIPPs comprise a set of widely accepted guidelines that are followed in e-business. The core principles of FIPPs are Notice, Choice, Access and Security (Pitofsky, Anthony, Thompson, Swindle, & Leary, 2000). Notice informs the user, subject, that he/she is asked to provide private information through the forms. Consent is used to obtain approval from the subject on the usage and dissemination of the collected private data the subject provides. Access provides for authentication when accessing the private data, while security enforces secure transmission of the private data. Although privacy services, of the Privacy Management Reference Model (**PMRM**) OASIS standards track document (Sabo, Willett, Brown, & Jutla, 2013), follow the recommendations of the Privacy By Design principles (**PbD**) (Cavoukian A., 2013) and provide for notice, consent, access, and security, PMRM is still in the standard's

approval phase and hence has not been adopted by organizations in software implementations.

Jutla, Bodorik, and Ali (2013) proposed tools that help in addressing privacy conserns in the software requirements specification phase to ensure that privacy services are included when collecting and using PI and thus support adoption of PMRM. If PMRM is acceptecd as a standard and followed by the industry, the tool would help in that ensuring notice, consent, and security services are included in the requirements specification phase and thus ensuring conformance to PMRM guidelines based on the PbD principles. However, there is still concern about inclusion of privacy services in legacy software and current development of software without awareness of privacy requirements.

1.2 Research Problem

There is still software being developed without the consideration of privacy and security and, of course, much software is in use that has been developed without taking PbD, and hence privacy and security, into consideration. In this thesis we address the issue of injecting privacy services on the existing software modules developed without privacy services. We focus on the stages after the development stage in the Software Development Life Cycle (**SDLC**). Privacy Architecture for Web Services (**PAWS**) was devised to provide a knowledgebase and mechanisms/tools to fulfill privacy requirements in collecting and using private data. As part of PAWS, Privacy Engineer (**PE**), or privacy steward, should be able to inject privacy services into software that collects private data after the development phase in the SDLC. We use the best practice of separation of duty to decouple software development and privacy concerns. This will also enable existing software, developed without privacy consideration, to include privacy services.

1.3 Objectives

The objectives of this thesis are addressing the research question in a specific environment wherein the private data is collected from a subject by an organization using forms in web pages, such that the collected private information is delivered to the organization using Restful web services within the environment of PAWS. The objectives of this thesis are:

1. Architect and design software for the concept of automatically providing notice on the use of collected private data, obtaining consent for collection of private data, performing authentication if required, and ensuring secure delivery of collected private data to the organization's DBs.
2. Identify informational requirements for the architected software.
3. Create a prototype as a proof-of-concept.

1.4 Outline

Chapter 2 provides a literature review. Chapter 3 shows the software architecture and the required information. Chapter 4 then shows the prototype developed as a proof of concept for the architecture. Chapter 5 shows the execution delays in the web page with privacy services under various scenarios. Finally, Chapter 6 provides a summary and conclusions and suggestions for further research.

CHAPTER 2 : LITERATURE REVIEW

There has been huge growth in the Internet technologies and the number of new users has been increasing exponentially. The new users do not know that every post that they make on a forum, every web page they access, every web chat they do, every video call they make is monitored and logged. Users do not realize the large amount of private information that they provide explicitly or implicitly while accessing online services over the Internet to some unseen third party and how this collected information is used for sales, marketing, etc. (Goldberg, Wagner, & Brewer, 1997) Due to standards and legal regulations, organizations need to take necessary measures when collecting private data and to safeguarding collected customers' private data. In the following, we review the various laws and regulations that are imposed by the governments to safeguard citizens' private data. We then provide background on approaches to providing privacy for collected PI; technologies available for web services development, including technologies used for authentication and authorization; and then briefly review PAWS.

2.1 Laws and Regulations

There are various laws and regulations, imposed by governments, by which organizations must abide concerning the private data that they collect. Fair Information Practices Principles (**FIPP**) are guidelines, of the Federal Trade Commission (**FTC**) (Commission, 1998), to be followed by organizations when collecting, storing, and using private data in the field of electronic business. The FTC has been in the field of addressing privacy issue since 1995 and its report was released in 1998.

In Canada, FIPP led to the enactment of the Personal Information Projection and Electronic Documents Act that governs how organisations gather and manage private data. It states that individuals are given the right to access and request the private information that the companies keep about them (Justice, 2011). The customers should be able to make changes to the personal information that companies have collected about them. In the United States the Freedom of Information Act is a law that gives anyone access to information that the federal government has about them. The Privacy Impact Assessments, which fall under the privacy act, ensure that the public is aware that private information is

being gathered from users, that only private information that is necessary to administer the program is collected, and appropriate security measures are taken in order to secure the privacy of individuals' data (Act, 2002).

The core set of principles addressed by the FIPP is as follows:

1. **Notice**: The consumers must be provided with prior notice before gathering private information about them. This requires companies to explicitly notify the consumers when private information is gathered from them and to provide details on the following:
    - Identification of the entity collecting the data
    - Identification of the recipients of the data
    - The category of information/data collected
    - Identification if the data collected are voluntary/optional or mandatory
    - The steps taken to protect the confidentiality, integrity and quality of data.
2. **Choice/ Consent**: This means that the consumers are given options by which they can control how the data is used by the company. Usually companies supply private data collected from the users to third party companies for various reasons such as marketing, sales, promotions, etc. There should be options by which the subject can control the flow of their private data to third-party companies. There are usually two consent models that can be followed, *opt-in* or *opt-out* (Spiekermann & Cranor, 2009). The *opt-in* model assumes that the user must explicitly give permission to the company before the company can use the private information for other purposes. The *opt-out* model is when the user must explicitly decline to provide approval to the company to use their private information for other purposes. Alternatively the user can be provided with another mechanism in which the user can control how company can use his/her private information only for certain purposes by, for instance, using checkboxes.
3. **Access/Participation**: There must be options by which the user is able to see his/her private data to check for accuracy and make changes if required.
4. **Integrity/ Security**: Companies, after collecting information from the user, can verify the accuracy of the information by comparing the data with reliable

databases. After verification the data can be provided to the subject to approve details. Access to the data can be restricted to only certain employees who need access to the information. There can be encryption mechanisms incorporated to enable secure transmission of the private data during storage and retrieval.

5. Enforcement/Redress: There should be enforcement measures to ensure that the company follows FIPP.

2.2 Approaches to Ensure Privacy

There are currently two different models that exist for implementing privacy services in software systems: Privacy by Architecture and Privacy by Design.

2.2.1 Privacy by Architecture (PbA)

In this model, FIPPs need not be followed as in this architecture the user's private information is stored on the client/user system rather than in the company's database. This gives the consumer/user proper control of his/her information. Thus the security measures for storage of the private information are transferred to the client. However, this is only a partial solution, as companies are forced by legal requirements to keep certain client information involved in transactions.

Additionally, aggregation can also be used in certain situations to avoid storage of client information in companies' DBs. For instance, in cases of collaborative filtering systems, private information is stored on the customers' system and an aggregate is calculated on the client's machine and sent to the organisation's database for later processing (Canny, 2002).

2.2.2 Privacy by Design (PbD)

This approach implements all of the FIPPs at software development time (Cavoukian A. , 2013). Although PbD advocates inclusion of privacy services in the early cycles of the SDLC, the approach is just being standardized and hence has yet to be adopted by the software industry. Thus, much software has been and is still being developed, particularly by small and medium enterprises, without inclusion of privacy services for collection, storage, and use of private information.

To support PbD, extensions to UML have been proposed to help software engineers to specify the requirements for privacy services for application (Jutla, Bodorik, & Ali, 2013), namely for showing notice, obtaining consent, and secure data transfer and storage.

2.3 Web Services

The exponential growth of the number of Internet users has been accompanied by the exponential growth of web services. Many organisations are moving to the Cloud, in which Service Oriented Architecture (**SOA**) based on web services is a prominent feature. This has led to the adoption of SOA and web services by many organisation for software architecture and hence for software integration. Based on the World Wide Web Consortium's, a web service is defined as a software module that is able to perform inter-operable machine-to-machine interaction over the Internet. There are two different approaches towards creation of web services – they are discussed below.

2.3.1 SOAP-based Web Services

SOAP-based web services consist of three entities (Belqasmi, Singh, Melhem, & Glitho, 2012):

- Service provider, who creates SOAP web services and publishes them.
- Service registry, which enables online service discovery of the web service and makes the created web service available over the Internet.
- Service requestor, who finds the SOAP web service by querying the service registry. Then the requestor builds a service description through which it communicates with the service provider.

Web services interact with applications and each other over the Simple Object Access Protocol (**SOAP**). The SOAP messages are XML-based and are transferred over Hypertext Transfer Protocol (**HTTP**). The SOAP-based web services are published using Web Services Description Language, which is based on XML, to describe what the web services do and their methods, parameters, and binding information. Unique Description Discovery and Integration (**UDDI**) is the service registry that enables the developed SOAP web services to be accessible over the Internet.

2.3.2 RESTful Web Services

Representational State Transfer (**REST**) (Fielding, 2000) is based on three design principles: addressability, uniform interface, and statelessness. For addressability, the RESTful web services consider resources as entities addressable by Unique Resource Locators (**URLs**). The uniform interface is because the web service requests and replies are always using URLs and the stateless is due to the fact that the base protocol is HTTP, which in itself is stateless, and hence web service requests are independent from each other.

A resource can be a data record from the database, an image on the web server, an integer value in an array of integers, etc. (Hartikainen, Tampere, Laitkorpi, Ruokonen, & Systa, 2011) There are four major operations that are performed by a RESTful web service: create, read, update, and delete, which are mapped to the HTTP verbs of PUT, GET, POST, and DELETE, respectively. It should be noted, however, that there are no restrictions on the usage of the HTTP verbs with respect to the operations that they perform. For instance, a DELETE HTTP verb can be used to create a resource as well. As web services are stateless, they can scale well when the number of web requests increases significantly.

The advantages of RESTful web services over SOAP-based web services are: the time to create RESTful web services is shorter; they do not require client-specific implementation; RESTful web services are stateless and thus can scale up well with the number of requests; and RESTful web services work with HTTP and thus do not require additional communication protocol layer (Chanda & Foggon, 2013). Thus there has been great adoption by industry to use RESTful web services particularly in areas where strict formality, such as in the financial domain, is not required. This thesis is scoped to RESTful web services as they have major advantages over SOAP-based web services and their current adoption outpaces the adoption of SOAP-based web services.

2.3.3 Security Mechanisms

Not all of the web services must be sent securely as this would cause a downfall to the performance of the web service – only those web services that use private data must be secured. There are various security mechanisms that can be used in RESTful web services. The RESTful web services can be secured by using the HTTPS protocol, OAuth 2.0, and

a custom security model by using headers to authenticate the RESTful web services (Serme, Oliveira, Massiera, & Roudie, 2012).

2.3.3.1 Securing using Secure Socket Layers

The protocol that is used for secure transmission at a socket layer is HTTPS. It provides a secure connection between the server and the client by means of certificates that are exchanged between the server and the client. The server gives each user a certificate and unless there is a certificate the server cannot authenticate the client. The keys are exchanged with the help of the Public-Key Infrastructure, while the secure transfer of the actual message achieved using symmetric key encryption. The client and server negotiate which of the symmetric key encryption algorithms, in the ciphersuit, should be used. The certificates are identified with the help of Certificate ID, which is a serial number followed by the issuer‘s name. The name of the issuer in our thesis is identified using X509 certificate. The Secure Socket Layer (**SSL**) addresses the issue of secure transmission in the RESTful web services.

2.3.3.2 OAuth 2.0

The OAuth2.0 is the new version of OAuth 1.0, which was released in 2006. OAuth 2.0 enables third party applications to obtain restricted access to an HTTP service. OAuth has four roles:

Resource owner: An entity who can grant access to the protected resources. The owner is often referred to as end-user.

Resource server: The server hosting the protected resources that is capable of accepting web requests and responding with either declining access or providing access access to the protected resources through access tokens.

Client: Software, such as application or a web service that is making requests to access the protected resources.

Authorization server: The server that issues access tokens to the client after making sure the resource owner is authenticated and authorized.

The protocol flow to access the protected resource is shown in Figure 2.1. An authorization request is sent to the resource owner and the owner replies with an authorization grant back to the client. The client now sends the authorization grant along with the credentials to the authorization server that, upon successful authorization, replies with an access token. Finally, the client supplies the access token to the resource server and the resource is sent back to the client.

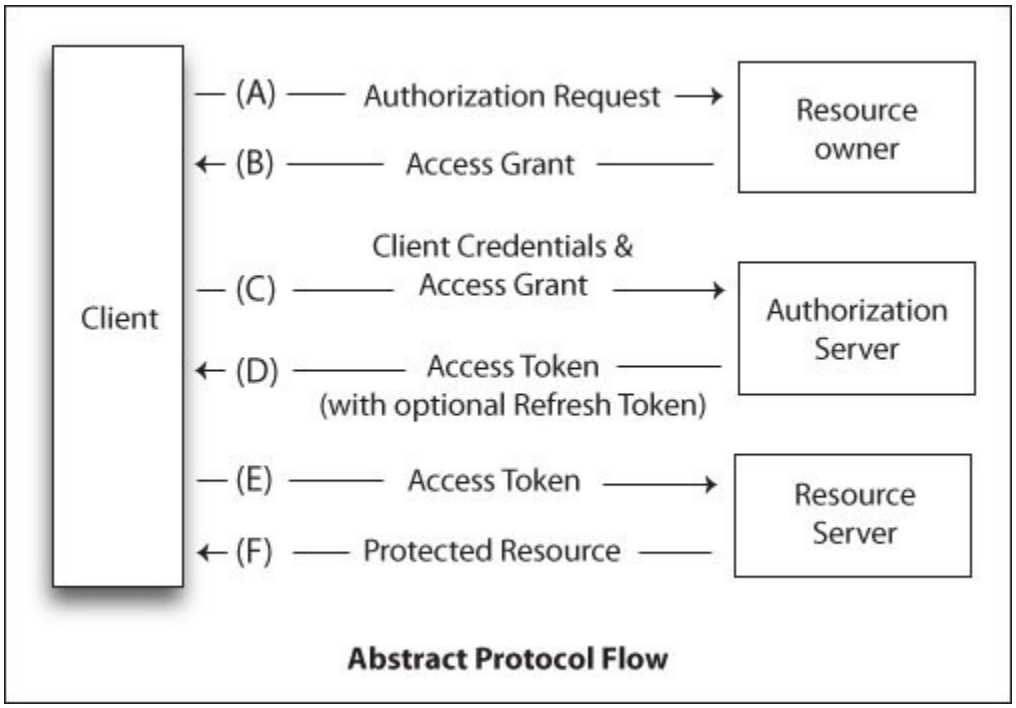

Figure 2.1 Introduction to Facebook APIs
(Adopted from IBM Developer Works, 2010)

2.3.3.3 Use of Hidden Fields on the Forms

Hidden fields can be used by the form's script to store information, which is required by the form's script or web services invoked by the form, that should not be visible to the user. However, without encryption, hidden fields do not provide any security in data transfer.

2.3.3.4 XACML Policy

eXtensible Access Control Mark-up Language (**XACML**) is an OASIS standard, which promotes open standards to offer potential to reduce the cost, increase innovation, widen global markets and protect the right choice of technology (Xu, Wijesekera, & Zhang, 2011), is used to enforce authorization. XACML ensures that the right person has the right level of access to the resource on the web server.

XACML is an *xml* file that holds policies specifying authorization. It is an Attribute Based Access Control system. The XACML is structured into three levels of elements: Policy Set, Policy, and Rule. A Policy Set holds a number of policies and Policy Set elements. A Policy can contain any number of Rule elements. A policy rule element is expressed as: Subject, Resource, Action, and Environment. The Subject, which has one or more attributes, denotes a person, device, or entity accessing the resource. The Resource denotes the actual resource that is accessed, in our case by a web service. In the case of a REST web service, it is the URL that identifies the resource. Action is used to denote the action to be performed on the web service. Environment denotes the context of the web application.

2.4 Privacy Architecture for Web Services (PAWS)

In (Bodorik, Jutla, & Dhillon, 2009), architecture is proposed to support provision of privacy in software environments based on web services (see Figure 2.2). The architecture provides for monitoring and logging of web service requests/replies and for gaining knowledge on the use of private data by applications and web services. This knowledge is used to enforce the privacy policies on the use of private information in SOA environment based on WSA.

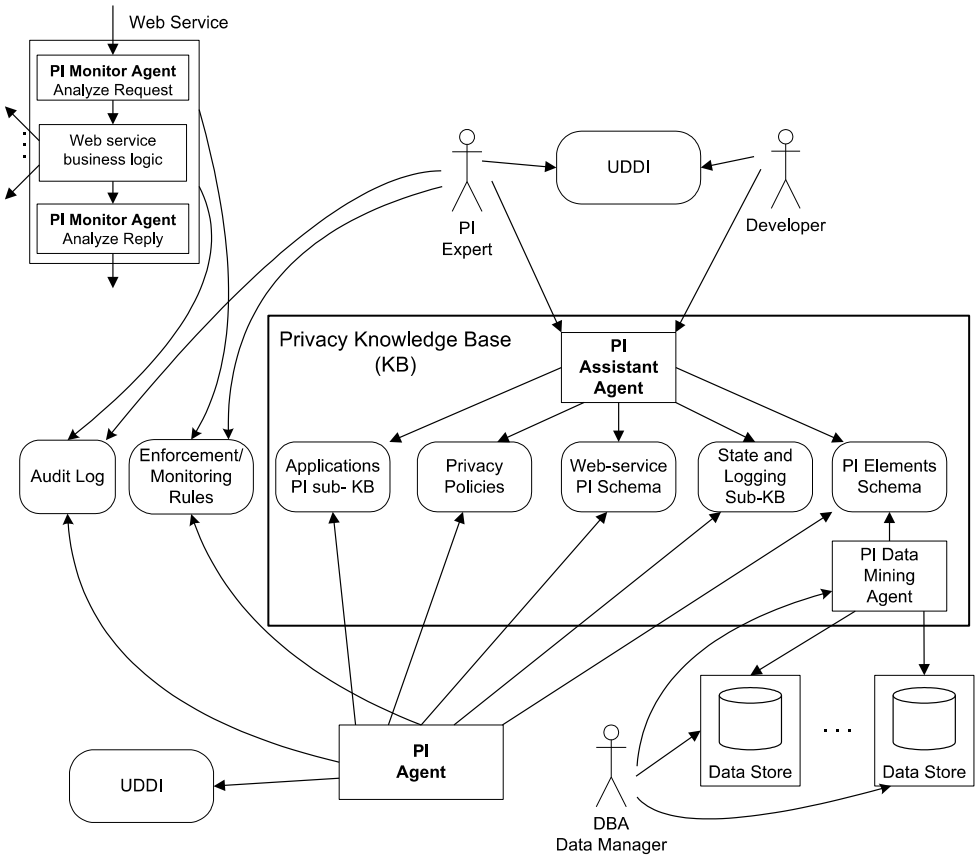

Figure 2.2 Privacy Architecture for Web Service
(Adopted from Bodorik, Jutla, & Dhillon, 2009)

The PI monitor agent is used to intercept a web service request before the web service's business logic is executed. The PI monitor examines the parameters for private data and consults its knowledge base, which also includes privacy policies for use of private data, in order to determine whether or not the web service should be permitted. A PI agent is a program that mines log files, containing logs of web service request and replies, in order to gain information on the use of private data by web services and applications and on the flow of private data through the organization's software and DBs. The privacy knowledge base contains the following components:

- PI Elements Schema
- PI Data Mining Agent

- Privacy Policies
- Web Services PI Schema
- Applications PI sub-KB

The PI Elements Schema has information on the PI data elements in the organization's DBs. The PI Data Mining agent mines the log files, verifies and augments the content of the KB. The Web Services KB has details on the web services and what kind of private information they use. The Applications PI contains details on the application and the environment that was used to send the private information such as IP address, web interface, web application, stand-alone application, etc. The PI Expert makes the decision on making changes to the Enforcement/Monitoring Rules by taking appropriate information from the knowledge base.

CHAPTER 3 : SOFTWARE ARCHITECTURE AND REQUIRED INFORMATION

In this chapter we describe our software architecture for providing privacy web services using a typical scenario in which web pages are used for collection of private data in Resource Oriented Architecture **(ROA)**. In this scenario there are web pages that collect Private Information (**PI**) from the subject without showing notice, consent, authentication, and secure transfer. Most web pages, which collect private data, have forms with fields to collect data and invoke web services to store the collected data in organizations' databases. Our objective is to (semi)automatically modify such a web page, which collects private data without privacy service, to (i) show notice, (ii) obtain consent, (iii) authenticate, and (iv) ensure secure transfer of data.

The modification of a web page consists of the following steps: identify the web page; ensure secure data transmission; authenticate; show notice; obtain consent; enter PI; store consent; and store the collected PI. The constraint is that the web service to store the PI, collected through the form, in a DB cannot be modified. The reason is that modifying such web services would entail examining and modifying the code generating the web service, which is a difficult and complex task as, for instance, in Java software environment, there are many classes that, in general, are used in creating web services.

From the perspective of this thesis, webpages can be broadly classified into two types: **HTML** pages, such as pages specified in HTML 5, or **server-script generated**, such as XHTML, PHP or ASPX pages. The distinction is that privacy services can be injected into an HTML, XHTML page by modifying its source code, while modifying source code of a PHP or ASPX pages may be too complex and therefore some other approach needs to be used. Thus, in the following, two architectures are described, one for injecting privacy services by modifying the XHTML / HTML pages, while the other architecture for injecting privacy services for pages generated by other set of server-side scripts such as PHP or ASPX pages. After the two architectures are described, we analyze information required and whether and how that information can be gathered in PAWS. However, before we proceed, we will describe transactions occurring when a form on a

web page is used to collect PI without privacy services, i.e., without notice, consent, authentication, and secure data transfer.

