LONG-TERM SESSION PERSISTENCE FOR A JAVA-BASED WEB APPLICATION FRAMEWORK

Martin Rosin

33303

Master Thesis in Software Engineering

Supervisor: Ivan Porres
Software Engineering Laboratory
Department of Information Technologies
Division for Natural Sciences and Technology
Åbo Akademi University

ABSTRACT

This thesis presents the design and implementation of an experimental module modifying the internal session management of the framework Vaadin to enable long-term session persistence. The module persists the state of clients outside the server environment to external storage medias *Amazon Simple Storage Service* and HTML5 *Local Storage*. The module was develop to evaluate whether the proposed storage medias are viable options for storing the state of clients for a long period time, therefore lengthening the session life length of clients.

Keywords: Session management, session persistence, session manager, long-term sessions

ACKNOWLEDGEMENTS

My gratitude goes out the Software Engineering laboratory at Åbo Akademi and Professor Ivan Porres for giving me the opportunity and idea to research such an interesting topic. I also want to acknowledge Artur Signell for helping me understand the inner workings of the Vaadin framework.

Furher I want to give thanks to Max Weijola, Kristian Nordman and Benjamin Byholm for constructive discussions and encouragements.

Martin Rosin

Turku, May 2014

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

INTRODUCTION

The demand for more complex functionality in web applications has increased, bringing web applications closer to traditional applications running on a native operating system. One big part of this is the possibility to create stateful web applications. The HTTP-protocol is one of the most significant underlying technologies of the web [17]. Unfortunately the protocol is completely stateless by default. Today though there are methods developed to keep the state of clients alive and available over multiple requests.

One of the common ways to keep track of state in modern web applications is by storing it server-side in the main memory. Unfortunately this puts strain on the server, since memory is of a finite size. Therefore, the state data of clients needs to be managed so that there is enough memory available for all clients. This is usually done by giving the state data a lifetime, meaning that the state of clients only exists for a certain period of time. The state, though, could be removed from memory prematurely, causing inconvenience for clients. For example, when a client has been using a web application for a long period of time and then takes a pause. Once the client returns, all the state data may have been deleted, forcing the client to redo all requests. What if the state data could be moved out from main memory into another storage media, allowing the data to be persisted over a longer period of time? Could this be done effectively for modern web applications of any kind?

To be able to persist the state of clients for a longer period of time effectively would benefit both users and developers of web applications. Having the possibility to keep state for a longer period of time can bring forth a more intuitive programming model for a developer. From a users perspective, a web application can be seen as more efficient if it supports keeping track of state, since it often means time saved spent remaking the same requests. Time saved for a customer can be beneficial for an application provider, giving the state data of clients value.

This thesis investigates whether it is possible to move the state from main memory outside the server environment for a longer period of time. This is done by presenting a module for persisting state data to storage media outside the server environment. More precisely, this has been done by modifying the Vaadin framework to incorporate a session manager that changes the default internal session management to allow long time session persistence. The goal is to research if it is possible to change Vaadin in such a way that the complete state of clients can be persisted completely outside the server environment to other storage medias. The chosen storage medias were the *Amazon Simple Storage Service* (S3) and the HTML5 *Local Storage*.

Thesis structure

This thesis will start of by presenting the necessary background information in order to understand the concepts of this thesis. The first Chapter includes a closer look and definition of what constitutes a web application and state in a web application. Also presented are the different technologies used in this thesis, namely the *Amazon Simple Storage Service* (S3) and *Local Storage*. The Chapter will then finish off by explaining serialization, since it is a big part of network programming.

The second Chapter of this thesis will take a closer look into the internal workings of the Vaadin framework. The application lifecycle of Vaadin applications will be explained, along with Vaadin session management. The session management part includes a presentation of what the actual state of clients is in a Vaadin environment.

The third Chapter is the design Chapter of the developed module. The Chapter presents the requirements of the module, along with how the work was partitioned. Then the actual design for the module is presented. After this, the implementation of the module is presented in Chapter 4.

Chapter 5 presents the results of time and size profiling done for the module. The module is also evaluated in the Chapter. Based on the evaluation, the Chapter presents possible optimization proposals and future work needed. The thesis will then round off in Chapter 6 with conclusions.

CHAPTER TWO

BACKGROUND

This Chapter will present the necessary background for the rest of this thesis. The Chapter starts off by addressing web applications, their underlying concepts and technologies, so that the reader can better understand the meaning of state. After this, the Chapter will go through the main technologies used in this thesis, namely *Amazon Simple Storage Service* (S3) and HTML5 *Local Storage*, and finally finish off by explaining serialization.

2.1 What is a Web Application

There exist many software services on the Internet that are partially-, or fully web-based. Even though the standard website dominate the Internet, there is an increasing amount of web applications being created. This is due to the increasing demand of more complex functionality that a simple website simply can't fulfill. What then is a web application? In short, a web application can be defined as a "client-server application that uses a web browser as its client program" [17]. It is important to understand the difference between a website and a web application. Even though both follow the server-client pattern in which there is a server (or multiple) that serves clients, there are notable differences. Where a website is based on static data served by the web server, a web application is much more complex. A web application is able to serve content based on different parameters, often dynamically. Furthermore, a web application can track user behavior, making a web application more similar to a traditional desktop application rather than a website. This is ultimately what web applications are striving to become, namely to be as similar to a desktop application as possible [17]. This of course has proven to be quite a challenge, since there are vast differences in the underlying foundation of how web applications and desktop applications work.

Both web applications and traditional applications have their respective advantages and disadvantages. Web applications have an advantage in distribution, since they can cater the needs of many clients at the same time, without having to install software explicitly on the client machine [9]. This makes web applications very scalable and easy to maintain, since it removes the overhead of supporting multiple platforms. Unfortunately for web applications, scalability is an issue, since there will always be more restricted resources in such an environment. From a developers standpoint a desktop application might be easier to implement in the sense that there will always be more resources like computing power, memory and storage available, whereas this is not the case when developing web applications.

2.2 The Stateless Nature of the HTTP-Protocol

Even though there are many different technologies available for web application development, there are some aspects that are mutual for all. Luckily for a web developer, the whole system of the web has been built on a solid foundation of Internet protocols and services, creatively called the Internet Protocol Suite [17]. The Internet Protocol Suite is a model consisting of different layers as shown in figure 2.1. The different layers are separated by abstraction from each other, meaning that a developer does not have to learn every underlying layer to create web applications, since there are ready-to-use software taking care of many tasks. The most important protocol a web-developer should have knowledge about is the Hypertext Transfer Protocol (HTTP), which resides in the application layer. Today the HTTP-protocol is the foundation of modern websites and is classified as the transport mechanism of all World Wide Web traffic [17].