3.1 Web Page Actions without Notice and Consent

Figure 3.1 shows the activities, a set of transactions, that occur without privacy services. The web page is displayed; the subject fills the fields of the form; the subject clicks on the submit button on the form; the script of the submit button invokes the *Store-PI-ws* web service; and *Store-PI-ws* stores the PI in the database using SQL statements. Note that, although the figure shows that *Store-PI-ws* uses SQL statements to directly store PI in a DB, it could actually invoke another web service(s) for this task. However, for simplicity in discussion and figure presentation we show that *Store-PI-ws* directly accessing the DB. Also note the web service, invoked by the script of the submit button to store the collected data in a DB, can have any name/URL; however, for simplicity we refer to it as *Store-PI-ws*.

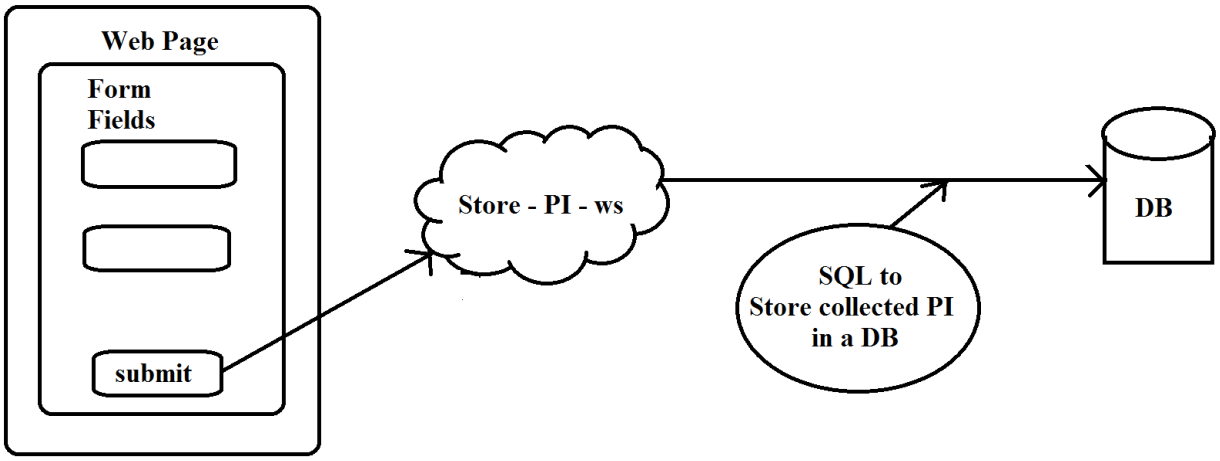

Figure 3.1 Web page and its form to collect PI

3.2 Architecture for Injecting Privacy Services

In this section, we provide an overview of our proposed architecture by describing the components that are provided to enable PE, who would be interacting through the *Controller* module, to inject privacy services into a web page. The *UI* module discussed in this section is the interface that is used by PE to interact with KB's and guide the process

of inserting privacy services into the XHTML page(s). Figure 3.2 shows the architecture for injecting Privacy Services for XHTML pages along with the various components at a higher level of detail.

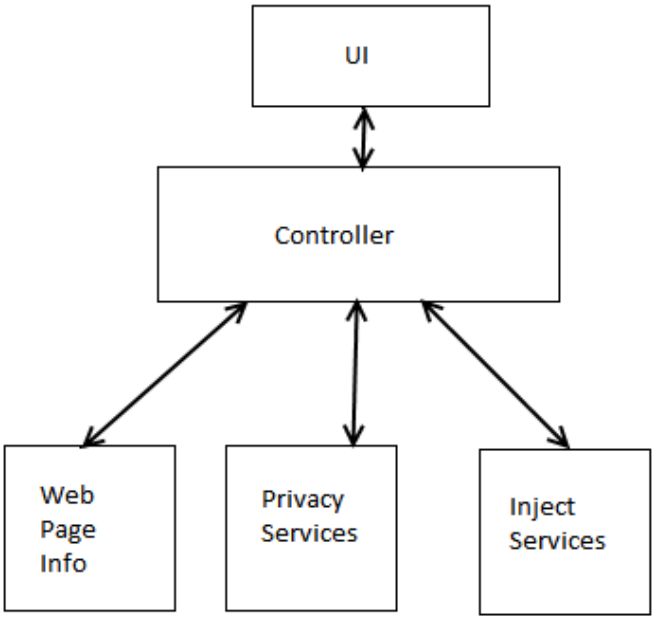

Figure 3.2 Architecture for injecting privacy services in XHTML pages

The *UI* component, interacts with the *Controller* module that interacts with *Web Page Info*, *Privacy Services*, and *Inject Services* components. The *Web Page Info* component is used to provide information on the XHTML page that collects PI from subjects. The *Privacy Services* component provides information on web service(s) – information dealing with storage/retrieval of PI in/from the DB, authentication, secure data transmission, and notice and consent details. The *Inject Services* component performs insertions in the *XHTML* source code to inject privacy services and also update the relevant KB's with details on the injected privacy services.

Tasks Performed by the Architecture

In this section we explain our architecture in detail with the tasks performed by the sub-components that facilitate insertion of privacy services into *XHTML* pages (see Figure 3.3). The PE interacts through the Controller module, which is the interface to various components in the architecture. The *Web Page* module interacts with *XHTML Source*

component to find the source code of the XHTML pages and *Web Page kb* component to find information if the web page collects PI and information on which privacy services, namely notice, consent, authentication and secure data transfer, are required. In our thesis it is assumed that whenever PI is being transferred then secure data transfer is required and provided.

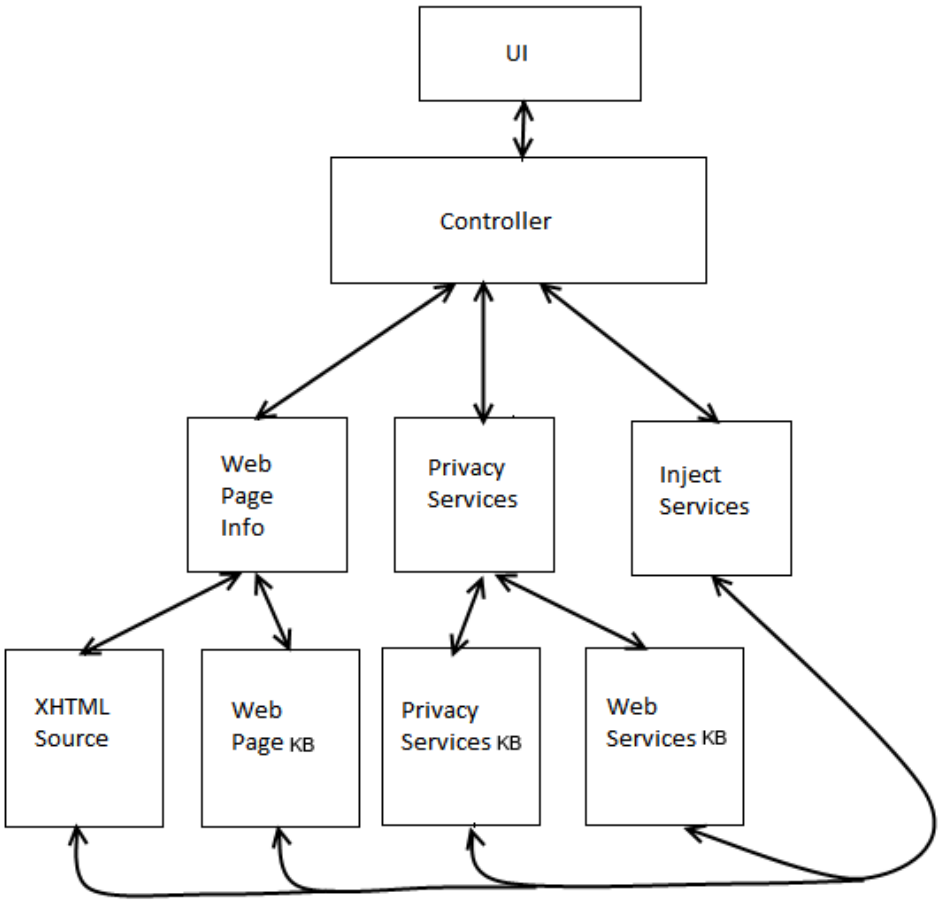

Figure 3.3 Architecture including sources for required information

The *Privacy Services* module interacts with *Privacy Services KB* to find which privacy services of notice, consent, authentication, and secure data transfer are available, and if any of these privacy services were used for the collection of PI. *Privacy Services* module interacts with *Web Service KB,* which has information on the web services that store PI in the database, and also information on the web services that retrieve those PI from the database.

Inject Service component does the injection of the web services into various components. The source code of the web page is modified to include insertion of privacy services. Then, details on injected privacy services in the *XHTML* page(s) are stored in *Privacy Services KB* and in *the Web Pages KB*. The same architecture is applicable to *HTML* pages with minimal changes in the insertion of the source code.

3.3 Architecture for Other Server-side scripts

In this section we provide an architecture for injecting privacy services for other server-side scripts, including, but not limited to, PHP and ASPX. Architecture includes the Monitor agent module that does the insertion of privacy services in the web pages generated by server-side scripts. The Monitor agent is a part of PAWS that intercepts both web page request/replies and web services requests/replies. In our scenario the monitor agent intercepts the response to the PHP/ASPX request, identifies from the *Web Page KB* if it collects PI and dynamically updates the web page response to inject privacy web services. The privacy web services would still require all the information from the KB. The information on *Web Page KB, Privacy Services KB, Web Service KB* components are discussed in the following section.

3.4 Required Information

3.4.1 Web Page KB

Web Page KB contains privacy related information on web pages. It contains information on if the web page collects PI or not and which privacy web services, such as notice, consent, authentication, and secure data transfer, are required in the web pages. In our thesis we assume that a web page collects data from a subject using a form. The following steps are followed to build the KB: *XHTML source component* in our architecture provides access to the source code; submit button of a form is searched for in the source code; invocation of the web service is found to store private data in the DB, directly or indirectly. If the web service contains PI in its parameters and stores it in the DB, then the web page is found to collect PI. Information on the web services is found from the *Web Service KB*.

3.4.2 Privacy Services KB

Privacy Services KB holds information on the privacy policy used to collect PI, which privacy services of notice, consent, authentication and secure data transfer are available, and which are used in the web page for collection of PI. The information on privacy services used in the collection of PI are found by traversing the source code of the web page and looking for key words, such as notice and consent, in the source code that denote privacy services that may already exist in the web pages. The source code is retrieved from the *XHTML Source* component.

3.4.3 Web Service KB

Web Service KB has the details on each web service: whether the input parameter and/or its output contain PI; details on if PI is used to store/ modify/ retrieve PI in a DB; if it invokes other web services; and which applications, web services, or web pages invoke it. The details on web services are found by the PAWS's PI Agent that mines, the logs of both web service requests and replies recorded by PAWS's Monitor.

3.5 Injection of Privacy Services to Show Notice and Obtain Consent

We now provide an overview of modifications to the web page with the injection of notice and consent.

Form Identification and Web Page Information

The web page has a form tag that is used to solicit information that, potentially, includes private data. Once the data is collected by the form, it is delivered to the organization's servers/DBs by invoking a REST web service. The web service, invoked by the form's submit button, stores the form's data, passed as the parameters, in the organisations' database. Once this web service, which stores the collected data, is identified, the PAWS's knowledge base is consulted to determine whether or not the web service's parameters include private data and, if they do, also whether or not notice has been provided and consent has been obtained.

If notice and consent have not been facilitated by a web service with parameters containing private data, modifications are made to the web page. The URL of the web page provides the exact location of the web page on the web server. In some cases, there is a

specific mapping of the file on the web server for each web application, which maps the URL of the web page to the exact location of the web page. These files vary in format and hierarchy depending on the language used and the operating system environment.

Tasks to Inject Notice and Obtain Consent

Once it is determined that the web page collects PI without privacy services, the web page is modified to show notice before the web pages' form is displayed to the user/subject. More specifically, the page is modified with insertion of a script on the form – the script causes invocation of a web service that will display notice that informs the user that private data will be collected by the form. Together with displaying notice, consent is also sought. The consent is displayed as checkboxes on the screen where the user can either select or deselect to either grant or deny permission to the organization on the use of their private data.

The first step is to find the form tag. The form has the *onclick event* that will, when a form's submit button is clicked, be invoked and it, in turn, invokes a web service to store PI in the database. The submit button's functionality (form's submit method) is modified to call the *Notice-Consent-ws*, which performs the following tasks shown graphically in Figure 3.4: (i) finds if the web page displays a form that collects PI from the subject; (ii) queries the *Notice-Consent-kb* to retrieve the appropriate notice and consent to be displayed; and (iii) returns the notice and consent to the browser that displays it to the subject.

The PE decides on the notice and consent that is to be displayed when the form is invoked and stores it in the *Notice-Consent-kb*. The notice is shown as plain text on the web page and consent is displayed as a set of checkboxes, from which the subject selects his/her preferences to share their PI to third party companies for advertisements, marketing, sales, etc.

We have selected the checkbox mechanism to display the consent over opt-in and opt-out as it provides flexibility to subjects regarding the usage of his/her PI. Once the subject consents to the collection of PI, by clicking on the accept button, the script associated with the accept button stores the consent details in the web page as hidden fields.

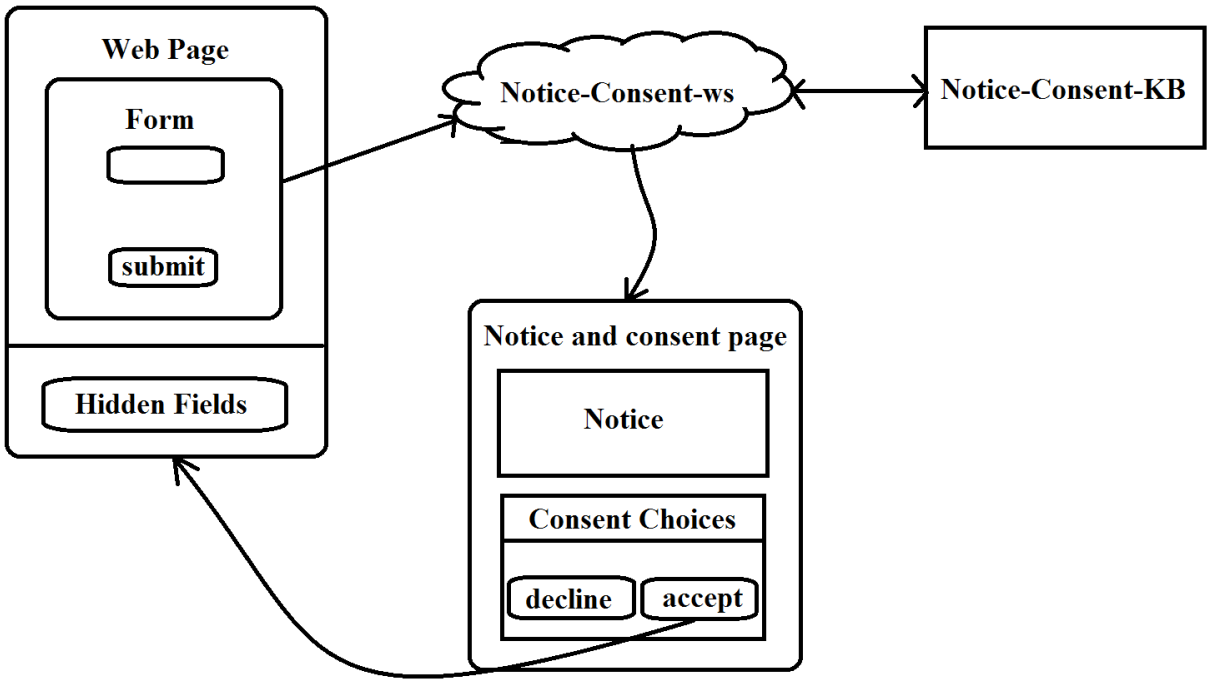

Figure 3.4 Display of Notice and Consent

Enter the PI and Submit

After providing the consent, the subject now can enter data in the form's fields consisting of text boxes, radio buttons, combo boxes, and other form data entry mechanisms. Once the subject enters the fields, he/she clicks on the submit button, which has a script associated with it. The script first validates the data entered by the subject and if there are problems it displays a message and asks the subject for corrections and resubmission. As well, we assume that measures, like CAPTCHA, are taken to prevent forms from being used by robots. A CAPTCHA image shows a random string of number or characters, which the user has to type to submit the form (Cui, et al., 2010).

Store Consent

If the data is successfully validated, the newly inserted script for the submit button invokes a web service *Store-Notice-Consent-ws* that will store information on the notice and consent in the organization's DB. Its parameters are obtained from the hidden fields in which information was stored by the script associated with the accept button of the consent form. The script invoking the *Store-Notice-Consent-ws* is inserted just before the

invocation of the web service that stores the form's data in the DB – web service to be invoked next. Thus, the mechanisms to store the PI in a DB remains the same as no modifications are made to the storage of PI in the DBs.

Figure 3.5 graphically shows the transactions of the form after insertion of the notice and consent. It shows the following actions:

1. Once the subject submits the PI form by clicking on the submit button, the *Store-Notice-Consent-PI-ws* is invoked with parameters retrieved from the form's hidden fileds.
2. The *Store-Notice-Consent-ws* stores the notice and consent choices in the Consent DB.
3. The *Store-PI-ws* stores the PI in the DB.

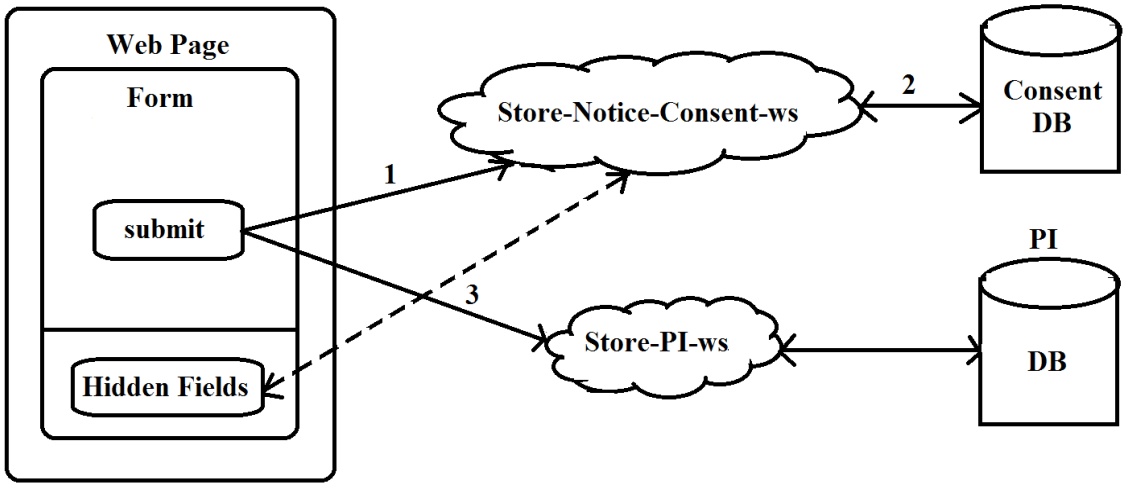

Figure 3.5 Storing Consent and PI

Store PI

If the consent is successfully stored in the database, there is another invocation to store the PI in the database. The form invokes *Store-PI-ws* that will store the PI in the database, with are passed as parameter to *Store-PI-ws*. *Store-PI-ws* is invoked, which stores the PI in the DB through SQL statements or it can actually invoke another web

service(s) for this task. However, for simplicity in discussion and figure presentation we show that *Store-PI-ws* directly accessing the DB to store PI.

3.6 Security Mechanisms for Messages

The World Wide Web (**WWW**) is open to attackers and hence we are forced to adopt security measures for protecting PI and our digital identities. Authentication and secure transfer are the main security mechanisms used for securing data, including PI, when collecting PI from an online subject. In this section we discuss how these mechanisms are used in the context of this thesis. Whenever a web service has PI in its parameters we: (i) ensure secure data transfer and (ii) authenticate the subject if it is required to do so. These security measures are necessary to ensure the confidentiality and integrity of private data when transferred over the network.

When the web page is requested we ensure that the web service uses a secure channel, i.e., that request/reply messages of such a web service are secured, and that we authenticate the subject who invoked the web service. *OAuth* is not used for our prototype as it needs to rely on third party authentication servers for authentication.

3.6.1 SSL for Secure Data Transmission

To ensure secure network transfer, SSL is used. In order to successfully implement and use SSL, a web server holds and uses certificates to identify itself to the client user. This helps the Internet users to trust the web site that publishes web page(s). These certificates can be issued by either third party companies, which can be expensive, or by using "self-signed" certificates, which can be implemented by using JAVA's command-line interface, called keytool. For the prototype in this thesis we use the self-signed certificates.

A form is used to collect PI from the subject. When the web page is loaded, before the subject views the form and the submit button is clicked, the *Secure-ws* is invoked. Figure 3.6 shows graphically the sequence of steps that are followed: (i) *Secure-ws* is invoked when the web page is loaded and, just before the form is loaded; (ii) *Secure-ws,* then displays a certificate which shows that data is transferred with a trusted source ;(iii)

once the subject accepts the certificate, subsequent data transmission, including messages containing web services requests/replies, will be over a secure channel (HTTPS protocol).

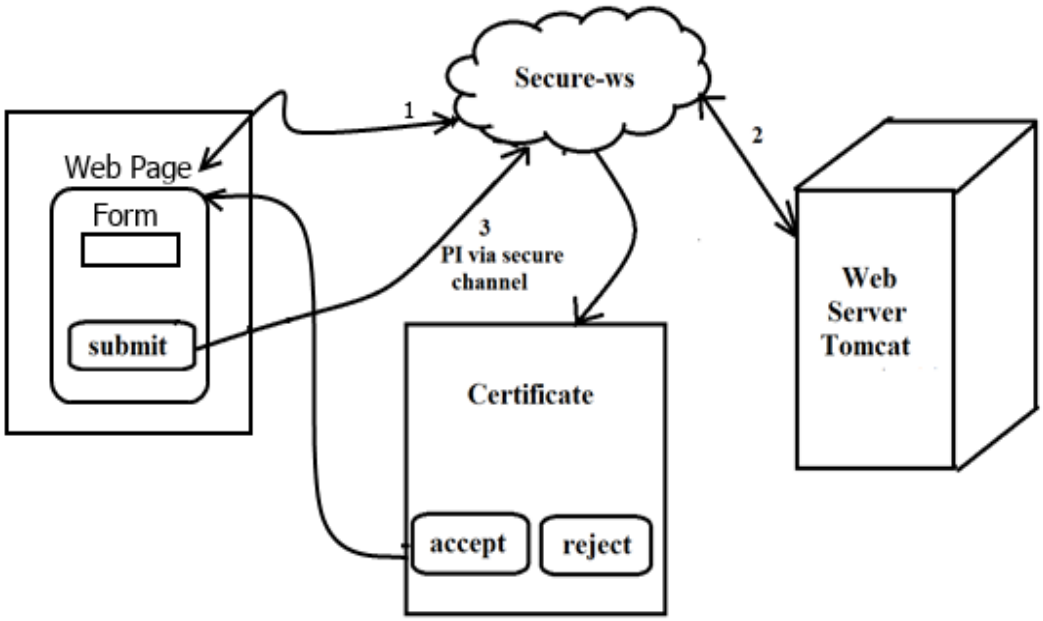

Figure 3.6 Securing web services that use PI

3.6.2 Tomcat Authentication Mechanism

In general, authentication is used to identify a person, device, software component, or any other entity accessing the computer system, often as a prerequisite to allow appropriate access to specific resource in the computer system (Sun, 2000). The authentication method helps to successfully identify the user as a precursor to authorization.

In the context of this thesis, the PE makes the decision if the form needs authentication before the notice and consent are displayed and stores this decision in the *Authenticate-kb*. When the web page collecting private data is invoked, the following steps are followed: (i) *Authenticate-Tomcat-ws*, queries the *Authenticate-kb* to find if authentication is required or not; (ii) if required, a separate authentication page with a form is shown on the browser, to get the authentication credentials (preferably username and password) from the subject; (iii) when the form is submitted, it invokes *Authenticate-Tomcat-ws* to use Tomcat's authentication mechanism to verify the authentication

credentials; (iv) If authentication succeeds, notice and consent are displayed followed by a form that collects PI from the subject.

There are various authentication mechanisms that are available and followed in the industry, but for the purpose of this thesis' prototype, we used the Tomcat's authentication mechanism. Figure 3.7 graphically shows the steps that are followed for authentication. It should be noted that, in order to use some other authentication mechanism, the modifications required are localized to changing the invocation of the web service: for instance, to invoke the OAuth authentication mechanism, instead of invoking *Authenticate-Tomcat-ws*, a web service *Authenticate-OAuth-ws* would be invoked.

Tomcat is the web application server used in this thesis for the PAWS prototype to ensure that only authenticated users are accessing the web service. The configuration is on *tomcat-users.xml* file that contains the list of authenticated users of the web application. There can be roles created on the web application to ensure certain level of authorization on the use of web services.

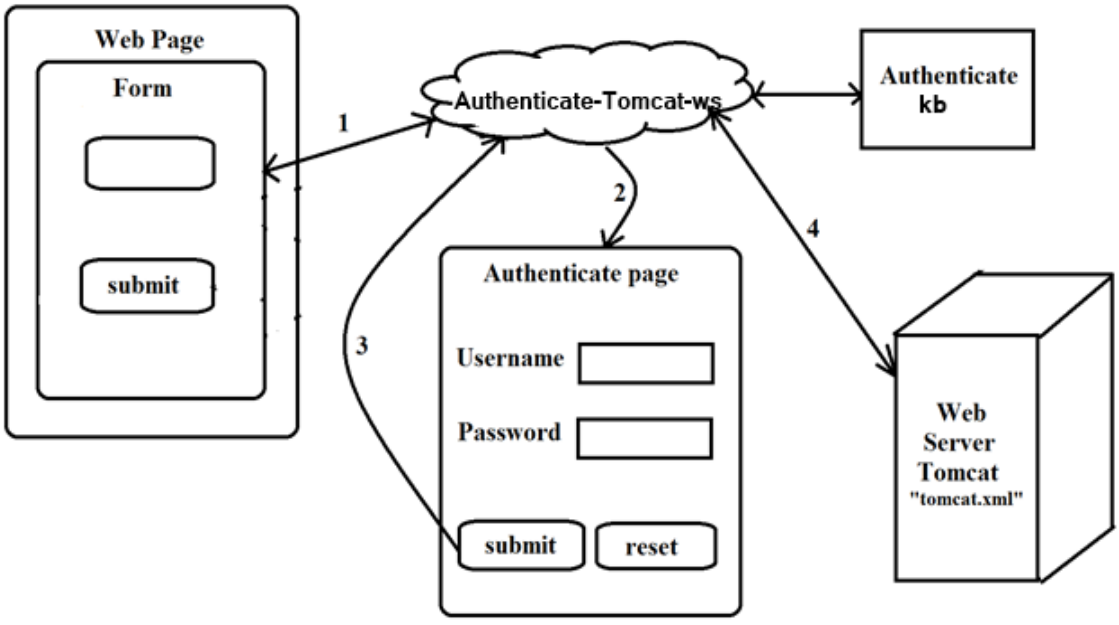

Figure 3.7 Authentication of the Subject

CHAPTER 4 : PROOF-OF-CONCEPT PROTOYPE

In this chapter we describe the prototype implementation. We first briefly describe the tools used to build and test the prototype and then describe the results of inserting privacy services using code snippets and execution snapshots for explanations. Finally, we also describe the PE's interface to inject privacy services into the web pages.

4.1 Web Service Development Tools

Some of the tools described below were used only for testing purposes and creation of snapshots of code and messages exchanged – they are not required for injections of privacy services.

Jersey

In order to create and implement REST web services in JAVA, Jersey is used. Jersey is an open source, production-quality RESTful Web Services framework that is used to create RESTful Web Services in JAVA. In our thesis, we used the stable release of Jersey 2.5.1. Jersey is Oracle's JAX-RS (JSR 311 & JSR 339) implementation.

The steps to configure Jersey Framework are to place all the required jar files for Jersey in the *lib* folder of the web application and to configure the *web.xml* deployment descriptor file. Figure 4.1 shows the configuration required for Jersey in the deployment descriptor. After successful configuration, any URL with (application-name)/api will be routed to the Jersey Framework.

```
<!-- Rest Jersey Mapping -->
<servlet>
    <servlet-name>Jersey REST Service</servlet-name>
    <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>com.sun.jersey.config.property.packages</param-name>
        <param-value>privacyinnovate</param-value>
    </init-param>


</servlet>

<!-- Rest Jersey Mapping -->
<servlet-mapping>
    <servlet-name>Jersey REST Service</servlet-name>
    <url-pattern>/api/*</url-pattern>
</servlet-mapping>
```

Figure 4.1 Required configuration for Jersey in web.xml

jQuery

jQuery is an open source, cross-platform JavaScript library. In our thesis, we use jQuery to create web service requests from the browser. Figure 4.2, shows the script that uses jQuery to request a RESTful web service by sending a GET to a URL *http://localhost:8080/digbyFE/api/v1/user/securews*. The web service does not have any query parameter. Successful execution of the web service will return one parameter, data, that is of type plain text.

```
<script>
    function loadsecurews() {
        jQuery.ajax({
            type : "GET",
            crossDomain : true,
            async : false,
            url : "http://localhost:8080/digbyFE/api/v1/user/securews/",
            dataType : "html",
            success : function(data) {
                // on success diplays the data
                alert(data);

            },

            error : function(xhr, ajaxOptions, thrownError) {
                alert('failuredata');
                console.log(xhr.status);
                console.log(thrownError);
                console.log(xhr.responseText);
                console.log(xhr);
                // error handler
            }
        });

    }
</script>
```

Figure 4.2 Jquery script to create a GET RESTful web service request

Log4j

Log4j is Apache's open source logging API that provides efficient and customizable ways to log both the web service requests and replies. Log4j is used in the web application to create log file on the server by monitoring the web services. Log4j is implemented in various languages. In this thesis we used Log4j's Java implementation. Log4j has three main components: loggers, appenders and layouts. All of these three components are required to create the log file, to provide format for the message type, such as date format, parameterized format, custom format, etc., and to set the level of logger

such as Debug, Info, Warn, Error, and Fatal. *Logger* is a static class that has *getLogger* method which takes a class name as input parameter and returns a Logger that is used to set the level of logging. *Appender* holds the type of logging, such as console or file, and the location of log messages. There are various appenders, such as *ConsoleAppender* that logs the messages unto a console or *RollingFileAppender* that logs the messages unto a file. *Layout* is used to provide the pattern of logging the error messages like data format, etc.

```
#Root logger option
log4j.rootLogger=INFO,file

#Direct log messages to a file
log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=/local/data/tomcat/logs/serverlog.log
log4j.appender.file.MaxFileSize=1MB
log4j.appender.file.MaxBackupIndex=1
log4j.appender.file.layout=org.apache.log4j.PatternLayout
```

Figure 4.3 Log4j.properties file for logging

Figure 4.3 shows the log4j.properties file that is read from the web application to configure the logger. Providing this information in the *.properties* file avoids hard coding in the web application. Thus the type of logger or appender or layout can be changed on the fly without restarting the web application. This provides flexibility to configure the web application.

Advanced REST Client

This is a web browser plugin that is installed as an extension to chrome. Advanced REST Client helps to create RESTful web service requests with custom-header fields and payload with content-type, such as *JSON, xml, form-data*, etc. The plugin allows also to select which request to make, such as GET, PUT, POST, DELETE, etc. When the send button is pressed the plug-in shows the web response with appropriate request and response headers.

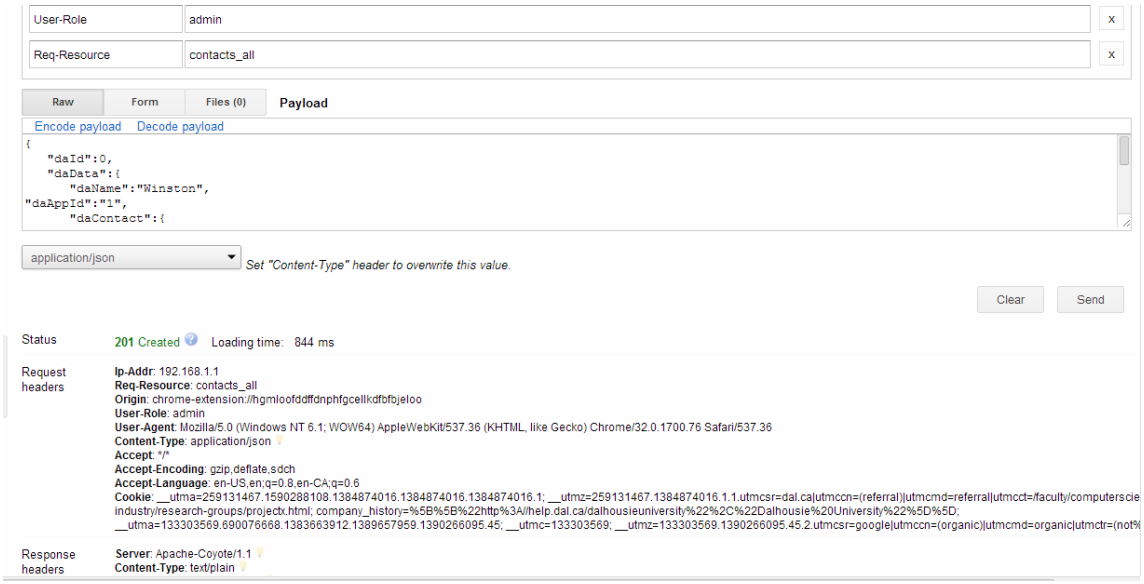

Figure 4.4 Advanced REST Client to create Restful web service requests

Figure 4.4 shows that there was a POST web service request made to store the payload in *JSON* format and the web service response along with its headers are shown. It should be noted that this plug is not required for the implementation of the privacy web services described in this thesis. Rather, it is used for presentation of material in this thesis to capture requests so that they can be presented as snapshots in figures, demonstrating and explaining our method.

Java Server Faces 2.2 (JSF)

JSF simplifies building user interfaces for web applications by reusing components and by use of built-in tags and modules that are readily provided and thus reducing the development time. In our thesis we use primefaces, which is an implementation of JSF 2.2. Primefaces is not tied up with any middleware framework like Spring, Hibernate, etc. Primefaces is UI framework that is helpful in creating rich interface applications.

Integrated Development Environment (IDE)

The web application was developed using Eclipse JUNO IDE. Eclipse is an open source IDE that aids in development of JAVA applications. The IDE has options to configure various frameworks, such as Jersey, and to deploy the web application on a web server. Eclipse provides means to test the application through its interface.

4.2 Proof-Of-Concept on the Injection of Privacy Web Services

We describe in this section the prototype that is implemented as a proof-of-concept of the architecture, which is described in Chapter 3. We start with a description of the web page without privacy services. We then describe the page with injected privacy services by showing the injected code, messages exchanged, and also rendered pages/windows.

4.2.1 Web Page Without Notice and Consent

We created a web page with a form that collects PI. The form collects information from the subject using various input fields like text boxes, radio buttons, check boxes etc. Figure 4.5 shows a web page with a form that collects the student's ID with his/her email address. In our scenario, the student ID and his/her email are considered to be PI. The form is deliberately simple in order to minimize the size of diagrams associated with this page.

The web page is developed using primefaces's wizard module. When the subject clicks on submit button on the form, the submit button's script is executed. It retrieves the data collected by the form's input fields and passes it as parameters to the web service *Store-PI-ws,* which it invokes. Figure 4.5 shows the rendered page with the form asking the subject for the student ID and email address, in which the subject already has filled in the student ID and the email address.

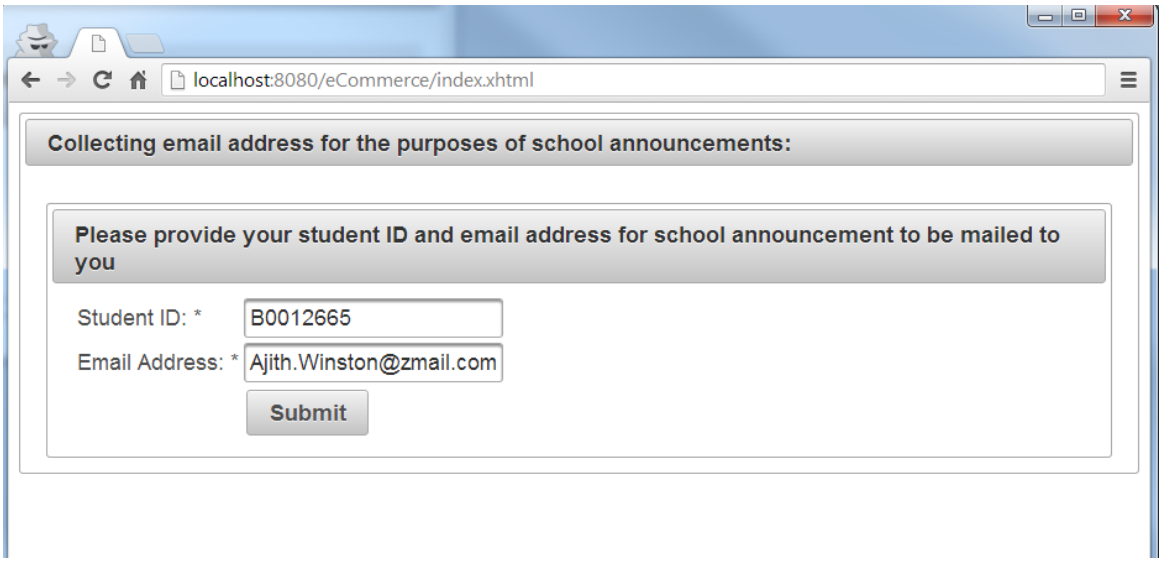

Figure 4.5 Web page that collects PI from the subject

Figure 4.6 shows a portion of the *XHTML* source code for the same page – it shows the source code for the form with input parameters and the submit button's script that invokes *Store-PI-ws*. *Store-PI-ws* is a Restful web service that stores the PI in a DB using an SQL statement. *onclick* event has the script that listens to the *submit* action. Once the subject clicks the submit button, *storePI()* function is invoked..

```
<h:panelGrid columns="2" columnClasses="label, value"
    styleClass="grid" name="status">
    <h:outputText value="Student ID: *" />
    <p:inputText id="studentid" required="true" label="studentid"
        value="#{userWizard.user.studentid}" />

    <h:outputText value="Email Address: *" />
    <p:inputText id="emailaddress" value="#{userWizard.user.emailaddress}"
        requiredMessage="Please enter your email address."
        validatorMessage="Invalid email format">
        <f:validateRegex
            pattern="^[_A-Za-z0-9-\+]+(\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+(\.[A-Za-z0-9]+)*(\.[A-Za-z]{2,})$" />
    </p:inputText>
    <!-- Submit button to store consent and PI on the database -->
    <p:commandButton value="Submit" update="growl"  onclick="storePI()" />
</h:panelGrid>
```

Figure 4.6 Form to collect PI with the script invoked on click

Figure 4.7 shows *storePI()* function, which is a jQuery script to create an ajax request for a RESTful web service using URL *http://localhost:8080/digbyFE/api/v1/user/storepiws/*. Note that in this thesis we refer to this web service as *Store-PI-ws*. Furthermore, the ajax request uses a JSON parameter to send the form's parameters, the student ID and the email address, and receives a response with status code 201 to indicate successful storage of PI in the DB.

```
<script>
    function storePI() {

        var email = document.getElementById('myForm:emailaddress').value;
        var id = document.getElementById('myForm:studentid').value;

        jQuery.ajax({
            type : "POST",
            crossDomain : true,
            async : false,
            url : "http://localhost:8080/digbyFE/api/v1/user/storepiws/",
            contentType : "application/json",
            data : JSON.stringify({
                "bannerId" : id,
                "emailAddress" : email
            }),
            dataType : 'json',
            success : function(data) {
                // on success diplays the data
                alert(data);

            },

            error : function(xhr, ajaxOptions, thrownError) {
                alert('failuredata');
                console.log(xhr.status);
                console.log(thrownError);
                console.log(xhr.responseText);
                console.log(xhr);
                // error handler
            }
        });
    }
```

Figure 4.7 Submit script on the form that invokes Store-PI-ws to store the PI in the DB

Figure 4.8 shows the actual messages that are transferred when invoking *Store-PI-ws*, which stores the PI in the database, and receiving its reply.

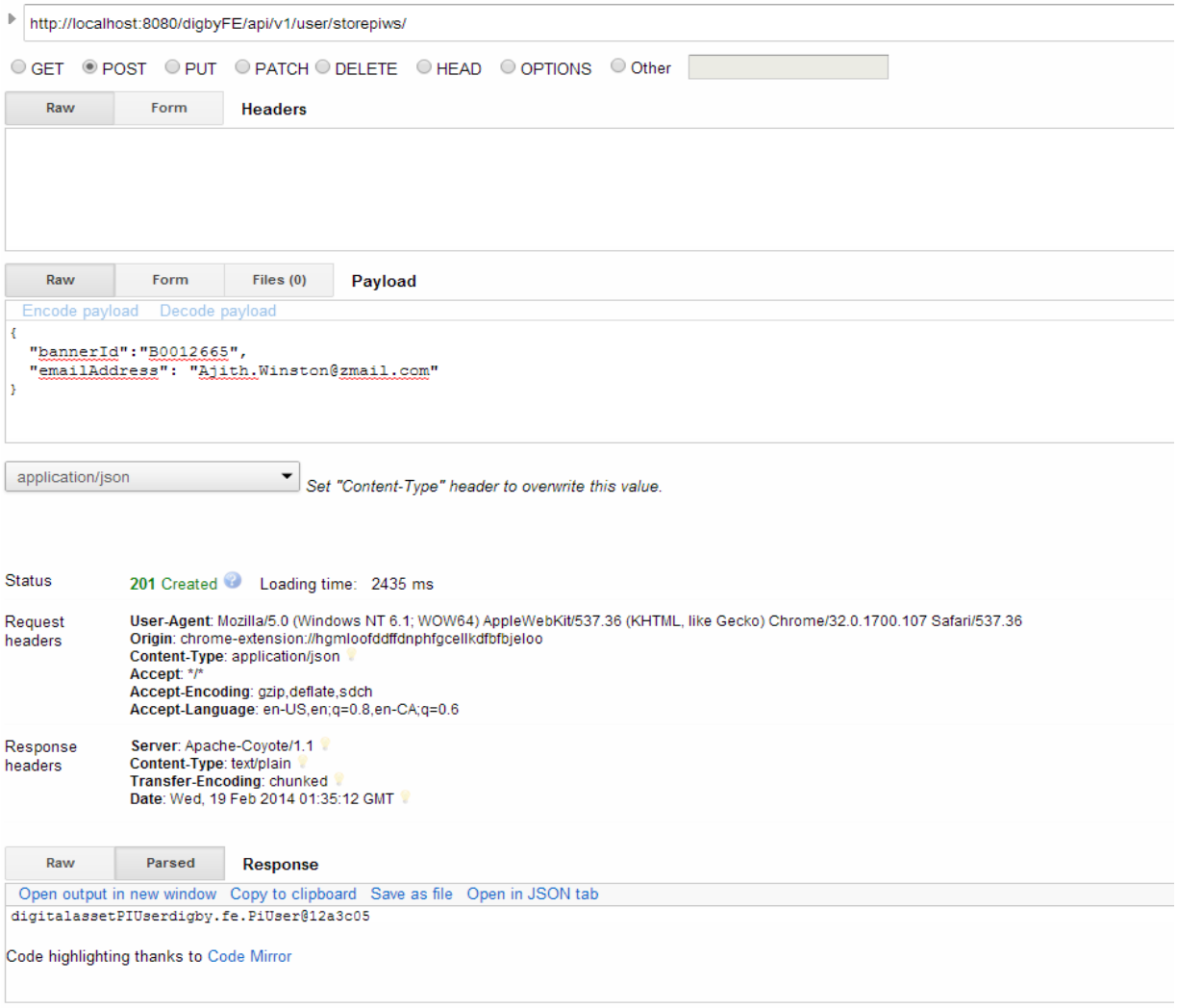

Figure 4.8 Request/reply message for web service Store-PI-ws to store PI in the DB

Figure 4.8 shows the URL to invoke *Store-PI-ws,* http://localhost:8080/digbyFE/api/v1/user/storepiws/, which accepts a POST request. The input parameter is of type "application/json" and is shown in the figure next as payload, where *bannerId, emailAddress* in the JSON request matches the input parameters Student Id, Email Address in the form in Figure 4.5. On successful storage of PI in the DB, the response from the web service is *201 Created*, which is shown in the *Status* section of Figure 4.6.