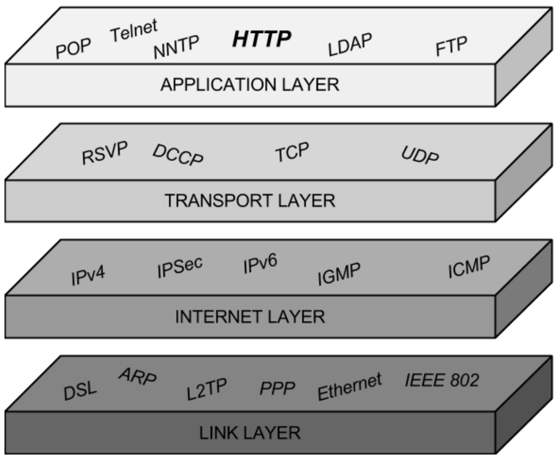


Figure 2.1: The layers of the Internet Protocol Suite

HTTP runs on top of TCP/IP, which is responsible for the communication of data between nodes in a network and makes sure that data is transmitted correctly. HTTP, an IETF (Internet Engineering Task Force) standard [10], is a request-response protocol, meaning that clients send requests through a web browser to a web server asking for content, based on some parameters and the server responds with content, which is displayed to the client. It is important to know that HTTP is comprised of single requests with corresponding responses at one instance of time without memorizing previous communication. This means that HTTP is completely stateless.

HTTP does not maintain state between requests, meaning that all the requests and corresponding responses are handled in isolation [17]. When a client sends requests, each and every one is treated as a new request. This means that the client has to include all parameters in every request. This becomes problematic when a web application needs to carry out operations spanning over multiple requests [15]. If this is the case, then all the data from previous requests has to somehow be stored, so that it can be sent back to the server at each request. This has been, and still is, done by saving some variables in the clients memory, e.g. using cookies or hidden forms. Unfortunately, this introduces scalability, performance and security issues as the complexity of the application grows. There are also drawbacks in security, since everything must be saved client side, not to mention the huge overhead created by including all the data in each request. Luckily today it is possible for a web application to store this data server side [19].

2.3 Stateful Web Applications

To address the statelessness of the HTTP-protocol, other protocols have been developed that allow support for keeping requests and responses in memory. FTP, SMTP and POP are examples of stateful protocols that allow a server to keep the state in memory throughout a series of requests until the interaction is terminated. The early implementations of stateful protocols and applications over a network were not done by use of open protocols. This meant that even though the client-server architecture was used, it was not as it is today. In most cases the client software had to be as heavy as the server software, nullifying the benefits of modern web applications. Luckily today most web applications are created on top of TCP/IP utilizing HTTP-protocols that are included in all the more popular operating systems, meaning that the client is a common web browser.[17]

2.4 State in a Web Application

Due to the increasing demand of more complex functionality in modern web applications, the technologies used in their development have been moving towards incorporating possibilities only previously provided by technologies used in development of applications running on a native operating system. This has proven demanding, since the underlying foundation of technologies that web applications are built upon are quite restrictive.

A web application, in contrast to an application running on a native operating system, is developed to cater many different clients at the same time, therefore sharing resources. The HTTP-protocol, which the web is largely built upon, is by design completely stateless, which poses a problem when state is required to be kept between requests and responses made between a server and a client. In a simple web site there is no need to keep the state of a client in memory, but in a web site where complex operations is to be supported, keeping track of state becomes a must. The definition of state could be summarized as the sequence of requests and corresponding responses, or in other words commands associated with a interaction between a client and a server. In more layman terms a session could be described as a conversation between a server and a client where everything is remembered as long as the conversation is alive[19][17]. Having the possibility to keep track of this conversation not only helps the development of web applications, but also allows for a better user experience.

From a developers point of view there are many technologies that gives opportunity to develop stateful web applications. For example in a Java based environment there are mechanisms in place to keep track of state, which will be presented later on in this thesis. The challenge with state though, is that it has to be saved as a variable (also called session) in memory. As long as the hosting server has enough memory there is no problem keeping the state in memory. Memory though is a finite resource. When the web application scales and the number of connected clients rise, the capacity for keeping state in memory will come to a critical point. This is why sessions have to be managed. Usually this means giving clients states a lifetime, usually measured in minutes. When the conversation between a server and client has been idle for a matter of minutes the session is discarded from memory. This can result in loss of data that can dissatisfy users and be costly in various ways.

2.5 The Vaadin Framework

Vaadin is a open-source framework for developing Rich Internet Applications. The framework supports both a server-side and client side programming model. The fact that Vaadin offers a server-sided programming model means that it allows development of web applications as any normal application only using Java. The client-side part of a Vaadin application uses Google Web Toolkit and HTML5, while the business logic resides on the server. Vaadin has a thriving community and has been growing rapidly over the last years both in adapting users and as a company. [1]

2.6 Amazon S3

Amazon Simple Storage Service, also simplified as Amazon S3, is a web service for storing data in a scalable way in the cloud. As the name suggests the owner of the service is Amazon, who has created the service to provide a scalable, secure and reliable service to store any kind of data in any amounts, at any time and from anywhere on the web. Amazon states that the service is "designed to make web-scale computing easier for developers" [5]. This has been made possible by an undisclosed implementation giving a minimal working set of features for the developer, instead making the service as easy to use as possible.

The service is based on a bucket/object pattern, meaning that any kind of object, ranging from 1 byte to 5 terabytes in size, can be stored in something called a bucket. The developer can assign keys and identifiers to both buckets and objects, and buckets can be configured to be physically stored in several geographical regions to maximize performance by minimizing latency. There are multiple options for securely downloading/uploading objects made possible by both encryption and authentication mechanisms that ensures the integrity of the access to objects in the buckets located in the cloud. Amazon S3 uses standard REST and SOAP interfaces, meaning that the service is possible to use with any Internet development kit. Amazon S3 provides a good API for Java, including documentation for the service. Amazon S3 can be governed both by code or from the Amazon S3 Management Console [5][15][4].

A closer look on how to use the Amazon S3 service will be shown later when presenting the Amazon S3 persistence manager.

2.7 HTML5-LocalStorage

One of the areas where applications running on a native operating system has held an advantage over web applications, is having persistent local storage. Probably the most conventional and earliest inventions to provide client-side local storage in web applications has been the introduction of Cookies. Cookies allow storing data in a clients web browser in small amounts, but have shown to have a number of downsides, especially when developing modern web applications. Cookies are limited to about 4 KB of storage space, which by modern standards is not that much to work with. Furthermore Cookies are included in every request made between the client and server even though they are not necessarily used. Also the Cookies are sent un-encrypted, introducing potential security issues. To address the limitations of Cookies the HTML5 standard [7] has introduced a storage media at the client side named local storage or also known as Web Storage or Dom Storage [16]. In this thesis the name local storage is used.

The local storage allows data to be persisted at the client and works according to a simple key/value principle [2]. This means that the data must be in a string format and is currently limited by most popular browsers to 5 MB. To be able to use the local storage a clients browser must support HTML5, which most of the popular web browsers do. Also, since the local storage is accessed by *JavaScript*, a clients web browser must allow *JavaScript*. As with Cookies, the local storage is fully persistent, meaning that even if the connection is terminated between a client and server the data will remain in memory.

Local storage was chosen for the persistence module presented in this thesis for its ability to store a considerable amount of data and persist it over a long period of time. Furthermore, since the local storage is on the client side this could allow long-term session persistence with a minimal cost to a web application provider. This also introduces the problem of integrity, since the storage is on the client side and can be easily manipulated.