4.2.2 Secure transmission, Authentication, Notice, and Consent

Overview

Privacy web services are injected into the web page. Figure 4.9 shows that when the web page loads, the *onload event* triggers a web service call to *Secure-ws*. *Secure-ws* establishes secure connection between the web page and the server. *Secure-ws* is a GET request, to show the certificate and to create a SSL connection. We assume the subject accepts the certificate.

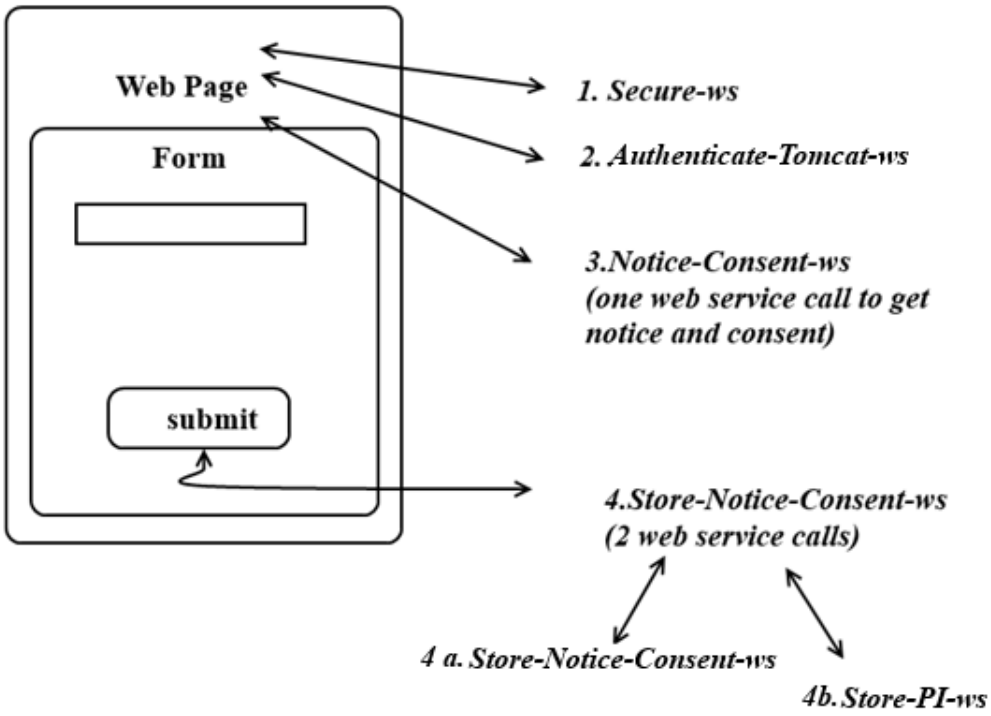

Figure 4.9 Overview of steps involved in the injected privacy services

Then, we assume that authentication is required and hence there is a web service call to *Authenticate-Tomcat-ws,* which is an injected call to use Tomcat's authentication mechanism. When the user submits the button, the credentials are passed to *Authenticate-Tomcat-ws* using the script for the submit button. We assume that the subject passes authentication.

The *onload event* triggers yet another script that invokes a call to *Notice-Consent-ws.* As with the previous invocations of Secure-ws and *Authenticate-Tomcat-ws*, this invocation is injected, such that notice-id and consent-id are included as parameters. The *Notice-Consent-ws*, makes one web service call to *Notice-Consent-ws* to get notice and consent details. The notice and consent, retrieved form the *Notice-Consent-kb,* are returned in the reply. Once the reply reaches the browser, the notice and consent are displayed on the web page using *<p:panel>* tag.

Then, after the subject fills-out the form and clicks on the form's submit button, injected script is invoked, which takes the consent choices from the form's hidden fields and sends them, together with the form's data, as the web service request parameters. Then script then makes two web services calls, one to *Store-Notice-Consent-ws,* which stores notice and consent in the database, while the other to the web service *Store-PI-ws* to store PI in the database. We now discuss individual steps in more details.

Secure Data Transfer

Recall that secure data transfer is obtained by invoking the *Secure-ws* web service that establishes SSL, developed using Jersey's framework. In our thesis, we assume that if the form is found to collect PI from the subject, *Secure-ws* is mandatory. Thus, Secure-ws is invoked before authentication. Secure transfer requires a certificate for the server and, for the prototype of our thesis, we use a simple self-signed certificate, as it is free. Furthermore, we also show it to the subject for approval. The self-signed certificate, shown in Figure 4.10, was created using *keytool*. Once the user accepts the certificate, any data transfer to/from the web page, is over a secure channel and hence encrypted. Figure 4.11 shows the SSL connection that is established on the web page.

```
S:\app\AjithWinston\product\11.2.0\dbhome_1\jre\1.5.0\bin>keytool -list -keystor
1.5.0\bin\keystore.key
Enter keystore password:

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry

root, 10-Feb-2014, trustedCertEntry,
Certificate fingerprint (MD5): E0:19:F5:FC:C0:9A:13:0E:38:B7:BF:0D:02:40:D3:C2

S:\app\AjithWinston\product\11.2.0\dbhome_1\jre\1.5.0\bin>
```

Figure 4.10 Self-signed certificate created using keytool

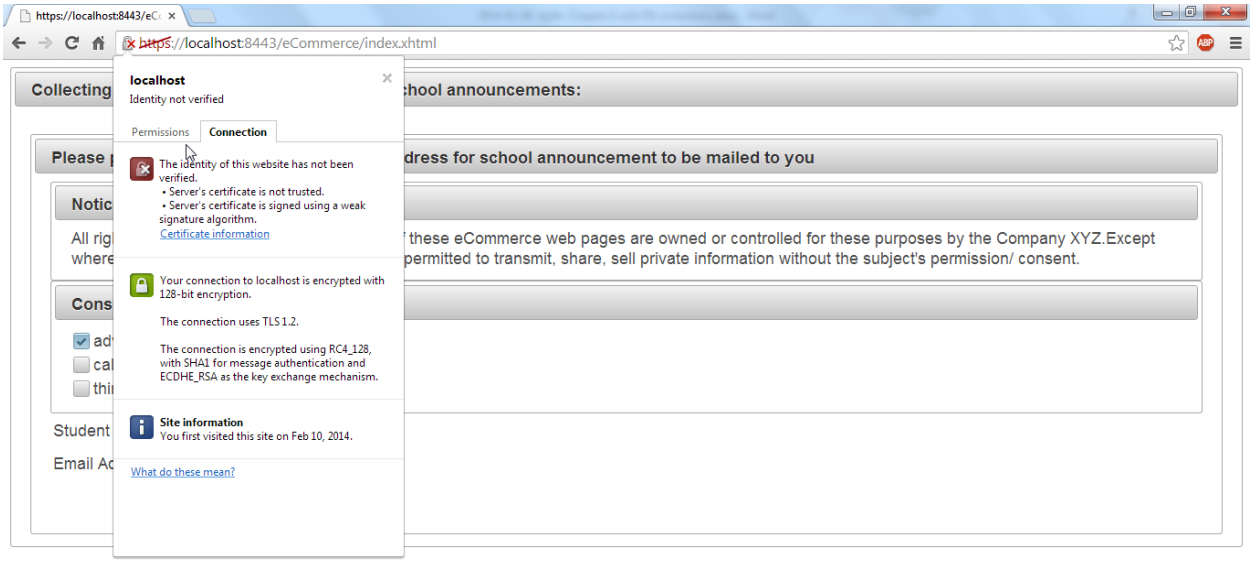

Figure 4.11 SSL connection through Secure-ws

Figure 4.12, shows the injected code in the body *onload* event to create a web service call to *Secure-ws* using the function *loadsecurews()*. Figure 4.13 shows *loadsecurews()* function, which is a jQuery script to create an ajax call to a RESTful web service using an URL http://localhost:8080/digbyFE/api/v1/user/securews/. In this thesis, we refer to the web service as *Secure-ws.* As a result *a GET request* is sent, the web services is executed, and a response message is received indicating "SSL established".

```
<h:body onload="loadsecurews()">

<h:form id="myForm" >
<p:growl id="growl" sticky="true" showDetail="true" />
```

Figure 4.12 Body load invokes a Secure-ws web service call

```
<script>
    function loadsecurews() {
        jQuery.ajax({
            type : "GET",
            crossDomain : true,
            async : false,
            url : "http://localhost:8080/digbyFE/api/v1/user/securews/",
            dataType : "html",
            success : function(data) {
                // on success diplays the data
                alert(data);

            },

            error : function(xhr, ajaxOptions, thrownError) {
                alert('failuredata');
                console.log(xhr.status);
                console.log(thrownError);
                console.log(xhr.responseText);
                console.log(xhr);
                // error handler
            }
        });

    }
</script>
```

Figure 4.13 Script invoking Secure-ws web service

Figure 4.14 shows the web service call to *Secure-ws*, to establish SSL connection. Figure 4.14, shows a GET request to an URL, *http://localhost:8080/digbyFE/api/v1/user/securews/*. Upon successful SSL connection establishment, *SSL connection established* is the response that is sent from Secure-ws shown in the response section in Figure 4.14.

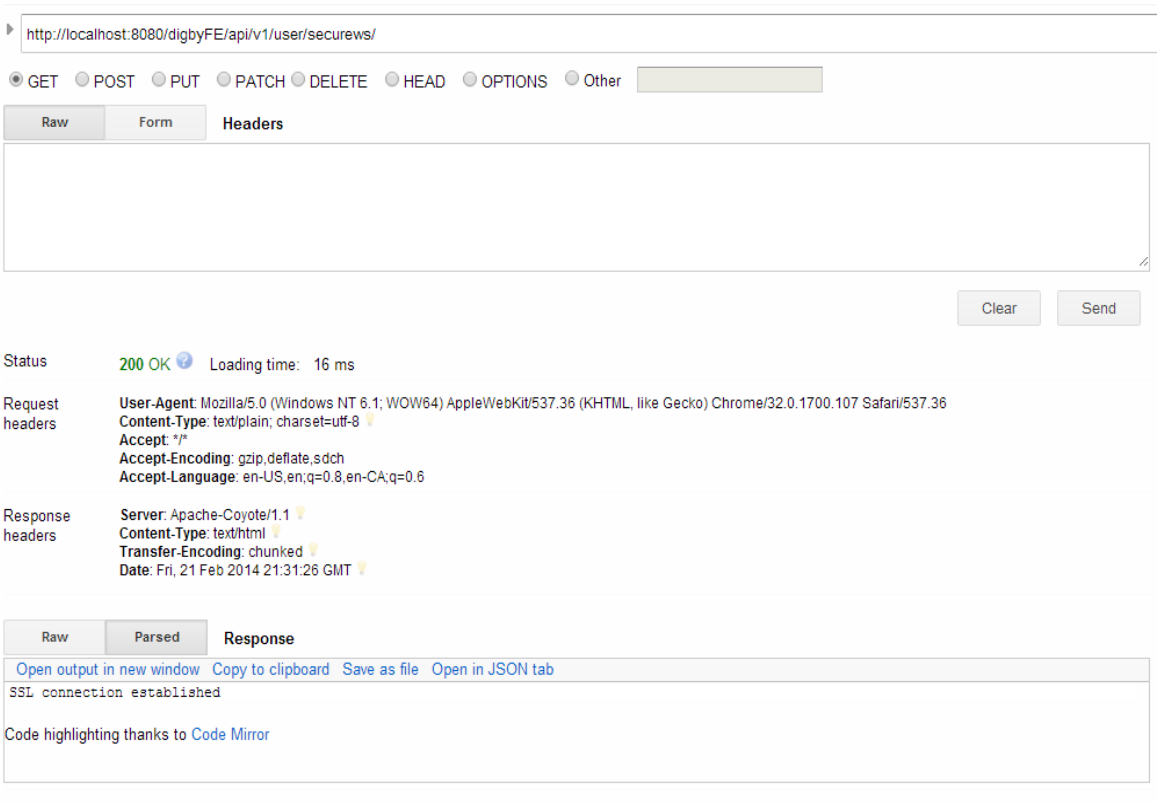

Figure 4.14 Messages transmitted in a web service call to establish SSL

Authentication

When the subject opens the web page, after establishing a secure channel, the second step is authentication of the subject. Authentication would not be injected by PE in a situation when the web page is collecting PI from the subject for the first time. Authentication is achieved by invoking the web service *Authenticate-Tomcat-ws*. As a result of invoking the *Authenticate-Tomcat-ws* web service, a form is displayed to collect authentication details from the subject as shown in Figure 4.15. When the subject fills the authentication details and clicks on the *Login* button, Tomcat's authentication mechanism is invoked. (Note that the Tomcats' authentication mechanism uses a "Login" button that we refer to as Authenticate button.) The mechanism has *tomcat-users.xml* file that holds the authentication details of permitted users. Once the subject is authenticated, the form that collects PI is displayed. If authentication fails, an error message is shown informing the subject about the problem with authentication. *Authenticate-Tomcat-ws* is a RESTful web service that is created to authenticate subjects.

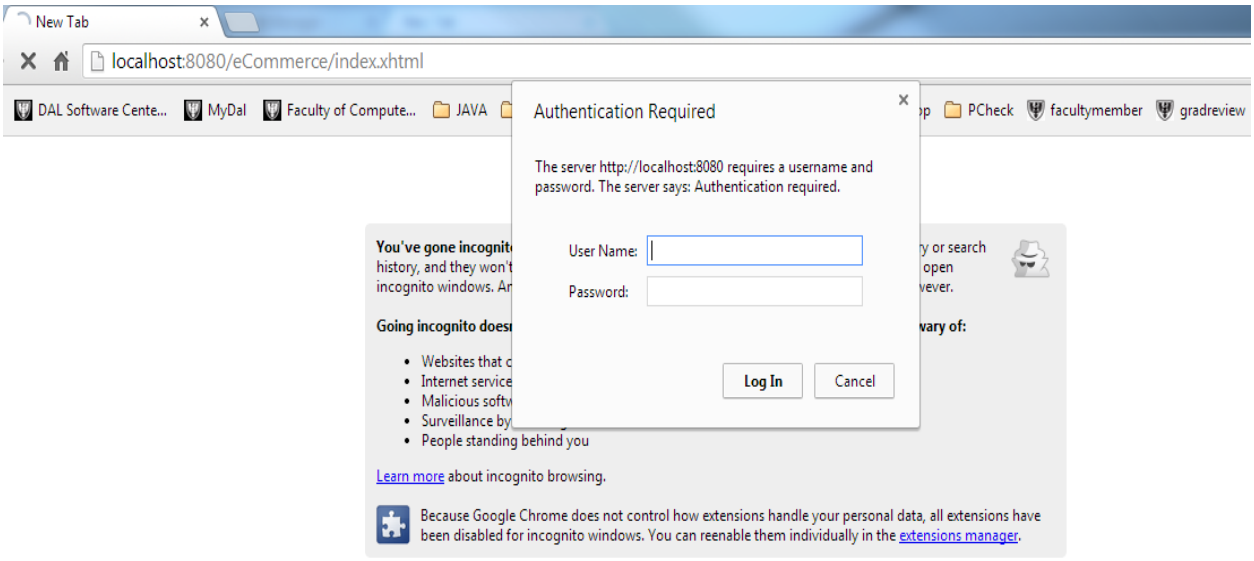

Figure 4.15 Invoking Authenticate-Tomcat-ws that collects credentials from the subject

Figure 4.16 shows the portion of the web page source containing the authentication form, and on submit creates a web service call to *Authenticate-Tomcat-ws* using the function *authenticate()*. Figure 4.17 shows the *authenticate()* function, which is a jQuery script to create an ajax call to a RESTful web service request : *Authenticate-Tomcat-ws,* using URL *https://localhost:8443/digbyFE/api/v1/user/authenticatews/, sends a JSON request parameter to send the form's credentials*. Then, *Authenticate-Tomcat-ws* queries *Authenticate-kb* to preform authentication on the web page.

```
<p:panel header="Authentication Required">
        <h:outputText
            value="#{authenticationBean.authenicationinfo.authenticationDetails}" />
        <h:inputText id="username" name="User Name:"
            value=#{authenticationBean.authenticationUsername} />
        <h:inputText id="passwd" name="User Name:"
            value=#{authenticationBean.password} />
        <p:commandButton value="Log In" update="growl" />
        <p:commandButton value="Reset" update="growl"
            onclick="loadauthenticatetomcatws()" />
    </p:panel>
```

Figure 4.16 Injected call to Authenticate-Tomcat-ws

```
<script>
    function authenticate() {

        var username = document.getElementById('myForm:username').value;
        var password = document.getElementById('myForm:passwd').value;

        jQuery
                .ajax({
                    type : "POST",
                    crossDomain : true,
                    async : false,
                    url : "https://localhost:8443/digbyFE/api/v1/user/authenticatetomcatws/",
                    contentType : "application/json",
                    data : JSON.stringify({
                        "username" : username,
                        "passwd" : password
                    }),
                    dataType : 'json',
                });
    }

</script>
```

Figure 4.17 Script creating a POST request to Authenticate-Tomcat-ws

Figure 4.18 show the documentation that is created to help developers to invoke the *Authenticate-Tomcat-ws* that is created for the prototype. Similar to this documentation there are web pages created for all the privacy web services that are created in this prototype.

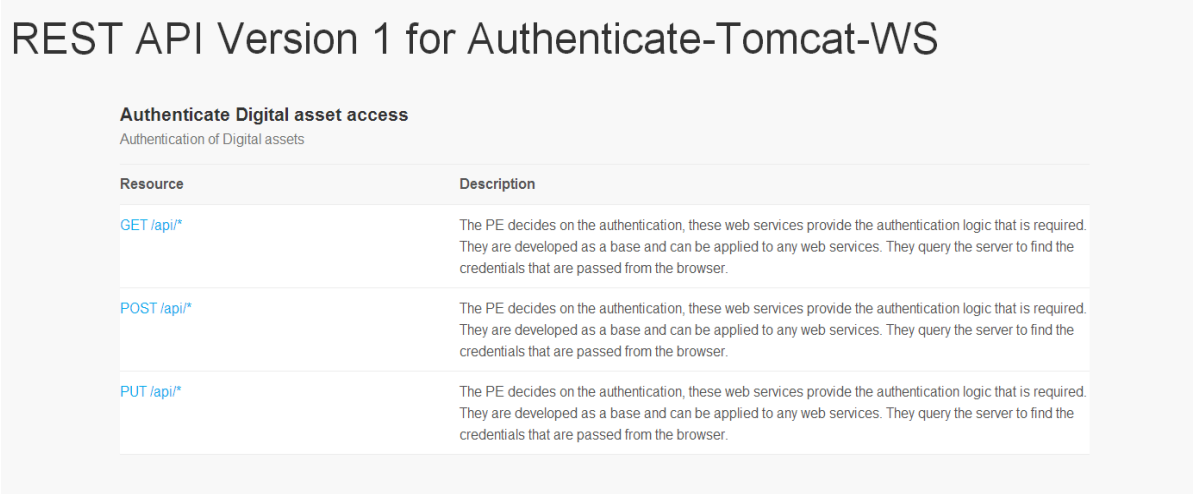

Figure 4.18 Documentation for the Authenticate-Tomcat-ws web service

Authenticate-Tomcat-ws is an a priori developed web service that takes the *user-name* and *password* as parameters from the form and are sent as web service request header fields. The *username* and *password* are encoded using Base-64 encryption, encryption method we have chosen from a number of available encryption methods. When the subject submits the authentication form, the authentication logic is invoked. Figure 4.19 shows the messages that are transmitted to the *Authenticate-Tomcat-ws* to authenticate the user along with the header fields and Base64 encoding that is used in by *Authenticate-Tomcat-ws*.

Header Name: *host*, Header Value: *localhost:8080*
Header Name: *connection*, Header Value: *keep-alive*
Header Name: *authorization*, Header Value: *Basic YWppdGg6YWppdGg=*
Header Name: *accept*, Header Value: *text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8*
Header Name: *user-agent*, Header Value: *Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/32.0.1700.107 Safari/537.36*
Header Name: *accept-encoding*, Header Value: *gzip,deflate,sdch*
Header Name: *accept-language*, Header Value: *en-US,en;q=0.8,en-CA;q=0.6*

Base64-encoded Authorization Value: *YWppdGg6YWppdGg=*
Base64-decoded Authorization Value: *ajith:ajith*

Figure 4.19 Base64 encoding that is performed on the username and password

Notice and Consent

When the subject opens the web page that collects PI, the web service *Notice-Consent-ws* is invoked to show notice and consent by using the notice-id and consent-id. The notice and consent are displayed on the web page using <*p:panel*> tags. The notice

and consent are shown in Figure 4.20 as rendered by the browser. Thus, both notice and consent are handled by one injected web service.

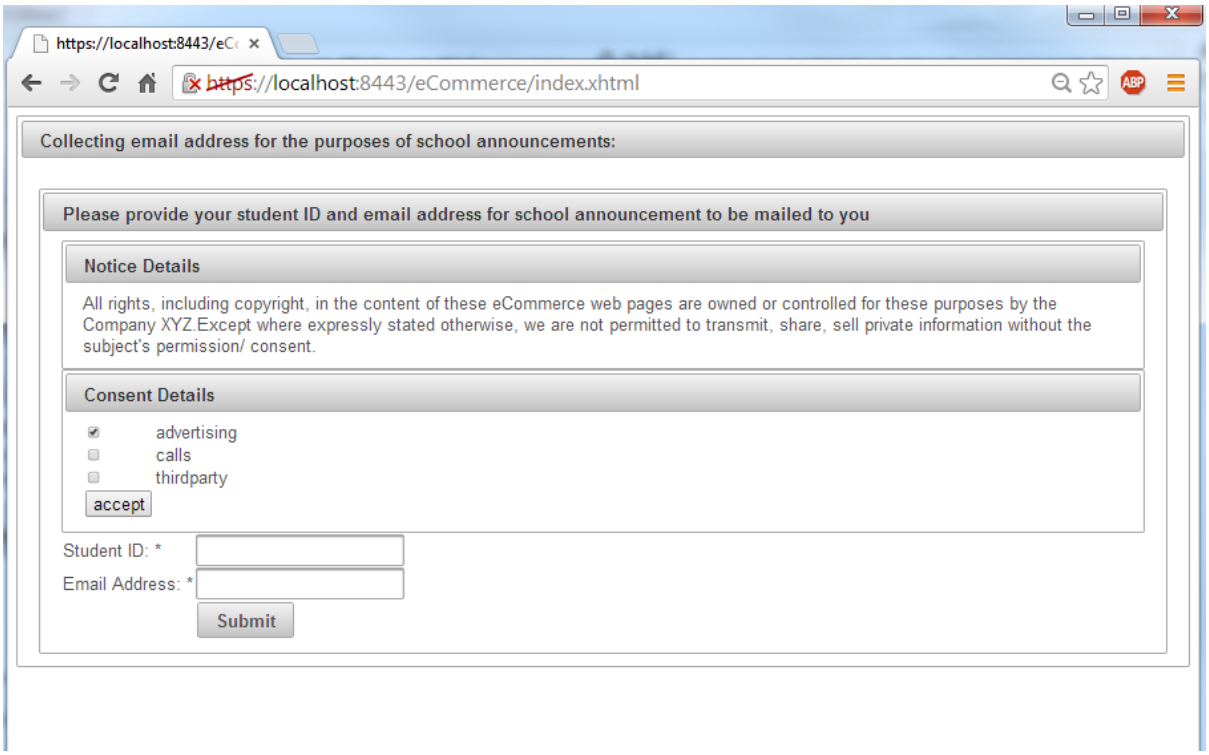

Figure 4.20 Web page showing notice and consent details

Figure 4.21, shows the injected code in the body *onload* event to create a web service call to *Notice-Consent-ws* using the function *loadnoticeconsentws()*. Figure 4.22 shows *loadnoticeconsentws()* function, which is a jQuery script to create an ajax call to a RESTful web service request :*Notice-Consent-ws,* using URL *https://localhost:8443/digbyFE/api/v1/user/noticeconsentws/*. This results in *a GET request with noticeid* and *consent id* as query parameters and a reception of a response of datatype JSON, which has the notice information and consent information to be displayed on the web page. The script dynamically updates the panel to show notice and consent using the script shown in the *success* section.

```
<h:body onload="loadsecurews(); loadauthenticatews(); loadnoticeconsentws();">


    <h:form id="myForm">
        <p:growl id="growl" sticky="true" showDetail="true" />
```

Figure 4.21 Onload event to call jquery script

```
<script>
function loadnoticeconsentws() {
    jQuery.ajax({
        type : "GET",
        crossDomain : true,
        async : false,
        url : "https://localhost:8443/digbyFE/api/v1/user/noticeconsentws/?noticeid=1&amp;consentid=1",
        dataType : "json",
        success : function(data) {
            // on success diplays the data
            alert(data.noticeDetails);
            document.getElementById('myForm:notice').innerHTML=data.noticeDetails;

            document.getElementById('myForm:consentParam1').innerHTML=data.consentParam1;
            document.getElementById('myForm:consentParam2').innerHTML=data.consentParam2;
            document.getElementById('myForm:consentParam3').innerHTML=data.consentParam3;
            if(data.consentValue1=="Yes")
            {document.getElementById('myForm:consentBooleanValue1').checked =true;}
            if(data.consentValue2=="Yes")
            {document.getElementById('myForm:consentBooleanValue2').checked =true;}
            if(data.consentValue3=="Yes")
            {document.getElementById('myForm:consentBooleanValue3').checked =true;}

        },

        error : function(xhr, ajaxOptions, thrownError) {
            alert('failuredata');
            console.log(xhr.status);
            console.log(thrownError);
            console.log(xhr.responseText);
            console.log(xhr);
            // error handler
        }
    });


}
</script>
```

Figure 4.22 Script to show notice and consent using Notice-Consent-ws

Figure 4.23 shows the actual message that is sent to *Notice-Consent-ws*, returning the notice and consent, by taking the *notice-id, consent-id* as input query parameter. This web service call is required to retrieve the notice details and consent choices from the *Notice-Consent-kb.* The response of the web service, is of content-type, *application/json,* and has parameters *noticeDetails* which is the notice displayed as plain text on the rendered

web page, consentParam1, consentParam2, etc., which is the consent choices displayed as checkboxes on the rendered web page.

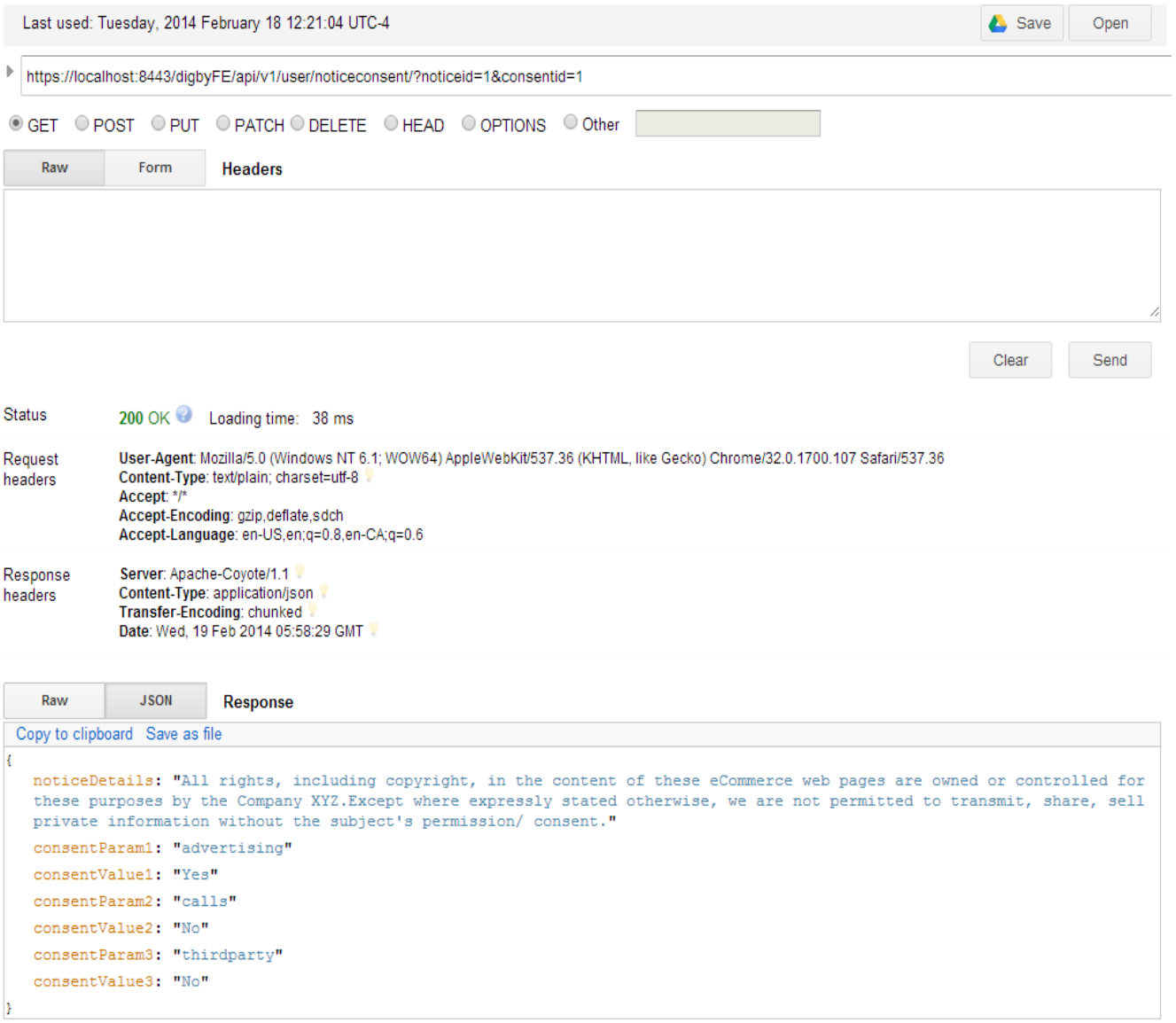

Figure 4.23 Messages transferred to and from Notice-consent-ws

When the subject clicks on the accept button on the notice-consent page (Figure 4.20), the script of the accept button calls the *storeInHidden()* function, which stores the selected consent input in hidden fields and makes the notice/consent frame to disappear (Figure 4.24).

```
<script>
        function storeInHidden() {

            var username = document.getElementById('myForm:studentid').value;

            var consentparam1 = document.getElementById('myForm:consentParam1').innerHTML;
            var consentval1=document.getElementById('myForm:consentBooleanValue1').checked;

            var consentparam2 = document.getElementById('myForm:consentParam2').innerHTML;
            var consentval2=document.getElementById('myForm:consentBooleanValue2').checked;

            var consentparam3 = document.getElementById('myForm:consentParam3').innerHTML;
            var consentval3=document.getElementById('myForm:consentBooleanValue3').checked;
            document.getElementById('myForm:storehidden').style.visibility='hidden';
            document.getElementById('myForm:consentId').style.visibility='hidden';
            document.getElementById('myForm:noticeId').style.visibility='hidden';
        }
</script>
```

Figure 4.24 Script to store consent in hidden fields

Storing Obtained Consent

When the subject clicks on the submit button on the notice-consent page (Figure 4.25), the script of the submit button calls the *Store-Notice-Consent-ws* web service. *Store-Notice-Consent-ws* is a Restful web service that stores the notice and consent choices in a DB using an SQL statement.

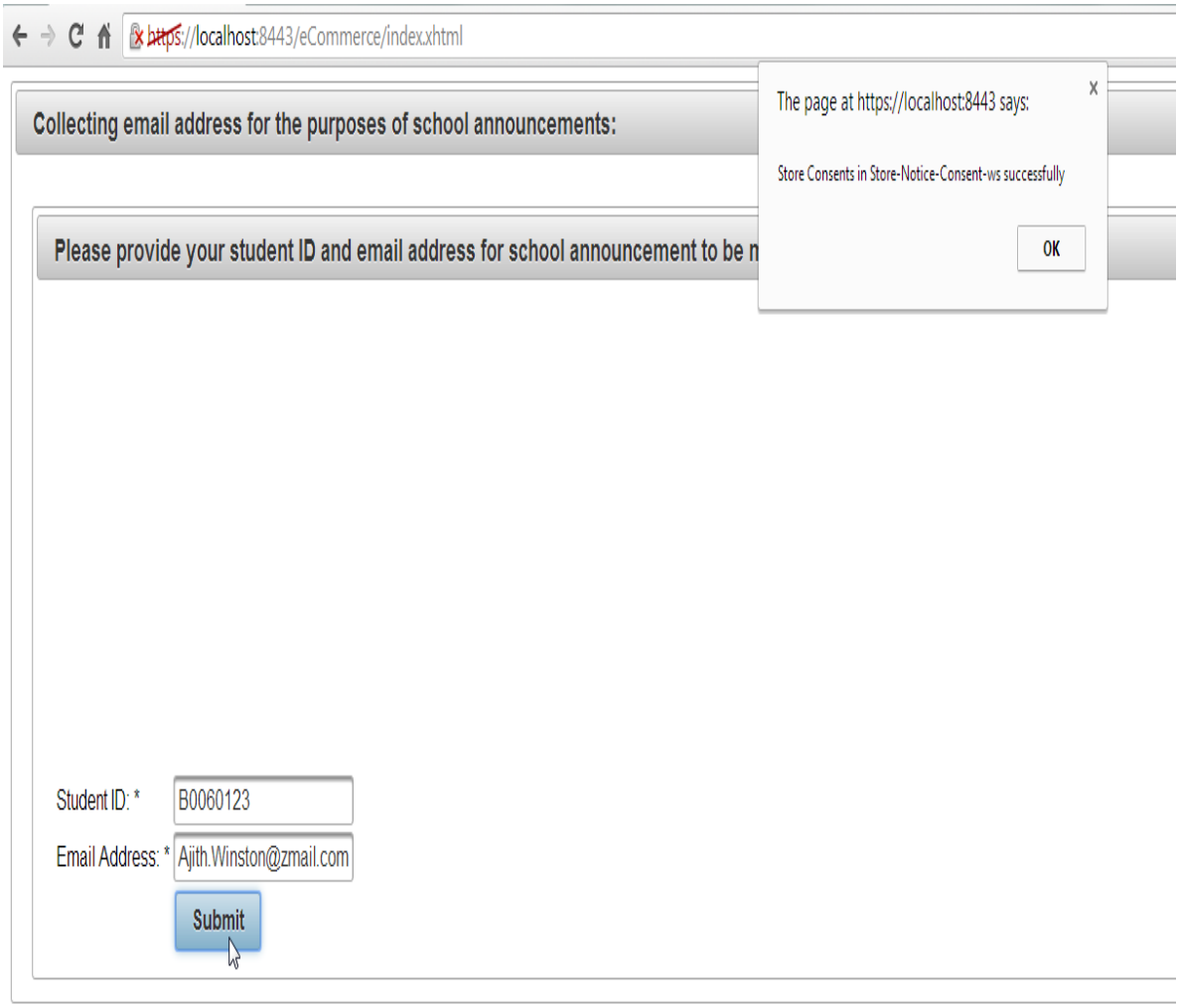

Figure 4.25 Web page after the user clicks on submit button

Figure 4.26, shows a portion of the *XHTML* source code for the same page – it shows the source code of the form with *<p:panel id="noticeOnly">* to display the notice, *<p:panel id="consentOnly">* to display the consent, input parameters to collect PI and the submit button's script which invokes *storeConsentPI()* function.

```
<!-- Injected code for notice and consent -->
                <p:outputPanel id="noticeId" rendered="true">


                    <p:panel id="noticeOnly" rendered="true" header="Notice Details">
                        <h:outputText id="notice"
                            value="#{noticeConsentBean.noticeInfo.noticeDetails}"/>
                    </p:panel>
                </p:outputPanel>

                <p:outputPanel id="consentId">
                    <p:panel id="consentOnly"
                        rendered="true"
                        header="Consent Details">
                        <h:panelGrid columns="2" id="display">

                            <h:selectBooleanCheckbox id="consentBooleanValue1"
                                value="false" widgetVar="consentBooleanValue1">
                            </h:selectBooleanCheckbox>
                            <h:outputText id="consentParam1"
                                value="#{noticeConsentBean.consentInfo.consentParam1}" />
                            <h:selectBooleanCheckbox id="consentBooleanValue2"
                                value="false" widgetVar="consentBooleanValue2">
                            </h:selectBooleanCheckbox>
                            <h:outputText id="consentParam2"
                                value="#{noticeConsentBean.consentInfo.consentParam2}" />
                            <h:selectBooleanCheckbox id="consentBooleanValue3"
                                value="false" widgetVar="consentBooleanValue3">
                            </h:selectBooleanCheckbox>
                            <h:outputText id="consentParam3"
                                value="#{noticeConsentBean.consentInfo.consentParam3}" />
                            <h:commandButton value="accept" id="storehidden" type="button" onclick="storeInHidden()" ></h:commandButton>
                        </h:panelGrid>
                    </p:panel>
                </p:outputPanel>


                <h:panelGrid columns="2" columnClasses="label, value"
                    styleClass="grid" name="status">
                    <h:outputText value="Student ID: *" />
                    <p:inputText id="studentid" required="true" label="studentid"
                        value="#{userWizard.user.studentid}"  />

                    <h:outputText value="Email Address: *" />
                    <p:inputText id="emailaddress"
                        value="#{userWizard.user.emailaddress}"
                        requiredMessage="Please enter your email address."
                        validatorMessage="Invalid email format">
                        <f:validateRegex
                            pattern="^[_A-Za-z0-9-\+]+(\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+(\.[A-Za-z0-9]+)*(\.[A-Za-z]{2,})$" />
                    </p:inputText>
                    <!-- Submit button to store consent and PI on the database -->
                    <p:commandButton value="Submit" update="growl" onclick="storeNoticeConsentws()" />
                </h:panelGrid>

            </p:panel>
        </p:tab>
```

Figure 4.26 Submit script invoked to store notice, consent and private data

Figure 4.27 shows *storeNoticeConsentws()* function, which is a jQuery script to create an ajax call to a RESTful web service POST request: *Store-Notice-Consent-PI-ws* using URL *https://localhost:8080/digbyFE/api/v1/user/storenoticeconsentws/*. It creates a JSON request parameter and, on successful storage of PI in the DB, receives a response with status code *201*. The JSON object is created from the form's parameters, such as *consentparam1, consentBooleanValue1*, etc., which are consent choices the subject has made that are stored in hidden fields.

```
<script>
function storeNoticeConsentws() {
    var username = document.getElementById('myForm:studentid').value;

    var consentparam1 = document.getElementById('myForm:consentParam1').innerHTML;
    var consentval1=document.getElementById('myForm:consentBooleanValue1').checked;

    var consentparam2 = document.getElementById('myForm:consentParam2').innerHTML;
    var consentval2=document.getElementById('myForm:consentBooleanValue2').checked;

    var consentparam3 = document.getElementById('myForm:consentParam3').innerHTML;
    var consentval3=document.getElementById('myForm:consentBooleanValue3').checked;

    jQuery
    .ajax({
        type : "POST",
        crossDomain : true,
        async : false,
        url : "https://localhost:8443/digbyFE/api/v1/user/storeconsentws/",
        contentType : "application/json",
        data : JSON.stringify({
            "userId":username,
            "noticeId": "1",
            "param1":consentparam1,
            "value1":consentval1,
            "param2":consentparam2,
            "value2":consentval2,
            "param3":consentparam3,
            "value3":consentval3
        }),
        dataType : 'json',
        success : function() {
            // on success diplays the data
            storePI();

        },

        error : function(xhr, ajaxOptions, thrownError) {
            alert('failuredata');
            console.log(xhr.status);
            console.log(thrownError);
            console.log(xhr.responseText);
            console.log(xhr);
            // error handlerstorePI();
            //storePI();
        }
    });

}
</script>
```

Figure 4.27 Script to show notice and consent using Store-Notice-Consent-ws

Figure 4.28 shows the actual message that is transmitted to invoke Store-Notice-Consent-ws, using an URL *https://localhost:8443/digbyFE/api/v1/user/storenoticeconsentws/*, which is a POST request. The input parameter is of type “application/json”, actual request message is shown

in the Payload section in the figure, where *userId, param1, value1,etc.* in the JSON request matches the input parameters *Student Id, Consent1, Consent Choice1*, etc. in the form in Figure 4.25. The response from the web service is *201 Created*, on successful storage of PI in the DB, which is shown in the *Status* section of Figure 4.28.