2.8 Serialization

Serialization is a fundamental part of networking programming. Serialization can be said to be the process where objects are converted, along with structure and state, into streams of bytes that can be transmitted over a network or stored to some media, and at a later point converted back to its original form, which is known as de-serialization.[8]

Serialization in Java has been from the start made easy by the creation of the `java.io.Serializable` interface. An object only has to implement this interface, which then makes it possible to be serialized to for example a file or a memory buffer. Usually when an object is to be transmitted or saved to a database it is serialized to bytes, but it is also common to serialize objects into files. A key part of serialization in Java is that the process serializes the complete object, meaning that all variables and child-objects are also serialized. This means that every object in another object has to implement the `java.io.Serializable` interface for the parent object to be serializable [8].

In the Vaadin framework all the classes implement the `Serializable` interface. This is to enable Vaadin to be as versatile as possible for developers, along with enabling clustering and cloud computing, such as Google App Engine usage. [1]

2.8.1 Kryo Serialization

Even though serialization has been made easy in Java, there are some drawbacks. As explained the default serialization of Java contains all the data of an object and corresponding objects reachable by the object. This can be considered inefficient since the ideal serialized form of an object should contain only the logical data of said object. To achieve the best performance of serialization one should implement custom serialization methods. [8]

Writing custom serialization methods for a framework like Vaadin is an enormous task, since this should be done for every object in the framework. Instead an alternative serialization mechanism was investigated in part of the development done in this thesis, namely Kryo.

Kryo is an open source serialization framework for Java with efficiency and speed as goals [11]. Kryo, in contrast to Java serialization, does not care what data objects contain and does not enforce any scheme. The framework provides a number of serializers, which provides efficient implementations for serializing Java objects. Kryo started as a project in 2009 and is still in active development by a small development team.

During the development of the proposed module in this thesis, the Kryo framework was used as an alternative to default Java Serialization to investigate whether the serialization process could achieve better performance. Unfortunately, as Vaadin released an update (7.1) the implementation of serializing state using Kryo broke and is therefore excluded from this thesis.

CHAPTER

THREE

STATE AND SESSION MANAGEMENT

This Chapter will go through Java web application development with the focus on session management. This starts off with a brief look into the basic architecture behind Java based web applications. After that the Tomcat servlet container is presented, along with the topic of session management. The Chapter will end by more specifically addressing the Vaadin aspect of it all.

3.1 Web Application Architecture in a Java Based Environment

Even though the framework Vaadin was the main technology used for the development in part of this thesis, it is important to understand the architecture of Java based web applications, since Vaadin is built upon this architecture. Even though Vaadin incorporates many different technologies, the base is built upon the Java Servlet API (`javax.servlet`), which is a part of the Java Enterprise Edition (EE) platform by Oracle [1].

When developing web applications with Java servlets, be it as part of a framework or not, there is a certain structure that needs to be followed, which is important to understand, since it greatly affects session management. The *servlet* class is the main class that is used to configure and extend the capabilities of a web server [6]. By using this class, a developer is able to handle requests. The important thing to understand is that a *servlet* does not run by itself, but needs a web container (also known as servlet container) that instantiates the *servlet*. The most used servlet container, and the one which will be used throughout this thesis, is the Tomcat Servlet Container by Apache. The Tomcat Container basically houses a web application intercepting raw HTTP-Request and passing them on to correct servlets. This is illustrated in Figure 3.1. Tomcat is also responsible for handling sessions by default [19]. When wanting to manipulate how sessions are created and maintained, one has to have knowledge of how Tomcat does this by default.

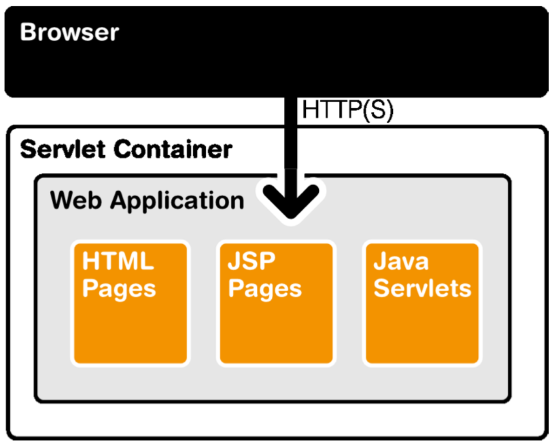


Figure 3.1: Illustration of using Java servlets in web applications [1]

3.2 A Look Into HTTP-Sessions

To keep track of the state of clients in a Java web application the, `HttpSession` interface is used [13]. The interface is part of the `javax.serlvet.http` package and provides the web container with a way to keep a session variable in memory. It is important to understand that it is the web container that implements this interface. This means that it is the web container that is responsible for the session management, including the creation, maintenance and destroying of said sessions. One of the more popular web containers recommended by Vaadin and used in this thesis, is the Apache Tomcat web container [1]. A web container is also called a servlet container.

The `HttpSession` interface implementation is responsible for handling client identification. This allows a web-developer access to the session variable from a servlet, giving access to information regarding the session and more importantly allowing the binding of objects to sessions. Binding objects to sessions is basically what makes developing stateful web applications with Java possible, since objects can be persisted over multiple requests even though the HTTP-Protocol is used. Binding objects to sessions is done by implementing the `HttpSessionBindingListener` interface. When an object does this, the object is notified when it is bound to or unbound from a session, in other words when a session is created or destroyed. This is important to understand, since it is basically what the Vaadin framework does. The framework binds data to the standard session variable by use of the wrapper pattern. The actual session object in Vaadin is named `VaadinSession` and will be further explained later on in this thesis.

3.3 The Servlet Container Tomcat

As previously mentioned, it is the servlet container that implements the `javax.servlet.http.HttpSession` interface, therefore being responsible for session management in a web application. Tomcat has implemented the interface and does create, destroy and maintain the state of clients. The state of a client is stored by default into main memory and each session object is given a unique identifier. The unique identifier is used to identify a clients state data. This means that the actual state data will always reside server-side and it is only the unique identifier that will be passed along and stored at the client. The unique identifier is by default stored in a cookie, but it can be modified to be included in the url or hidden forms. The identifier is by default named `JSESSIONID` by Tomcat, but this functionality can be overridden if needed. [19]

The fact that all state data for clients resides server-side, is what makes Tomcat a good design. Previous methods of storing a users state has e.g. involved storing the actual data client side in cookies. Storing data client side is a bad design, since this means that the data can be viewed, and even modified, by the client. By only storing the textual representation of the unique identifier client side, the client can not access the actual data, since it resides in the servers memory. Furthermore, this makes the web application more efficient when connectivity is an issue, since the state data does not need to be included in every request. On the other hand this can be seen as a drawback, since the need for increased memory for a server is greater, since all the state data will be stored in memory not utilizing the maximal resources of the client. This is a part of what this thesis tries to solve, namely to give an alternative to storing the state of clients server-side.

The internal workings of how Tomcat has implemented session management is not important in the scope of this thesis, but it is noteworthy that Tomcat allows persisting session data not only to main memory. It is possible to configure Tomcat to use storage media for sessions like standard hard drives or databases, the latter through a JBDC connection adapter [19]. It was thought of to develop an extension to Tomcat but was not chosen, since it was deemed easier to access the state data of Vaadin through modifying Vaadin itself.

3.4 State in Vaadin

Since the state is the conversation between the server and a client using the web application, the state is very different depending on the conversation and the web application itself. Arguably the state in web applications are similar if the Tomcat servlet container is used, but this is only if the Tomcat implementation is not extended, which often is not the case. For example the Vaadin framework has extended the session object to contain a lot of data. The following section will go through the Vaadin application lifecycle, what state is and how it is handled in Vaadin.

3.4.1 Vaadin Applicaton Lifecycle

Before being able to change Vaadin session management one has to understand the basic lifecycle of Vaadin Applications. Since Vaadin is a framework for creating Java based web applications, the lifecycle of a Vaadin application is quite similar to that of a web application developed with the Java Servlet API. This means the use of servlets.

Deploying a Vaadin application means the usage of a Java web server, which is the web container, also called a servlet container. The default implementation of the servlet used with Vaadin is named `VaadinServlet`. This thesis will not go into Portlets, but it is possible to create Portlets using Vaadin, then using the `VaadinPortlet` class. The `VaadinServlet` class receives all the mapped requests defined in the deployment descriptor *web.xml* and takes actions based on the request. The actual implementation of the actions that should be taken by the servlets is handled by a service class called `VaadinService`. This in fact, as will be shown later on in this thesis, is what is overridden to define custom functionality in regards to session management.

When a client makes requests to a Vaadin application, the requests are mapped to the servlet defined in the deployment descriptor. Handling request is at the time of writing this thesis undergoing a change. Previously request were identified using an enum called `RequestType`, which now is deprecated. The current way to handle request is done by implementing the `RequestHandler` interface. In the scope of this thesis though, it is important to understand that there are many different requests in Vaadin. Depending on what request has been made by a client, Vaadin takes different action. The requests that are relevant in this thesis are `BROWSER_DETAILS`-requests and `UIDL`-requests. The `BROWSER_DETAILS`-request is made when a client connects to a Vaadin application. When this happens a session for the client is created if one does not still reside in main memory. `UIDL`-requests are clients interacting with a Vaadin application. These request are state changing requests. Both of the mentioned requests are handled by different `RequestHandlers`.

How then is the Vaadin application lifecycle from the clients perspective? When a client connects to a Vaadin application, a `BROWSER_DETAILS`-request is made. At this point the server identifies the request and a loader page is sent to the client. Included in the loader page is a *JavaScript* file named *vaadinBootstrap.js* that handles loading the widget set. This happens by `XMLHTTPRequests` asynchronously. The loader page sends information about the client so that the actual application content can be loaded. This data contains for example the `JSESSIONID` cookie so the server can check whether a client already has an active session. Basically the data contains all needed client specific data so that the Vaadin application can be loaded correctly.

3.4.2 Vaadin Session Management

Another thing that one has to have knowledge about before changing Vaadin session management is how this is done by default. This means knowing when session objects for clients are created, when the state is modified and destroyed. As per Vaadin application lifecycle, a session object is assigned each client if one does not already exist in memory. A session object is active as long as the client is active. The session object in Vaadin is represented by the `VaadinSession` class [1]. The actual class will be handled later on in this Chapter.

When, where and how then are `VaadinSession` objects created? After the loader page is loaded and the `BROWSER_DETAILS`-request is made, a process is started. The servlet container creates a thread to process the request by passing it to the default servlet implementation. The default servlet implementation in Vaadin is done by the `VaadinServletServlet` class. Even though request are executed on different threads, they are all mapped to this single servlet. When a servlet gets a request, the `service` method is called. Vaadin contains a separate class for the service implementation, namely the `VaadinServletService` class. This class will handle the request and make the appropriate calls to either create a session for the client or find an existing from memory.

The stack diagram in figure 3.2 shows the execution path of the service class. The `handleRequest` method is called from the servlet and is therefore on the top. The `handleRequest` then calls the `findVaadinSession` method to find out the session related to the request. The `findVaadinSession` methods in turn calls `findOrCreateVaadinSession`, which in turn handles the locking of the sessions so that two threads cannot access and modify the same session. This means that at this point the Tomcat session will be locked. After this the `doFindOrCreateVaadinSession` is called. This method checks that the current thread actually has the lock and is permitted to access the `VaadinSession` object associated with the request and service. Lastly, the `getExistingSession` is called, which actually fetches the `VaadinSession` object from memory, or creates a new one if one does not exist. When all this is done there will exist a `VaadinSession` object in memory for the client that made the request.

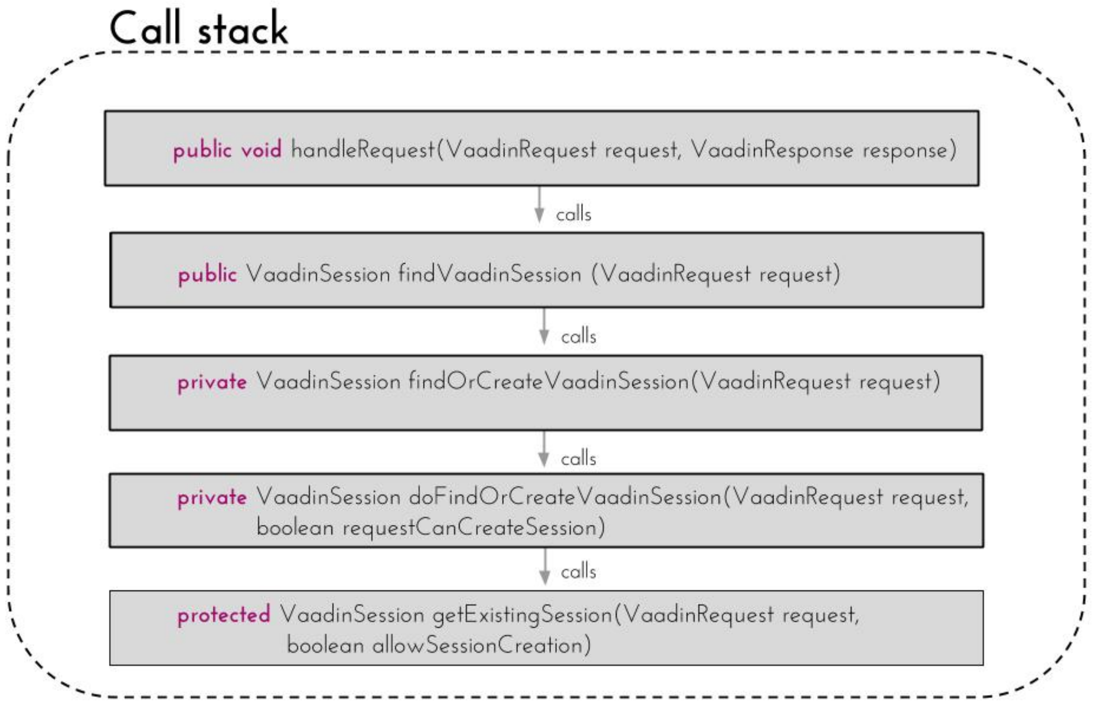

Call stack
public void handleRequest(VaadinRequest request, VaadinResponse response)
calls
public VaadinSession findVaadinSession (VaadinRequest request)
calls
private VaadinSession findOrCreateVaadinSession(VaadinRequest request)
calls
private VaadinSession doFindOrCreateVaadinSession(VaadinRequest request, boolean requestCanCreateSession)
calls
protected VaadinSession getExistingSession(VaadinRequest request, boolean allowSessionCreation)