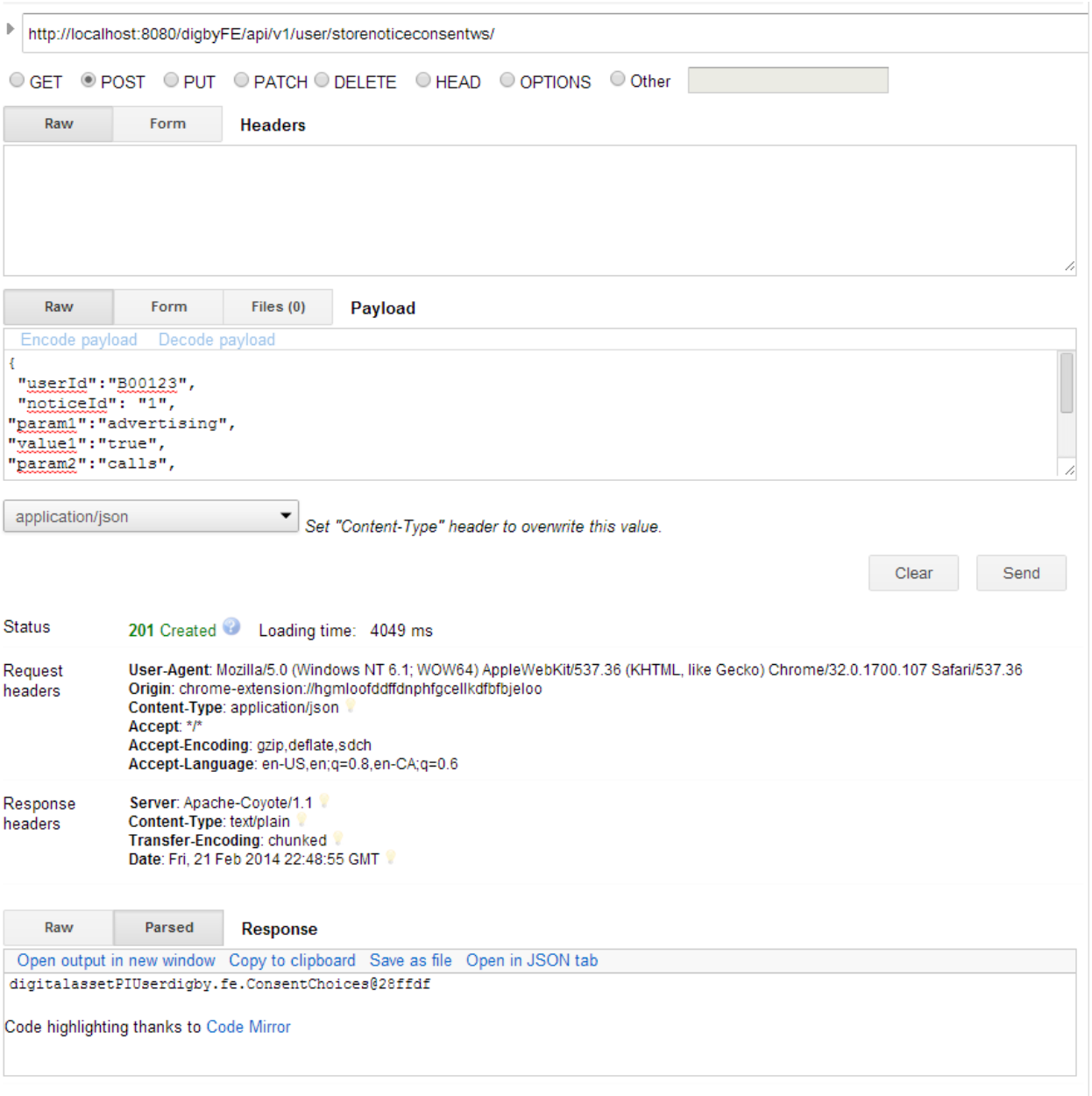

Figure 4.28 Web methods invoked to store notice and consent choices in the DB

Storing PI

Figure 4.29 shows the messages on the web page displayed using the submit script that stores the consent in the database and PI in the database.

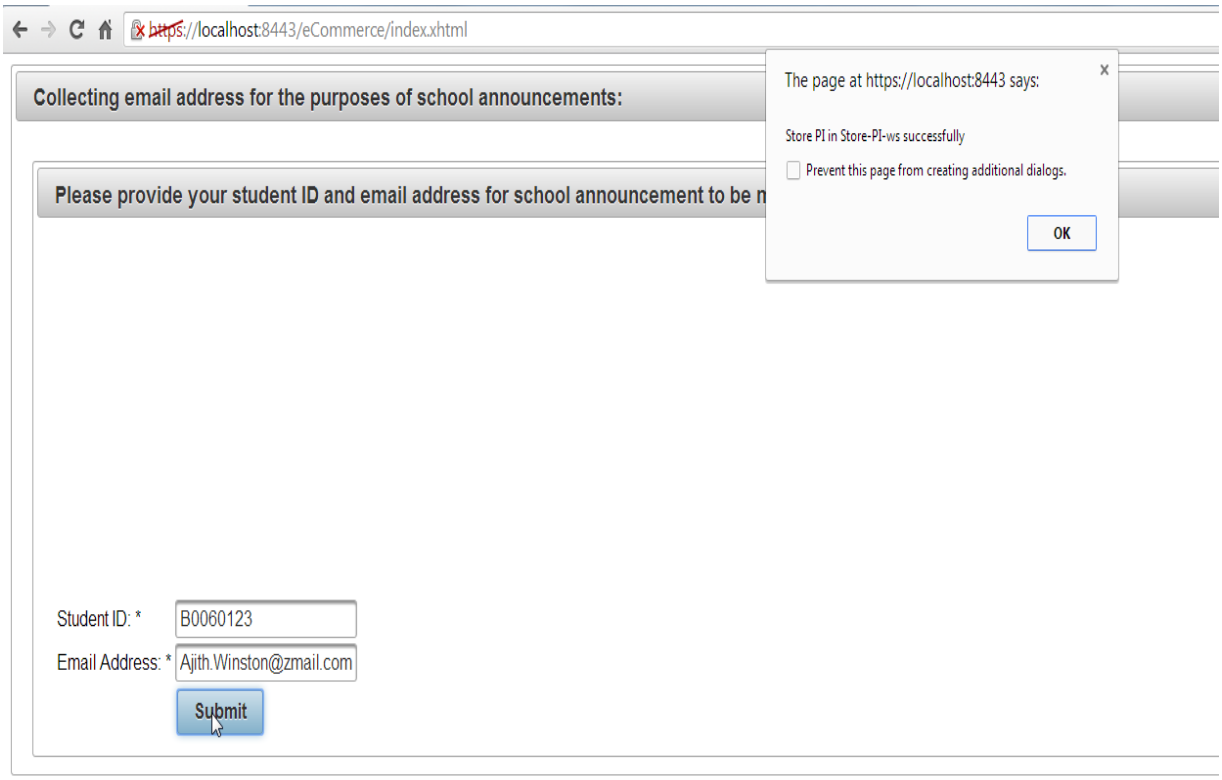

Figure 4.29 Web page once submit button is clicked

Figure 4.27 shows *storeNoticeConsentws()* function in the rendered web page, which executes *success:function* once the consent is stored successfully in the database. Then, the *storePI* function is invoked to create a web service call to *Store-PI-ws,* which stores the PI on the database.

The message parameters and the web service *Store-PI-ws* invocation was not changed. The invocation of the *Store-PI-ws* is initiated once the consent is stored successfully in the DB.

4.3 User Interface

In this section, we describe UI we developed for our prototype for injection of privacy services into a web page. It should be noted that the scope of the UI is limited to injection of privacy services for a given page. How the page is chosen, for injection of

privacy services by PE, is considered to be beyond the scope of this thesis; however, it is briefly discussed.

In terms of selecting a web page for insertion of privacy services, the easy case is of a web page being newly developed by a web-page designer who then forwards the page to PE for privacy services considerations. However, to identify existing web pages that require injection of privacy services relies on the PAWS's PI agent and KB. Briefly, the PI agent uses the PAWS's KB and its reasoning facilities to identify which web pages collect PI from subjects, possibly without notice and consent. Recall that the KB contains information on web pages, applications, web services, and DBs and how they collect, use, and store PI. Briefly, the PI agent searches for web services that insert PI to DBs and then for web services and web pages that invoke them. For instance, if a web service *Store-PI-ws* stores PI in a DB, the agent finds out which web pages invoke it and then find out in the KB whether such pages show notice and obtain consent. If *Store-PI-ws* is invoked by another web services then PI uses transitivity to identify web pages invoking those web services and hence identifying web pages that collect PI without notice and consent.

Figure 4.30 shows the interface and also that it is divided into three sections: *Web Page without privacy services*; *Privacy Service Control*; and the resulting *Web Page with privacy services*. Each of these sections is discussed below.

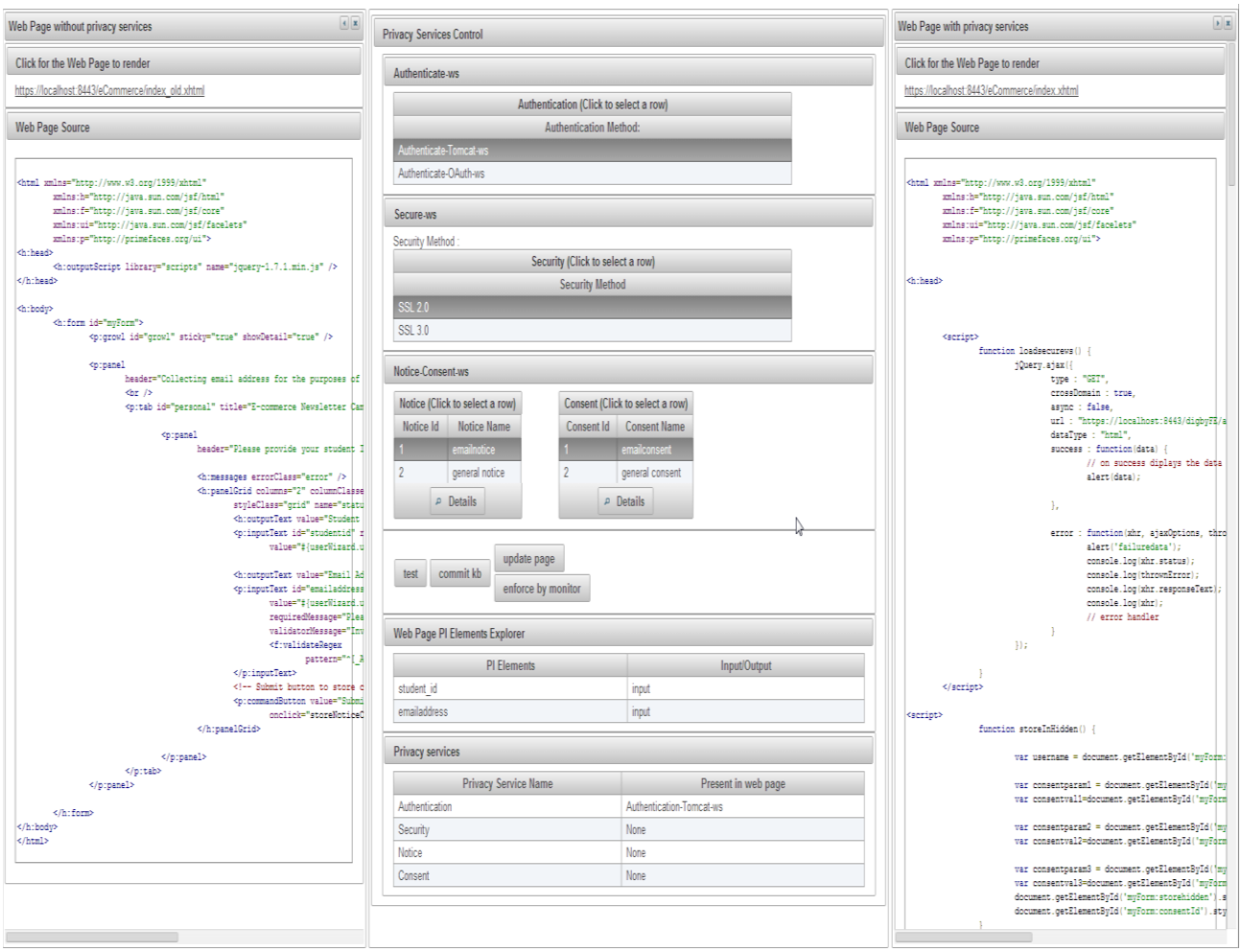

Figure 4.30 Interface for injection of privacy services into a given web page

A web page is shown in two panels: the top panel shows a link to a page that, when the PE clicks on it, renders the page as a browser would. Then, the second panel shows the web page with its source code as it would be shown by a browser when asked to show the web page source code (see Figure 4.31).

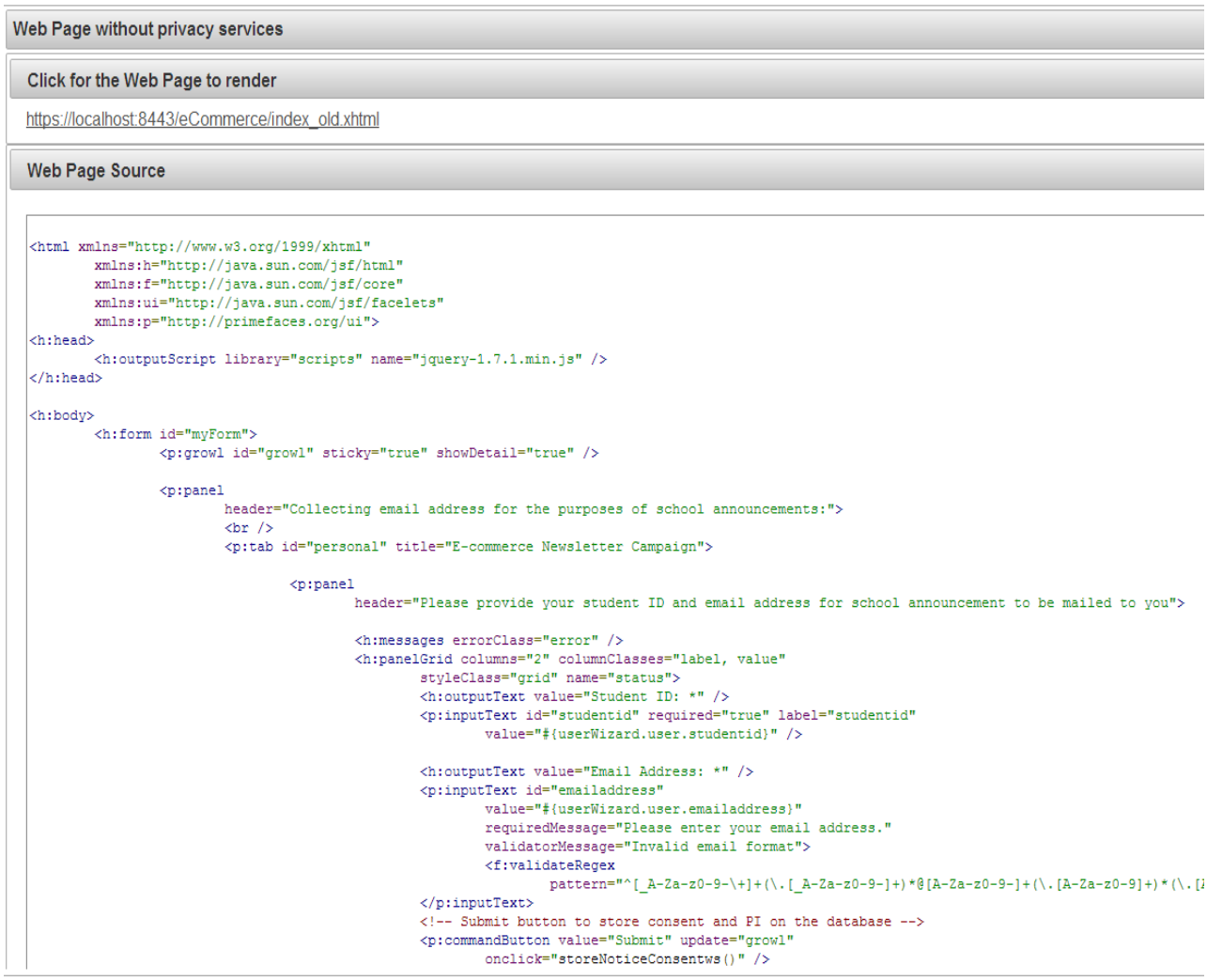

Figure 4.31 Snapshot of a section of interface: web page without privacy services

Privacy Services Control

The PE uses the *Privacy Services Control* part to make selections on which privacy services are to be injected into a web page. The *Privacy Services Control* part has four panels: *Authenticate-ws, Secure-ws, Notice-Consent-ws*, *Web Page PI Elements Explorer, and Privacy services*.

The *Authenticate-ws* panel allows the PE to select which authentication mechanism should be injected. For instance, authentication selections available may include *Authenticate-Tomcat-ws*, or *Authenticate-OAuth-ws*. Figure 4.32 shows a snapshot of *Authenticate-ws* panel with *Authenticate-Tomcat-ws* and *Authenticate-*

OAuth-ws available as authentication mechanisms, and with *Authenticate- Tomcat-ws* being selected for injection.

The *Secure-ws* panel allows the PE to select which secure transfer mechanism should be injected. For instance, secure selections available may include *SSL 2.0* and *SSL 3.0*. Figure 4.32 shows the *Secure-ws* panel with *SSL 2.0* and *SSL 3.0* as secure transfer mechanisms and with *SSL 2.0* being selected for injection.

The *Notice-Consent-ws* panel allows the PE to select which notice should be shown and which consent should be sought. Notices are identified by IDs with notice names also being shown. Similarly, consents are identified by IDs with consent names also being shown. In Figure 4.32, the panel shows two notices and two consents being available with *notice id: 1* being selected for notice and *consent id: 1* being selected for consent. There is a button *Details* beneath the notice table and there is also a *Details* button under the consent table. If a *Details* button is clicked, it shows a page with the details on the selected notice/consent.

Web Page PI Elements explorer panel shows PI elements in the web page in one column, and also information if they are *input* or *output*. When a PI element is *input*, it means that the subject provides its value and the value is stored in the DB, while for an *output* element, its value is retrieved from a DB and shown to the subject. An element can be both *input* and *output*. This information is retrieved from the *Web Page KB*.

Web Services panel shows existing privacy services that are applied on the web page already; this information is retrieved from the *Web Page KB*. *Privacy Service Name* in one column, while the second column, *Present in web page,* indicates which mechanism is used, if any. Figure 4.32 shows that the web page has only authentication service using *Authenticate-Tomcat-ws*.

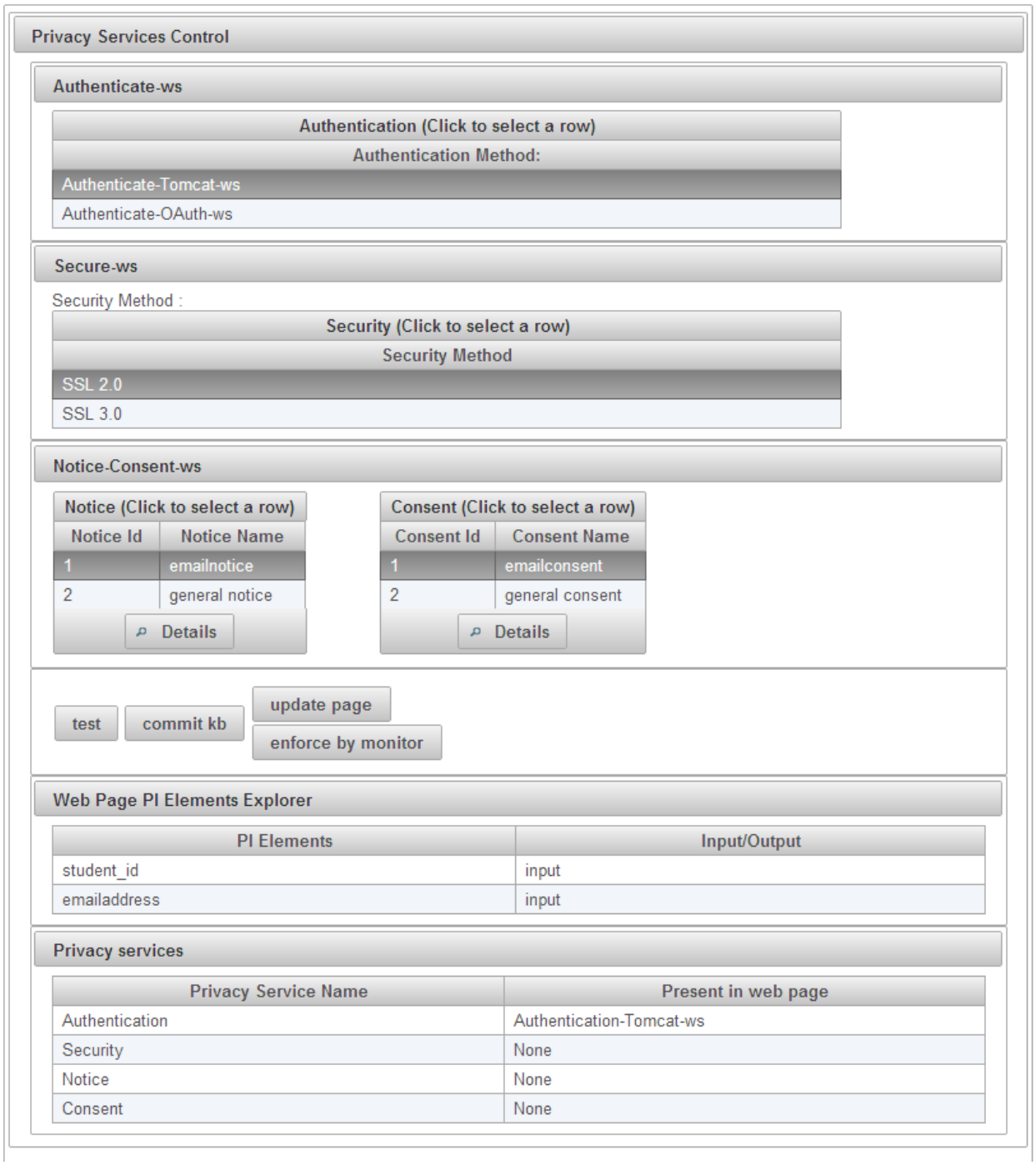

Figure 4.32 Snapshot of a section of the interface: privacy services control part

There are three steps that are followed to inject privacy services, which are enabled by the following buttons: *test*, *commit kb*, and *update page* or *enforce by monitor*. They are discussed below.

Web page With Privacy Services

This UI section shows two views of a web page with injected privacy services in two panels: the top panel shows a link to a page that, when the PE clicks on it, is rendered as it would be by a browser. Then, the second panel shows the web page with its source code (see Figure 4.33). It should be noted that this section shows the web page with injected services only after the PE clicks on the *test* button, as described next.