Figure 3.2: Call stack of the servlet service

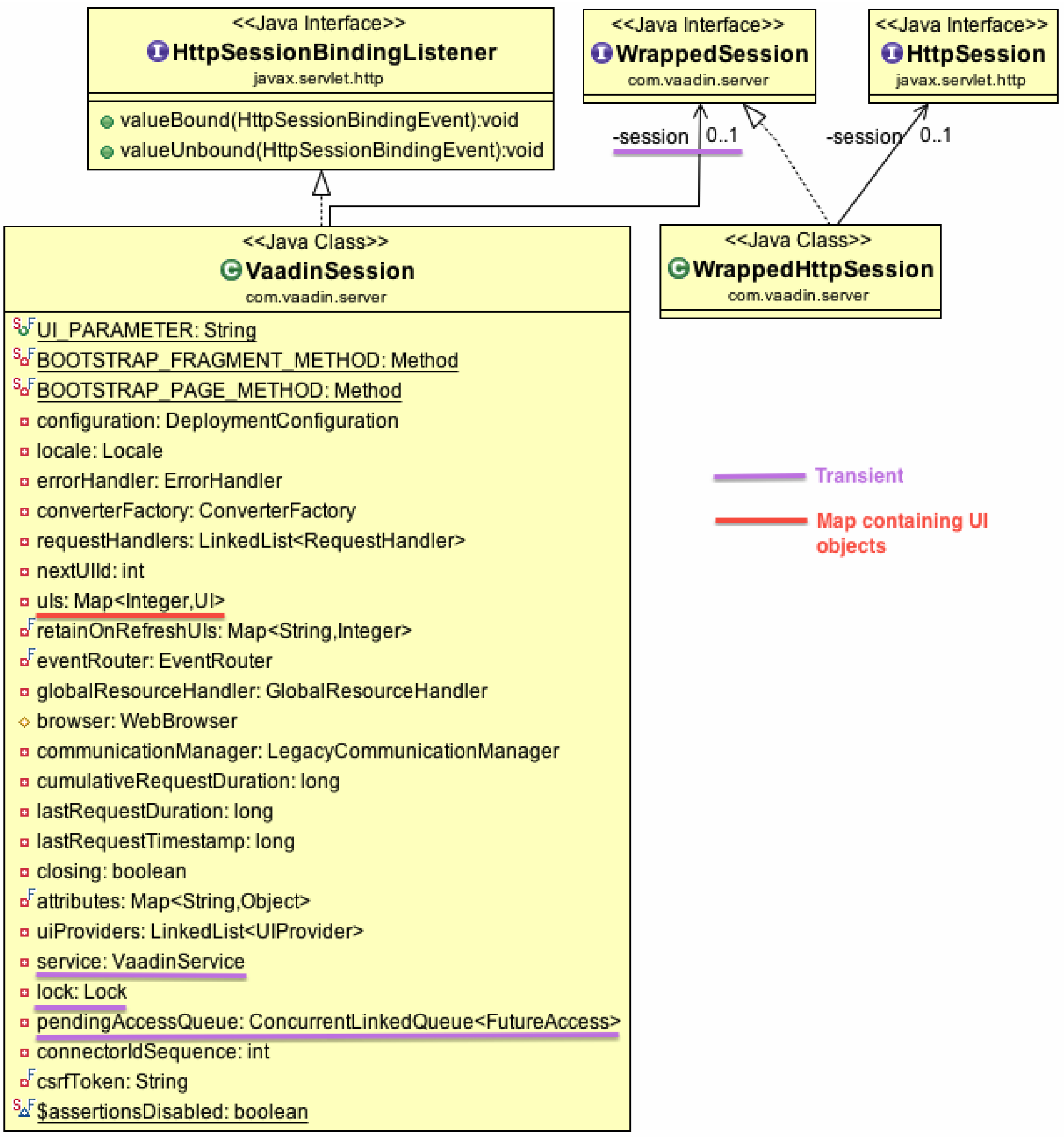
3.4.3 The VaadinSession Object

In the Vaadin framework sessions are represented by the `VaadinSession` class. The `VaadinSession` class implements the interface

`HttpSessionBindingListener` , meaning that the object is bound to the default session of Tomcat. The `VaadinSession` class contains a lot of data. Figure 3.3 shows everything that a `VaadinSession` class contains. Since this is the session for a client, it is where developers can store client specific data. The `VaadinSession` class is the hard state of a client containing all information regarding the clients state.

This thesis will not go into to much detail of the individual variables in the `VaadinSession` class. However, there are some variables that are important to note, since they play a big role in the inner workings of Vaadin session management. Figure 3.3 illustrates that a session object contains a *HashMap* consisting of `UI` objects. The `UI` class is the main class that a developer overrides to create a Vaadin application. The `UI` class is different for each application. It is the `UI` class that is shown to clients in the browser. Every tab or window is represented by its own `UI` object and is therefore included in the `VaadinSession` object. The other variables that are noteworthy are the transient variables `WrappedSession`, `Lock` and `VaadinService`. Since these are transient, they are non-serializable. The `VaadinService` variable is the service associated with the session. The `WrappedSession` is a wrapper object for the default state that Tomcat handles. The `Lock` object is used for protecting the data from concurrent access so that multiple threads can't modify the session at the same time.

As mentioned, the contents of a `VaadinSession` object varies depending on the actual application and the request made by the client. This means that the object can theoretically be of any size, especially since all the `UI` objects of a client are included. Since every opened window or tab of a client is a `UI` object, the session object can quickly grow to an unmanageable size, especially when wanting to persist sessions a long time. To limit the scope of this thesis, the optimization of state has been excluded, since it is a complex task in itself and would hence be a candidate for future work.



<<Java Interface>>
HttpSessionBindingListener
javax.servlet.http
valueBound(HttpSessionBindingEvent):void
valueUnbound(HttpSessionBindingEvent):void
<<Java Interface>>
WrappedSession
com.vaadin.server
<<Java Interface>>
HttpSession
javax.servlet.http
-session
0..1
-session
0..1
<<Java Class>>
VaadinSession
com.vaadin.server
UI_PARAMETER: String
BOOTSTRAP_FRAGMENT_METHOD: Method
BOOTSTRAP_PAGE_METHOD: Method
configuration: DeploymentConfiguration
locale: Locale
errorHandler: ErrorHandler
converterFactory: ConverterFactory
requestHandlers: LinkedList<RequestHandler>
nextUIId: int
uIs: Map<Integer,UI>
retainOnRefreshUIs: Map<String,Integer>
eventRouter: EventRouter
globalResourceHandler: GlobalResourceHandler
browser: WebBrowser
communicationManager: LegacyCommunicationManager
cumulativeRequestDuration: long
lastRequestDuration: long
lastRequestTimestamp: long
closing: boolean
attributes: Map<String,Object>
uiProviders: LinkedList<UIProvider>
service: VaadinService
lock: Lock
pendingAccessQueue: ConcurrentLinkedQueue<FutureAccess>
connectorIdSequence: int
csrfToken: String
$assertionsDisabled: boolean
<<Java Class>>
WrappedHttpSession
com.vaadin.server
Transient
Map containing UI objects

Figure 3.3: The Vaadin Session class structure

CHAPTER **FOUR**

DESIGN

To research whether it is possible to persist the state of clients outside the server environment effectively, a module was created and named session persistence manager. This Chapter presents the goals and requirements that was set for developing such a module. This is followed by the design for persisting the state of clients to 2 different storage medias.

4.1 Requirements

In the previous chapters the technological aspect of what a state is was presented along with how the state is implemented in the Vaadin framework. The main goal of this thesis is to research if the state of clients could be persisted for a longer period of time in a Java based server-sided web application and what tasks this consists of. To achieve this, a module for the Vaadin Framework was developed, which was named session persistence manager. Notably this module was developed for evaluation purposes and not to be used as is. Though this is the case, it could serve as a good starting point for a module to be developed to be used in an actual production environment. To start with some basic requirements were defined for the module and are shown in table 4.1.

Requirements
1. The module should be able to persist the hard state of a client.
2. The module should work with any Vaadin application without restricting developers.
3. The state of a client should be persisted without requiring anything from the client.
4. The module should be easily configured.
5. The module should be easily modified and extended.
6. The module should be developed so that every change in state triggers the process of saving the state.

Table 4.1: Table of requirements for the developed module

The defined requirements resulted in the identification of the following main tasks into which the development work was partitioned.

- Getting access to the state of a client, more precisely the `VaadinSession` object
- Getting access by code to the storage media to be used
- Being able to store and retrieve the state of a client to and from a storage media

Getting access to the state of a client consists of modifying the functionality regarding internal session management of Vaadin. To be able to persist the session object of clients, the persistence manager has to be able to access the session object `VaadinSession` for clients. The internal session management should be modified to check if a saved session resides in a storage media before creating a new one for a client. Both storing and loading of sessions from storage media requires that it can be accessed by code. Depending on the storage media there usually are different requirements for allowed format of data to be stored. As previously explained one has to transform the session objects before they can be moved out from memory to some other location. This includes serialization of objects into a format that can be transported over a link. Even though the solution is not optimal, the manager should store the session every time it is modified, since this is the worst-case scenario regarding performance.

4.2 Overall Design

As mentioned in the previous Section, the development work was partitioned to better understand the tasks to be done. One big part was defined to be to make changes to the internal session management of the Vaadin framework. This means modifying the behavior of how a server running Vaadin handles sessions. The other part was to develop the actual manager module that is responsible for making the connection to storage medias and store and retrieve session from said medias.

The design of the persistence manager is quite simple. The manager should receive the session objects and then make the connection to a storage media. Depending on which request has been made, the manager should either store or retrieve the session object from and to some storage media. The two different storage medias chosen were the *Amazon Simple Storage Service* S3 and HTML5 *Local Storage*. Each of them are very different in nature and both have their advantages and disadvantages. Where the S3 excels in reliability and security it is after all a service which one has to allocate monetary resources for, whereas the use of HTML5 *Local Storage* could be deemed free storage. The trade of using local storage is that there are considerable measures needed to be taken to ensure the integrity of the data, sine it resides at the client.

The development of the manager is only a part of the overall task. Vaadin is not by default meant to enable the re-initialization of sessions. This means that the internal session management has to be modified not only to store sessions, but also to re-initialize persisted session objects back into memory. As per requirements, the Vaadin framework has to be modified so that every time a `UIDL` request is recieved the state is sent to the manager to be stored. Every time a `BROWSER_DETAILS` request is received the framework should be modified to ask the manager if a session can be located for the client. To ease the identification of sessions the default identifier by Tomcat was used for identifying session when storing and retrieving them. If the manager returns a session object it is to be placed back into memory and the state of the client should be re-initialized. This means that the `UI` object should be re-fired and sent to the client instead of creating a new instance.

4.2.1 Amazon S3 as Storage Media

Figure 4.1 illustrates how the Amazon S3 instance was designed to work with the Framework to handle sessions. The client makes requests to the web server and gets responses as normal operation. The web server has a custom servlet and service which passes along the `VaadinSession` objects to the session persistence manager. The session persistence manager initializes and makes a connection to the S3 instance. Once an connection has been made the `VaadinSession` object is serialized into a format that can be transported and stored in S3. As explained, the S3 works according to a simple bucket/object pattern, meaning that the session can simply be serialized to a file. This is per requirements done every state changing request. When the client reconnects to the web server the new servlet service checks the S3 if a session has been stored. If that is the case, the session is downloaded from S3 and de-serialized back into `VaadinSession` format and read into the servers memory.

As evident from figure 4.1, the storage media is completely separated from the web server when using the amazon S3. This means that all the handling of the stored states of clients can be handled not by the web server, but by the Amazon S3 instance. As previously explained, an S3 bucket can be managed from the web interface. This interface allows the definition of for example how long sessions should be stored among other things. If such a persistence module as proposed in this thesis would be taken into use, the AWS management console would allow easy configuration of the lifetime of stored session objects [3].

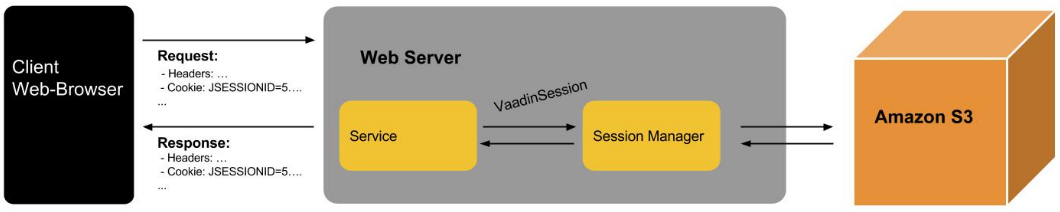


Figure 4.1: Amazon S3 storage manager design

4.2.2 Local Storage as Storage Media

Figure 4.2 illustrates the design of using the HTML5 local storage as the storage media for the state of clients. As in any web application the client makes requests to the web server and receives responses. Where the use of Amazon S3 was clearly a separate module from the web server, using the local storage is a bit more complex. Each time a client connects to a web application the stored session data should be somehow included so the server can persist the state of a client before creating a new session variable in memory as per normal operations. Then when a state changing request is made the session data should be serialized into a form compatible with local storage, more precisely string format. Already at this point a problem is evident regarding localization. It was defined that the `String` representation of a session should be transformed into a format that does not produce Unicode errors. `Base64` format was chosen.

As in the S3 design the web server environment, Vaadin, was modified to access the `VaadinSession` objects. The session persistence manager handles the serialization and de-serialization operations along with needed conversions and passes said sessions back to the framework. By default

Vaadin does not support the usage of HTML5 local storage. The connection between the web server and local storage of a client was achieved by using the `ClientStorage` add-on developed by Henrik Paul [14] with a few modifications making it fully serializable.