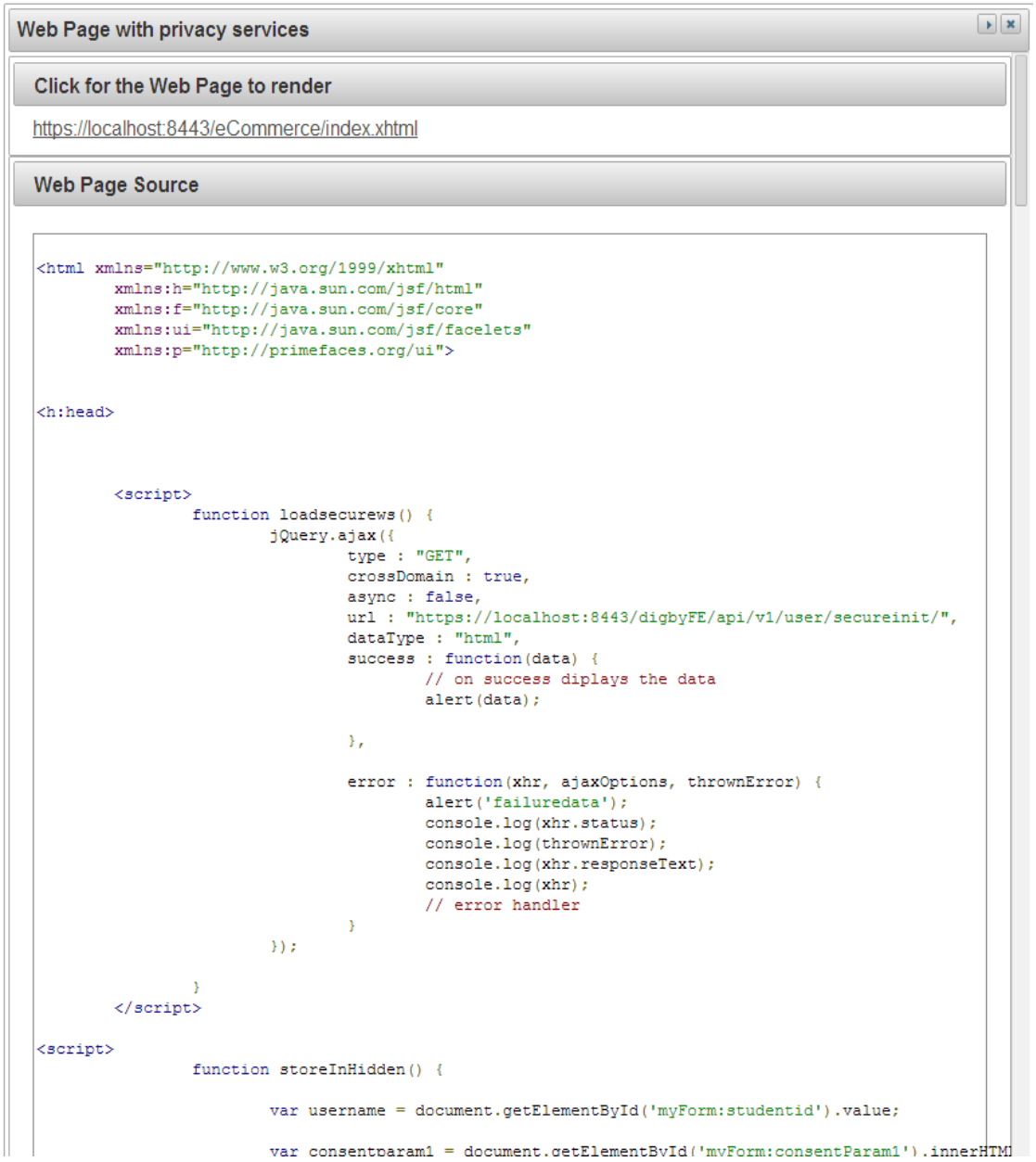

Figure 4.33 Snapshot of a section of the interface: web page with privacy services

Injection of Privacy Services Process

The process of injecting privacy services is controlled by three buttons: *test*, *commit kb*, and *update page* or *enforce by monitor*. When the page is initially selected, the PAWS's KB is accessed to retrieve information on the web page and present it in the *Web Page PI Elements explorer* panel. Following this, PE makes selections on which privacy services are to be injected using the *Authenticate-ws, Secure-ws, Notice-Consent-ws*, *and Privacy services* panels of the *Privacy Service Control.* Once PE selects privacy services to be injected, (s)he presses the *test* button which generates a new page with the privacy services being injected. The newly generated web page is shown in the *Web page with privacy services* panel (located to the right from the *Privacy Service Control*). PE may change the selections on which privacy services are to be injected and test again. Once PE is satisfied, (s)he commits the changes to the PAWS's KB. Finally, PE has to determine whether the privacy services are to be applied by actual modification of the web page or whether the privacy services should be injected on the fly by PAWS's monitor – PE either clicks on the *update page* or *enforce by monitor* button, respectively.

CHAPTER 5 : OVERHEAD DELAYS

In this chapter we show the execution delays on the scenario that is described below. We are not concerned about the execution delays incurred by PE guiding the injection of privacy services as this process is performed by the PE off-line, that is, subjects are not involved in this process. The delays to programmatically inject the code to invoke the privacy services were insignificant when compared to the delays in decision-making by the PE. We report on delays when the web page is used without privacy services and then, for comparison, when the subject uses the page with injected privacy services.

Each reported experiment was performed a number of times in order to determine variability of delays, and to identify their causes. As human action incurs high delay that is unpredictable and much higher relative to the operations performed by a web service, we avoid the human action by the subject by using a script on the web page. We performed the experiment in two scenarios. In one, all software is housed on one and only physical server with no other services running but those required for experimentation. In the other one, the client browser is connected to Internet to communicate with a web server that is connected to a DB server using a local area network.

5.1 Servers Housed on the Same Network

The set of experiments were performed on a physical server Power Edge that runs Cent OS 6.5 operating system, Tomcat 7.0 web server, and MySQL Server 5.0 database server. The test server is Power Edge that runs Cent OS 6.5 with:

- Intel Xeon processor at 2.6 GHz
- 7296 MB memory
- 1 TB Hard Disk

We ran each experiment 5 times and determined that the variation in delays for the five runs in each experiment were insignificant. Recall, human interaction is simulated by a script on the web page (filling-in data and clicking on buttons). There are no network delays, all software is on one platform, and, furthermore, only services needed for the experimentation were running. Thus, when each experiment was running, all operations

were executed in the same sequence and some variability was only detected when accessing a DB.

5.1.1 Delays for a Web Page Without Privacy Services

Table 5.1 shows delays for the following activities when an unmodified web page uses a form to obtain the student ID and the address:

- ***Store-PI-ws* invocation** – measured from the time the submit button is clicked until the time the actual invocation of the web service is executed (until just before the HTTP POST command that transfers the web service request to the server).
- ***Store-PI-ws* invocation message transit** – measured from the time of the invocation of the web service (issuing the HTTP POST command) until the first instruction of the web service.
- ***Store-PI-ws* execution** – measured from the first instruction of the web service, until the last instruction. It thus includes the delay to store the private data in the DB using an SQL insert statement.
- ***Store-PI-ws* reply delivery and display** – Once the private data is stored in the database, the web service replies with a status code 201 and, once the reply is received, a message is displayed on the browser that the user input was successfully stored.
- ***Total Delay*** – measured from the invocation of the *Store-PI-ws* until the end of the *Store-PI-ws* reply delivery. It thus excludes the delay to initially fetch the web page and display it on the browser.

It should be noted that the initial retrieval and load of the web page was 3 ms.

Modules on the web service invocation	**Delay *(in ms)***
Store-PI-ws invocation	1.6
Store-PI-ws invocation message transit	2.8
Store-PI-ws execution	3.2
Store-PI-ws reply delivery and display	2.3
Total Delay (excludes the first web page load)	9.9

Table 5.1 Delay for web page without privacy services (one physical server)

5.1.2 Delays for Web Page With Privacy Services

In this section we report on delays for a page with privacy services. Table 5.2 shows delays for the following activities when the web page, modified with privacy web services, is used to obtain information from the subject:

- ***Secure-ws* invocation** – measured from the time, the URL is loaded and before the form is rendered on the browser until the time the actual invocation of the web service is executed (until just before the HTTP GET command that transfers the web service request to the server).
- ***Secure-ws* execution**– measured from the time the web service is invoked until the final instruction of the web service that establishes the SSL connection. We assume, while calculating the time delays, that the subject has accepted the certificate without any delays.
- ***Authenticate-Tomcat-ws* page retrieval request message transit** - measured from the time of invocation of the web service (issuing the HTTP GET command) until the first instruction of the web service. This request is invoked right after the Secure-ws execution.
- ***Authenticate-Tomcat-ws* page retrieval execution** - measured from the first instruction of the web service being executed until its last instruction. This retrieves as a response the Authentication page to be displayed on the web page.
- ***Authenticate-Tomcat-ws* page retrieval reply message transit** - measured from the time of the last instruction of the *Authenticate-Tomcat-ws* page retrieval until the display of Authentication page by the browser.
- ***Authenticate-Tomcat-ws* invocation** – measured from the time the subject clicks the login button to the invocation of *Authenticate-Tomcat-ws*. (until just before the HTTP POST command that transfers the web service request to the server). *Note:* Tomcat's authentication mechanism displays login button, in our scenario we refer to it as authenticate button.
- ***Authenticate-Tomcat-ws* request message transit** - measured from the time of invocation of the web service (issuing the HTTP POST command) until the first instruction of the web service.

- ***Authenticate-Tomcat-ws* execution** – measured from the first instruction of the web service being executed until its last instruction. It includes the delay to decrypt the credentials and to invoke and execute the Tomcat's authentication mechanism using *tomcat-users.xml* file. We assume that the subject has passed authentication successfully.
- ***Authenticate-Tomcat-ws* reply message transit**- measured from the time of the last instruction of *Authenticate-Tomcat-ws* web service until the display of the reply by the browser with status code 200, which means the subject is authenticated.
- ***Notice-Consent-ws* invocation**– measured from the time the subject "enters" the form and the first instruction of the script invoking the *Notice-Content-ws* is executed until the HTTP GET command to send the message.
- ***Notice-Consent-ws* invocation message transit** – measured from the time of the invocation of the web service (issuing the HTTP GET command) until the first instruction of the web service.
- ***Notice-Consent-ws* execution** – measured from the first instruction of the web service being executed until its last instruction. It includes delay to open a DB connection and retrieve the notice and consent information using the notice and consent IDs as a key. We use one SQL statement on one relational table for this purpose.
- ***Notice-Consent-ws-reply* message transit and display** – measured from the time of the last instruction of the *Notice-Consent-ws* web service until the display of the reply (notice and consent request) by the browser.
- ***Storage-in-hidden-fields*** – measured from the subject clicking on the accept button, which causes execution of script to store the consent input by the subject in hidden fields, until the last instruction of that script.
- ***Store-Notice-Consent-ws* invocation** – measured from the time of the first instruction in the script associated with the form's Submit button, until the first HTTP POST command invoking the *Store-Notice-Consent-ws* web service. Recall that the subject, after filling the form's data, clicks on the Submit button invoking a script that collects the information from the form's fields and uses it as parameters to invoke the *Store-Notice-Consent-ws* web service.

- ***Store-Notice-Consent-ws* message transit** – measured from the time of the invocation of the web service (issuing the HTTP POST command) until the first instruction of the web service.
- ***Store-Notice-Consent-ws* execution** – measured from the first instruction of the web service being executed until its last instruction. It, however, includes the delay to store the information on notice and consent passed as parameters in the DB.
- ***Store-Notice-Consent-ws-reply* message transit and delivery** – measured from the time of the last instruction of the *Store-Notice-Consent-ws* web service until the reception of the reply by the browser and its display by the browser.
- ***Total Privacy Services Delay*** – sum of the above delays.
- ***Store-PI-ws* invocation** – as in Table 5.1.
- ***Store-PI-ws* invocation message transit** – as in Table 5.1.
- ***Store-PI-ws* execution** – as in Table 5.1.
- ***Store-PI-ws* reply delivery and display** –as in Table 5.1.
- ***Total Delay*** – measured from the load of the web page until the last message, informing the subject of successful storage of data, is received and displayed. It thus excludes the delay to initially fetch the web page and display it on the browser.

Note that the initial retrieval and load of the web page was 3 ms.

Modules on the web service invocation	**Delay *(in ms)***
Secure-ws invocation	1.6
Secure-ws execution	2.3
Authenticate-Tomcat-ws page retrieval request message transit	2.1
Authenticate-Tomcat-ws page retrieval execution	2.3
Authenticate-Tomcat-ws page retrieval reply message transit	2.4
Authenticate-Tomcat-ws invocation	1.6
Authenticate-Tomcat-ws request message transit	2.64
Authenticate-Tomcat-ws execution	2.9
Authenticate-Tomcat-ws reply message transit	2.3
Notice-Consent-ws invocation	1.6
Notice-Consent-ws invocation message transit	2.46

Notice-Consent-ws execution	2.6
Notice-Consent-ws-reply message transit and delivery	2.8
Storage-in-hidden-fields	1.3
Store-Notice-Consent-ws invocation	1.6
Store-Notice-Consent-ws message transit	2.92
Store-Notice-Consent-ws execution	3.2
Store-Notice-Consent-ws-reply message transit and display	2.3
Total Privacy Services Delay	40.92
Store-PI-ws invocation	1.6
Store-PI-ws invocation message transit	2.8
Store-PI-ws execution	3.2
Store-PI-ws reply delivery and display	2.3
Total Delay (excludes the first web page load)	50.82

Table 5.2 Delay for web page with privacy services (one physical server)

5.2 Servers Connected by Networks

Tables 5.1 and 5.2 show the delays when all software was housed on the same server and hence the network delay was non-existent. In this section we report on the same experiments but when network delays become a factor. The web server is hosted on the network and the client browser communicates with the web server over the Internet.

The set of experiments was performed on two physical servers *"projects.cs.dal.ca"* and *"db.cs.dal.ca"*. "*projects.cs.dal.ca"* is Power Edge that runs Cent OS 6.5 with: Intel Xeon processor at 2.6 GHz, 7296 MB memory, and 1 TB Hard Disk, hosts the web server, which is Tomcat Server 7.0. *"db.cs.dal.ca"* is Power Edge that runs Cent OS 6.5 with: Intel Xeon processor at 2.6 GHz, 7296 MB memory, and 1 TB Hard Disk, hosts the DB server, which is MySQL Server 5.0.

Figure 5.1 shows the network diagram for servers connected by networks. All the web services are housed on the web server (*projects.cs.dal.ca*) that interacts with the DB server (*db.cs.dal.ca*) that is housed on the same network. Both servers are connected by

local area networks. This is an industry standard to house the web server and the DB server on the same network, for security and faster retrievals.

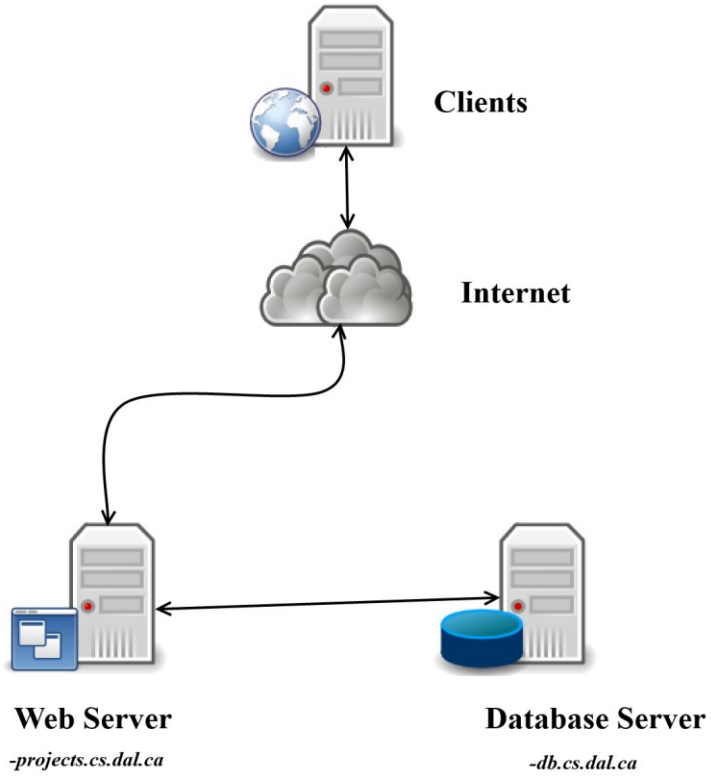

Figure 5.1 Connection for servers connected by networks

Figure 5.2 shows, using traceroute, the presence of routers when the client is trying to connect to the web server *"project.cs.dal.ca"*. Figure 5.3 shows that the connection from the web server to the DB sever does not pass through routers.

```
C:\Windows\system32>tracert projects.cs.dal.ca

Tracing route to nano.cs.dal.ca [129.173.213.134]
over a maximum of 30 hops:

  1     1 ms    14 ms     1 ms  192.168.0.1
  2    45 ms    27 ms    28 ms  71.7.156.1
  3    18 ms    18 ms    11 ms  blk-212-126-153.eastlink.ca [173.212.126.153]
  4    12 ms    22 ms    45 ms  ns-hlfx-dr001.ns.eastlink.ca [24.215.101.101]
  5    20 ms     9 ms    17 ms  ns-hlfx-asr001.ns.eastlink.ca [24.215.101.246]
  6    71 ms    22 ms    13 ms  dal-gw.eastlink.ca [24.222.79.122]
  7    28 ms    17 ms    12 ms  GigaPOP-gw.acorn-ns.Ca [198.166.1.37]
  8    15 ms    18 ms    13 ms  GWC6A60100.Backbone.Dal.Ca [198.166.1.18]
  9    16 ms    25 ms    18 ms  shovel.CS.Dal.Ca [129.173.212.16]
 10    16 ms    19 ms    14 ms  nano.CS.Dal.Ca [129.173.213.134]

Trace complete.
```

Figure 5.2 Traceroute to the web server "projects.cs.dal.ca"

```
C:\Windows\system32>tracert db.cs.dal.ca

Tracing route to Orange.cs.dal.ca [129.173.22.38]
over a maximum of 30 hops:

  1     1 ms    <1 ms     2 ms  wormhole-v222.research.cs.dal.ca [129.173.66.1]
  2    <1 ms     1 ms    <1 ms  GW81AD0D5C.Backbone.Dal.Ca [129.173.13.93]
  3    <1 ms    <1 ms    <1 ms  Orange.CS.Dal.Ca [129.173.22.38]

Trace complete.
```

Figure 5.3 Traceroute to the database server "db.cs.dal.ca"

Table 5.3 shows the delays as in the previous section. This time, however, the network delays are dominant. Table 5.4 shows the delays in the network with privacy services inserted to the web page. The Total Overhead Delay includes delays caused due to the other components after the insertion of privacy web services in the network along with the delay caused due to *Store-PI-ws* with the network delay. As long as the network delays on the web page are acceptable by the subject the network delays on the network due to insertion of privacy web services are acceptable as well. It should be noted that the initial retrieval and load of the web page was *107.4 ms* for a page without privacy services, and it was *110.7 ms* for a page with privacy services.

Modules on the web service invocation	***Delay (in ms)***
Store-PI-ws invocation	1.6
Store-PI-ws invocation message transit	62.3
Store-PI-ws execution	10.35
Store-PI-ws reply delivery	49.8

Total Delay (excludes the first web page load)	124.05

Table 5.3 Delay for web page without privacy services (servers connected by networks)

Modules on the web service invocation	**Delay *(in ms)***
Secure-ws invocation	1.6
Secure-ws execution	60.2
Authenticate-Tomcat-ws page retrieval request message transit	23.6
Authenticate-Tomcat-ws page retrieval execution	15.6
Authenticate-Tomcat-ws page retrieval reply message transit	20.5
Authenticate-Tomcat-ws invocation	1.6
Authenticate-Tomcat-ws request message transit	86.4
Authenticate-Tomcat-ws execution	20.8
Authenticate-Tomcat-ws reply message transit	10.3
Notice-Consent-ws invocation	1.6
Notice-Consent-ws invocation message transit	89.6
Notice-Consent-ws execution	26.8
Notice-Consent-ws-reply message transit and delivery	110.6
Storage-in-hidden-fields	1.3
Store-Notice-Consent-ws invocation	1.6
Store-Notice-Consent-ws message transit	130.15
Store-Notice-Consent-ws execution	12.2
Store-Notice-Consent-ws-reply message transit and display	50.36
Total Privacy Services Delay	664.81
Store-PI-ws invocation	1.6
Store-PI-ws invocation message transit	62.3
Store-PI-ws execution	10.35
Store-PI-ws reply delivery and display	49.8
Total Delay (excludes the first web page load)	788.86

Table 5.4 Delay for web page with privacy services (servers connected by networks)

5.3 Analysis

5.3.1 Delays when Software is Housed on One Physical Server

Delays for Web a Page Without Privacy Services

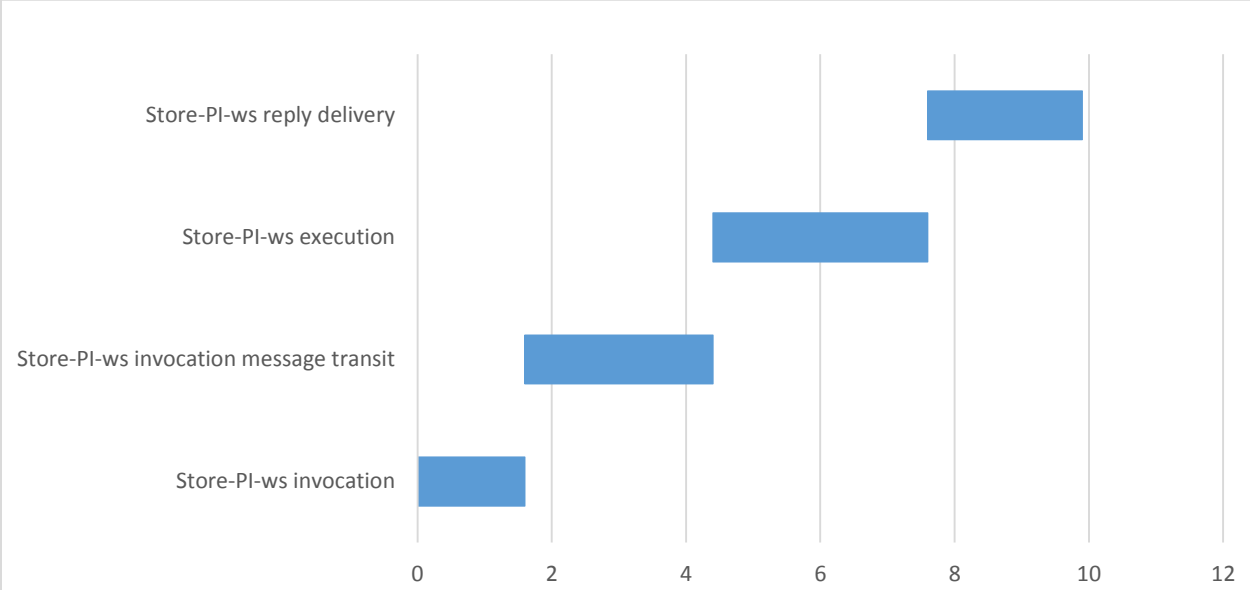

Figure 5.4 Delay for web page without privacy services (one physical server)

In this section, we overview the delay caused for a web page without privacy services, where the client browser and web and database servers were hosted on the *same physical server*. The delay is depicted in Figure 5.4 using stacked bar chart with the *components* invoked along *Y axis*, and time delay *in ms* along *X axis*. The time to invoke *Store-PI-ws* is *1.6ms,* which is minimal as the script is on the client side, which invokes a web service call to store PI. *Store-PI-ws message transit* takes *2.8 ms*, which is caused due to transmission of JSON data from the client to the web server. This also includes mapping of the appropriate web method that needs to be executed by the request. *Store-PI-ws* takes *3.2 ms* caused due to opening a connection and storing the data on the DB server. *Store-PI-ws reply delivery* takes *2.3 ms*, caused by the transmission of response from the web server back to the client. The time delay for *Store-PI-ws reply delivery* is less in comparison to *Store-PI-ws* invocation message transit, as the reply message does not have payload in comparison to the request, which has *JSON message*.