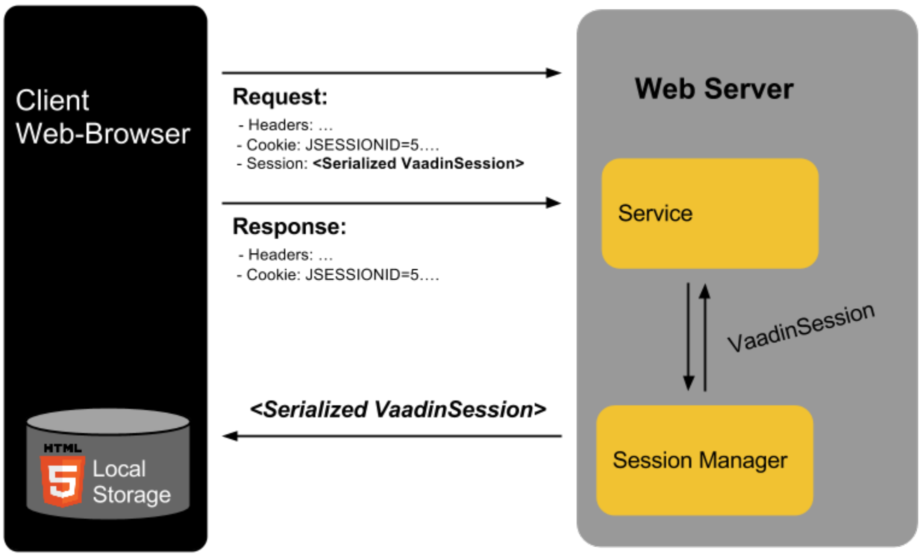


Figure 4.2: HTML5 Local Storage storage manager design

4.3 Configuring the Session Persistence Manager Module

One of the requirements of the developed module was ease of use. This was achieved by allowing configuration of the module from the deployment descriptor *web.xml*. As previously explained, the use of the developed module requires the use of a custom servlet. This is configured by passing the servlet class as the *<servlet-class>* property. The servlet implementation for using the session persistence manager was named `PersistenceStorageServlet`. As for configuring storage medias, an init parameter by the name of *SessionStorage* is used. To the parameter the implementation of the session manager for a storage media is passed. For S3 this corresponds to `AmazonS3SessionStorageManager` and for using local storage `HTML5LocalStorageSessionStorageManager`. When using

S3 as storage media there are other parameters that needs to be configured. These are shown in table 4.2. Furthermore, another configuration is available, namely the serialization mechanism to be used. During the development of the module presented in this thesis, another experimental serialization technology than Java was researched. This was, as previously presented, the Kryo framework. Even though it was excluded from this thesis the possibility to configure serialization mechanism was left intact, since it allows the freedom to implement one if needed.

An example deployment descriptor of using S3 is included in Appendix A.

<param-name>	**<param-value> - example**	**Description**
accesKey	*AKIAIAISAG7CFPUTZ6VQ*	The access key of an amazon account for authorization
secretKey	*pB9G2MAY2+VsG54+0fQuLIlvpCphuY2YRxY+jFA0*	The secret key of an amazon account for authorization
bucketName	*VaadinSessionPersistenceBucket*	The bucket name into which objects are stored.

Table 4.2: Configuration parameters for using S3 as storage media

CHAPTER
FIVE

IMPLEMENTATION

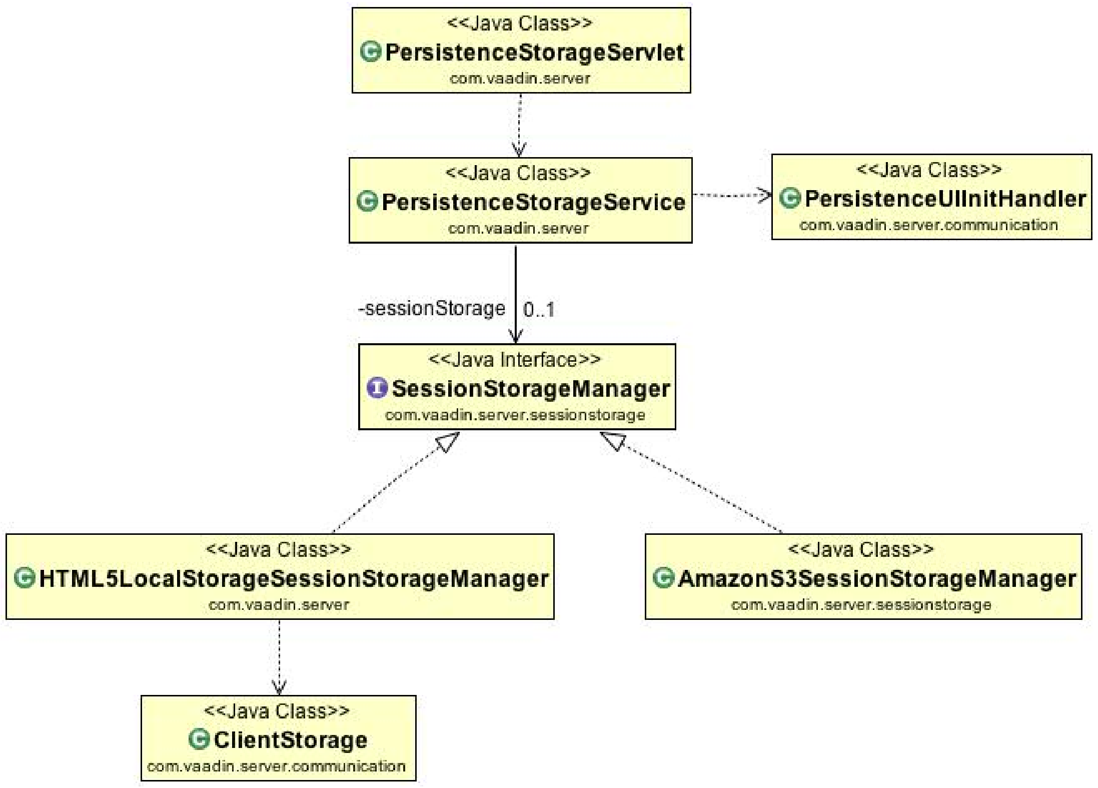
This Chapter presents the implementation of the session persistence manager module that was developed. This will be done by presenting the implementation for the actual manager that is responsible for persisting the state of clients to Amazon S3 and HTML5 local storage. This will be followed by a presentation of how the internal Vaadin session management was modified to incorporate the session persistence manager.

5.1 Implementing the Session Storage Manager

As described in the design Chapter, the development of the session persistence module was partitioned into different parts. One part was the changes needed to be implemented for the internal session management of Vaadin. The other part was the actual session persistence manager, which is responsible for actually persisting the state of clients from and to different storage medias. The class diagram in figure 5.1 shows the major classes that where implemented. The `PersistenceStorageServlet`, `PersistenceStorageService`, `PersistenceUIInithandler` are implementations that modify the session management of Vaadin. The `SessionStorageManager` interface is the actual session persistence manager and the `HTML5LocalStorageSessionStorageManager` and `AmazonS3SessionStorageManager` are the implementations of the interface for persisting sessions to and from local storage and the S3. The figure also shows that the `HTML5LocalStorageSessionManager` contains the class `ClientStorage`. This is the main class of the add-on that enables the communication between local storage and server. The add-on consist of many other classes, but since these are not that relevant, they have been excluded from the diagram. In regards of deployment there are some dependencies that have to be added to make the persistence manager implementations functional. These are added as any other dependencies to the *WEB-INF/lib* folder of the server running the Vaadin application. Table 5.1 show the dependencies used.