Delays for Web Pages With Privacy Services

Figure 5.5 Delay for web page with privacy services (one physical server)

In this section, we overview the delay caused by a web page with injected privacy services, where the client browser and the web and database servers are hosted on the *same physical server*. The delay is depicted in the stacked bar chart (see Figure 5.5) with the

components along Y axis and delay *in ms* along X axis. *Authenticate-Tomcat-ws page retrieval* takes *6.8ms (Authenticate-Tomcat-ws* page retrieval request message transit - *2.1ms, Authenticate-Tomcat-ws* page retrieval execution - *2.3ms, Authenticate-Tomcat-ws* page retrieval reply message transit - *2.4ms)* to render the Authentication page on the browser using the GET request. The delay to invoke *Secure-ws* or to invoke *Authenticate-Tomcat-ws* takes *1.6 ms,* which is minimal, as the script is on the client side. *Authenticate-Tomcat-ws invocation message transit* delay is *2.64 ms*, which requires messages to be sent as JSON string, and includes transfer from the browser to the web service invocation. *Secure-ws execution* takes *2.3 ms* to establish SSL connection. This delay is less than that of *Authenticate-Tomcat-ws* execution, as *Secure-ws* does not require querying information from *xml file* or DB, in contrast to the authentication mechanism which does. *Authenticate-Tomcat-ws execution* takes *2.9ms*, which includes authenticating the subject using Tomcat's Authentication mechanism.

Notice-Consent-ws invocation and message transit and *Store-Notice-Consent-ws invocation and message transit* take *1.6 ms* each, which is on par with other web service invocation as the script is on the client side. *Notice-Consent-ws message transit* and *Store-Notice-Consent-ws message transit* take *2.46 and 2.92 ms,* respectively. The delay is smaller for the *Notice-Consent-ws message transit* as the request is a GET request and there are query parameters that are transferred, whereas in *Store-Notice-Consent-ws message transit* the actual notice and consent choices are sent as JSON request.

5.3.2 Effect of Network Communication

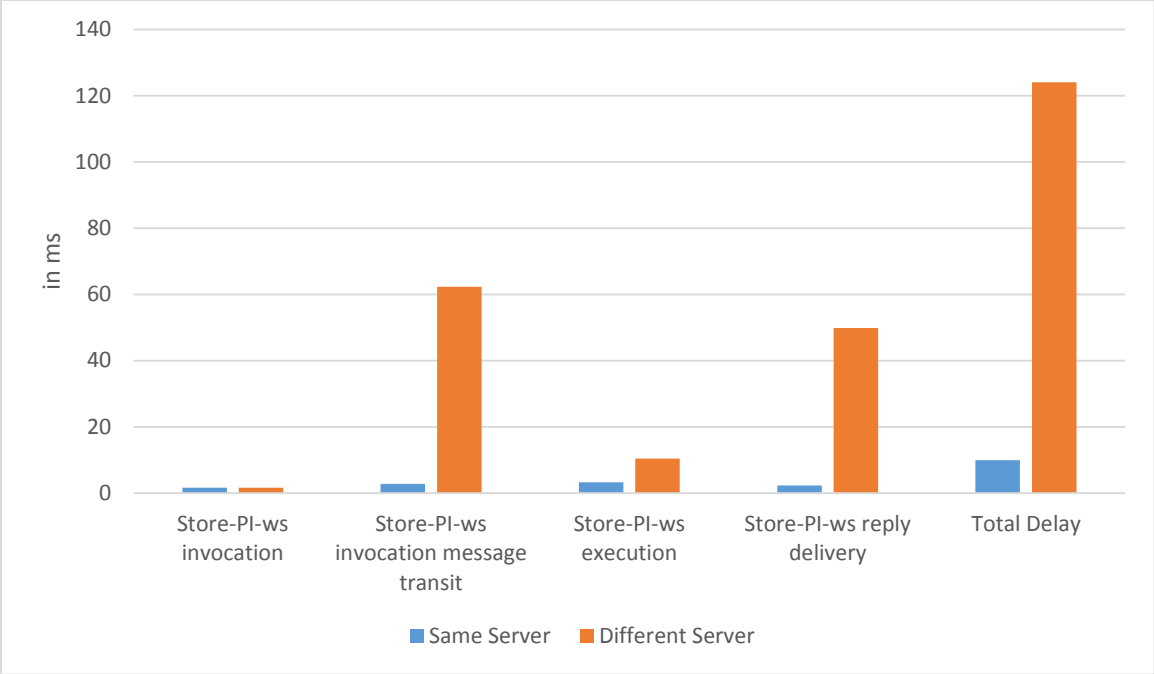

Figure 5.6 Delays for a web page without privacy services with and without network

Figure 5.6 shows the comparison for a web page without privacy services when there are no network delays and with network delays. Clearly, network delays are dominant as request and reply messages are routed over the Internet. This is clearly shown from the components, *Store-PI-ws invocation message transit*, and *Store-PI-ws reply delivery*. *Store-PI-w*s execution delays also differ. When all software, including the DB system, is housed on one physical server, there are no network delays. When the DB server is connected to the web server via a local area network, there is communication delay. It should be noted, however, that the communication delay over the local network is much smaller than the communication delay over the Internet as is the case for *Store-PI-ws invocation message transit* and *Store-PI-ws reply delivery*. *Store-PI-ws invocation* has the same delay as it is invoked from the client browser in both the cases.

5.3.4 Delays due to Message Size

In this section we perform experiments where the size of reply messages in *Notice-Consent-ws* varies. Recall that thus far we used we used a scenario in which the notice and consent were deliberately short in order to minimize the size of figures. Here we report on

a scenario, wherein the notice message is much larger. For this purpose, we adopted a notice from Treasury Board of Canada Secretariat at https://www.tbs-sct.gc.ca/pol/doc-eng.aspx?section=text&id=24227. Figure 5.7 a fictitious web page with the adopted notice under "Notice Details". It also shows consent choices.

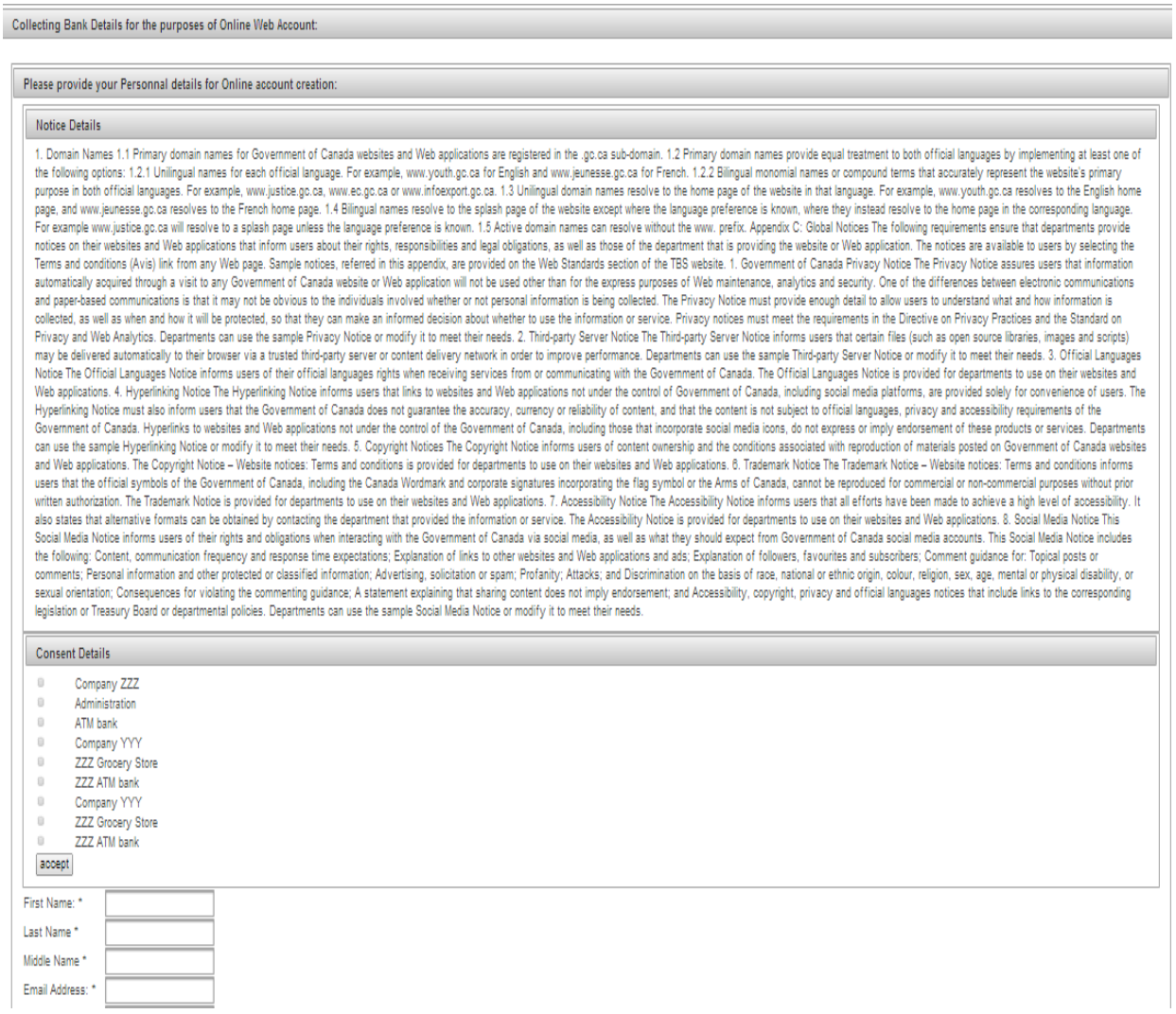

Figure 5.7 Web page with large message size

We performed the experiments as reported in the previous section when the client browser was communicating with the server over the Internet and measured the delays. We did not observe any increase in delays in comparison to the case when the notice and consent were those as shown in Figure 4.20. This is not surprising as the message size increase is not

large enough to be observable. HTTP uses TCP as its delivery mechanism and a marginal increase in the length of its payload message will not cause observable increase in delay.

5.3.5 Decreasing Internet Delays

Examination of Figure 5.6 reveals that delays over the Internet are dominant. Examination of Table 5.6 not only confirms that delays over the Internet are dominant, but also that there is a number of interactions that occur over the Internet between the client browser and the web server, namely due to the web services request/reply messages. We can decrease delays by bundling *Authenticate-Tomcat-ws and Notice-Consent-ws* into one web service call and thus reducing the number of web services request/reply messages travelling over the Internet. Bundling means, it is one web service that performs both functions to authenticate and to retrieve notice and consent. Figure 5.8 shows the resulting delays.

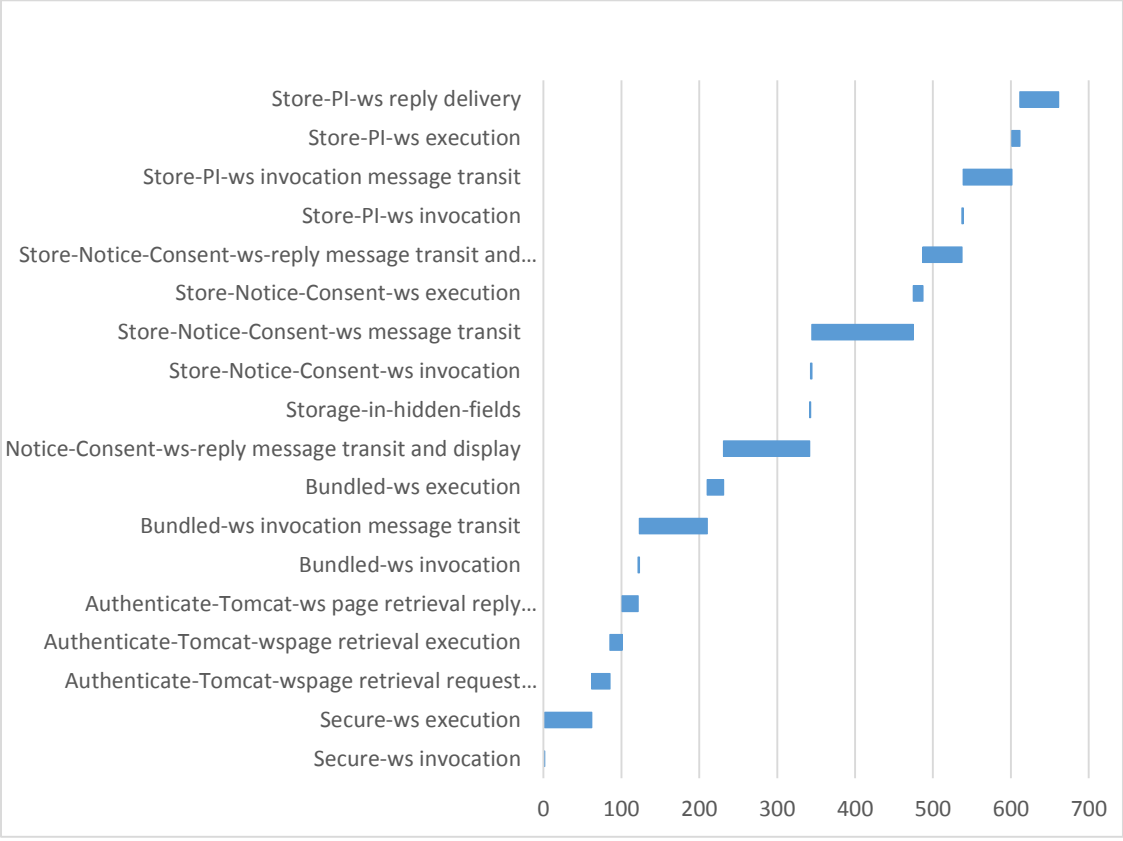

Figure 5.8 Bundling of web services requests

5.4 Discussion

Table 5.4 shows that the total delay caused by injecting privacy services to obtain secure data transfer, authenticate, show notice, and obtain consent, is about *788.86 ms (0.78 seconds)*. However, after *469 ms (0.46 seconds)*, the notice and consent page are displayed requiring interaction with the subject. If bundling of authentication with notice and consent is performed as discussed in section 5.3.5, this delay is *341 ms (0.34 seconds), which is equivalent to loading of a web page from the web server*. Once the subject provides consent, he/she is able to enter the data in the form and receive confirmation in *319 ms (0.3 seconds)*. Clearly, these delays are acceptable in that the subject will not be annoyed due to waiting for interaction. It is also clear, however, that the delays are primarily dependent on the delays due to Internet over which we have no control.

CHAPTER 6 : SUMMARY, CONCLUSIONS, AND FUTURE WORK

6.1 Summary and Conclusions

We focussed on privacy services in the stages of the SDLC after the development stage. We provide the ability to inject privacy services into the web pages that have been developed without privacy considerations, specifically, that have been developed without privacy services to show notice, obtain consent, authentication, and secure data transfer. We proposed software architecture for inclusion of privacy services into web pages. From the perspective of this thesis, webpages were classified into two types: ***HTML*** pages, such as pages in HTML 5, or **server-script generated**, such as *XHTML, PHP, or ASPX pages.* The software architecture relies on PAWS for information from its KB. We developed a prototype to be used by PE to inject privacy services into a web page by updating the KB with the selection of privacy services he/she has made, and by inserting appropriate code into a XHTML page. For pages generated by other scripting languages, such as PHP or ASPX, the privacy services are injected by the PI Monitor whenever the web page is requested by a browser. Individual components of the injected privacy services are discussed in detail and the feasibility of the approach is demonstrated with the prototype as a proof-of-concept. We also measure and report delays for individual components in the privacy services with and without network delays. We observe that Internet delays are dominant. Although we have no control over Internet delays, we decrease overall delays by reducing messaging over the Internet, and hence overall delays, by bundling of privacy services through one web service invocation.

Although by developing a proof-of-concept prototype we clearly demonstrated that our approach is valid and feasible and worthy of further pursuit, it must be acknowledged that it is highly dependent of the PAWS and its KB.

6.2 Suggestions for Future Work

The approach in this thesis was demonstrated on HTML and XHTML pages. Future work should explore whether it can also be applied to PHP and ASPX pages. As

this work depends on the PAWS's KB, further work is required on obtaining the prerequisite knowledge on use of PI by web services and applications and on storage of PI in DBs.

REFERENCES

Act, F. O. (2002). Privacy Impact Assessments. United States: U.S Department Of State.

Belqasmi, F., Singh, J., Melhem, S., & Glitho, H. R. (2012). SOAP-Based vs. RESTful Web Services. Programmatic Web Interfaces, 54-63.

Bodorik, P., Jutla, D., & Dhillon, I. (2009). Privacy Compliance with Web Services. Information Assurance and Security, 412-416.

Canny, J. (2002). Collaborative Filtering with Privacy. 2002 IEEE Symposium Proceedings Security and Privacy, (pp. 45-57).

Cavoukian, A. (2013). About PbD. Retrieved 01 05, 2013, from Privacy by Design: http://www.privacybydesign.ca/index.php/about-pbd/

Chanda, S., & Foggon, D. (2013). Beginning ASP.NET 4.5 Databases. Apress.

Commission, F. T. (1998). Privacy Online: A Report to Congress. United States: Federal Trade Commission.

Cui, J., Wang, L., Mei, J., Zhang, D., Wang, X., Peng, Y., & Zhang, W. (2010). CAPTCHA design based on moving object recognition problem. International Conference on Information Sciences and Interaction Sciences, (pp. 23-25).

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. IRVINE: UNIVERSITY OF CALIFORNIA.

Goldberg, I., Wagner, D., & Brewer, E. (1997). Privacy-enhancing technologies for the Internet. IEEE Proceedings Compcon, 103-109.

Hartikainen, M., Tampere, F., Laitkorpi, M., Ruokonen, A., & Systa, T. (2011). How to drill down to ReST APIs: Resource harvesting with a pattern tool. 13th IEEE International Symposium on Web Systems Evolution, (pp. 135-140).

Justice, M. o. (2011, 04 01). Personal Information Protection and Electronic Documents Act. Retrieved from Justice Laws Website: http://laws-lois.justice.gc.ca/PDF/P-8.6.pdf

Jutla, D. N., Bodorik, P., & Ali, S. (2013). Engineering Privacy for Big Data Apps with the Unified Modeling Language. IEEE International Congress on Big Data, 38-45.

NIST. (1974). Privacy Act. U.S: United States Department of Health, Education and Welfare.

Olivier, M., & Oberholzer, H. (2005). Privacy Contracts as an Extension of Privacy Policies. Data Engineering Workshops, 1192-1193.

Pitofsky, R., Anthony, S., Thompson, M., Swindle, O., & Leary, T. (2000). Privacy Online: Fair Information Practices in the Electronic Marketplace. A Report to Congress.

Sabo, J., Willett, M., Brown, P., & Jutla, D. (2013). Privacy Management Reference Model and Methodology, OASIS PMRM TC Standards Track Committee Specification, March 25 2013.

Serme, G., Oliveira, A., Massiera, J., & Roudie, Y. (2012). Enabling Message Security for RESTful Services. IEEE 19th International Conference on Web Services, 114-121.

Spiekermann, S., & Cranor, L. F. (2009). Engineering Privacy. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 67-82.

Sun, H.-M. (2000). An efficient remote use authentication scheme using smart cards. IEEE Transactions on Consumer Electronics, (pp. 958-961).

Tong, K. (2012). PRIVACY MONITORING AND ENFORCEMENT IN A WEB SERVICE . Dal Libraries.

Xu, M., Wijesekera, D., & Zhang, X. (2011). Runtime Administration of an RBAC Profile for XACML. IEEE Transactions on Services and Computing, (pp. 286-199).