Library	**Used for**
aws-java-sdk-1.4.1.jar	S3
aspectjrt.jar	Local Storage
aspectjweaver.jar	Local Storage
commons-codec-1.8.jar	Local Storage
commons-logging-1.1.1.jar	Local Storage
freemarker-2.3.18.jar	Local Storage
httpclient-4.1.1.jar	Local Storage
httpcore-4.1.jar	Local Storage
jackson-core-asl-1.8.7.jar	Local Storage
jackson-mapper-asl-1.8.7.jar	Local Storage
mail-1.4.3.jar	Local Storage
spring-beans-3.0.7.jar	Local Storage
spring-context-3.0.7.jar	Local Storage
spring-core-3.0.7.jar	Local Storage
stax-1.2.0.jar	Local Storage
stax-api-1.0.1.jar	Local Storage
kryo-2.22-all.jar	Kryo

Table 5.1: Table of dependencies needed for deployment



<<Java Class>>
PersistenceStorageServlet
com.vaadin.server
<<Java Class>>
PersistenceStorageService
com.vaadin.server
<<Java Class>>
PersistenceUIInitHandler
com.vaadin.server.communication
-sessionStorage
0..1
<<Java Interface>>
SessionStorageManager
com.vaadin.server.sessionstorage
<<Java Class>>
HTML5LocalStorageSessionStorageManager
com.vaadin.server
<<Java Class>>
AmazonS3SessionStorageManager
com.vaadin.server.sessionstorage
<<Java Class>>
ClientStorage
com.vaadin.server.communication

Figure 5.1: UML class diagram representation of implemented classes

5.1.1 The SessionStorageManager Interface

The `SessionStorageManager` interface shown in listing 5.1 is the interface that can be used to implement how sessions are stored and fetched depending on the storage media. Figure 5.2 shows in more detail how the interface works with the service implementation and different implementations of the interface.

The `loadSession` method is used for fetching stored `VaadinSession` objects depending on the request (`VaadinRequest`) passed along. The request contains all necessary information needed to identify clients.The `storeSession` method is used for storing `VaadinSession` objects into some storage media. The input parameters consist of:

- `VaadinSession vaadinSession` - The complete object that is the state of the client
- `int uiId` - The integer identifier of the `UI` object from which the client has made the state changing request. This allows the correct `UI` to be

shown when the session is persisted.

- `String Identifier` - A unique string representation that is used for storing `VaadinSession` objects.
- `VaadinRequest request` - The associated request made by the client.

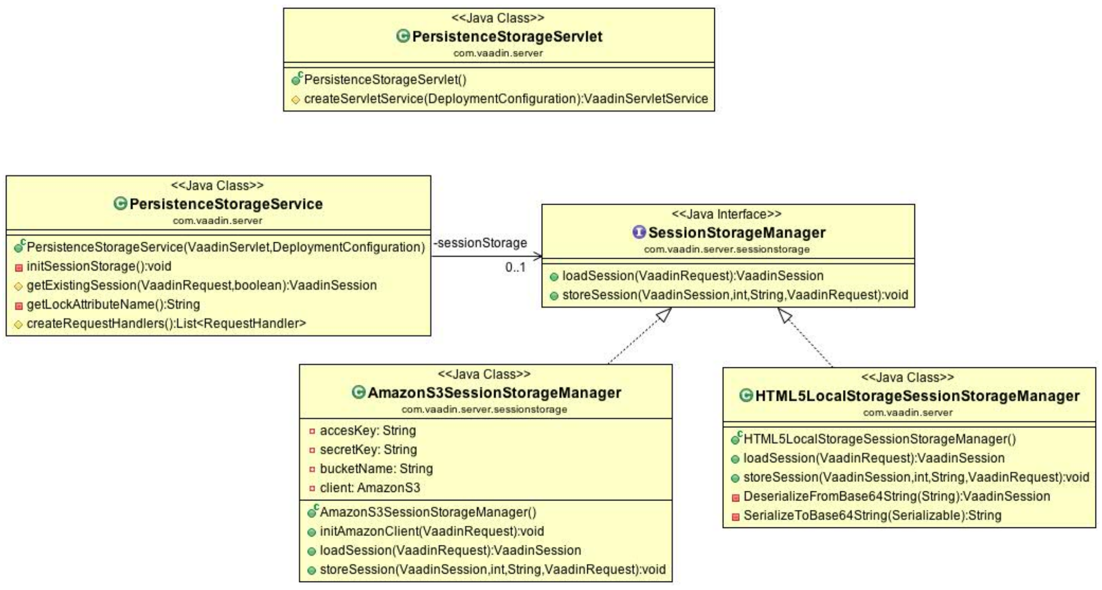
Listing 5.1: The SessionStorageManager Interface

```java
package com.vaadin.server.sessionstorage;

import java.io.Serializable;

import com.vaadin.server.VaadinRequest;
import com.vaadin.server.VaadinServletRequest;
import com.vaadin.server.VaadinSession;

public interface SessionStorage extends Serializable {
  VaadinSession loadSession(VaadinRequest request);
  void storeSession(VaadinSession vaadinSession, int uiId,
     String Identifier, VaadinRequest request);
}
```



<<Java Class>>
PersistenceStorageServlet
com.vaadin.server
PersistenceStorageServlet()
createServletService(DeploymentConfiguration):VaadinServletService
<<Java Class>>
PersistenceStorageService
com.vaadin.server
PersistenceStorageService(VaadinServlet,DeploymentConfiguration)
initSessionStorage():void
getExistingSession(VaadinRequest,boolean):VaadinSession
getLockAttributeName():String
createRequestHandlers():List<RequestHandler>
-sessionStorage
0..1
<<Java Interface>>
SessionStorageManager
com.vaadin.server.sessionstorage
loadSession(VaadinRequest):VaadinSession
storeSession(VaadinSession,int,String,VaadinRequest):void
<<Java Class>>
AmazonS3SessionStorageManager
com.vaadin.server.sessionstorage
accesKey: String
secretKey: String
bucketName: String
client: AmazonS3
AmazonS3SessionStorageManager()
initAmazonClient(VaadinRequest):void
loadSession(VaadinRequest):VaadinSession
storeSession(VaadinSession,int,String,VaadinRequest):void
<<Java Class>>
HTML5LocalStorageSessionStorageManager
com.vaadin.server
HTML5LocalStorageSessionStorageManager()
loadSession(VaadinRequest):VaadinSession
storeSession(VaadinSession,int,String,VaadinRequest):void
DeserializeFromBase64String(String):VaadinSession
SerializeToBase64String(Serializable):String

Figure 5.2: Detailed UML class diagram of central classes implemented

5.1.2 Amazon S3 Implementation

The implementation of the `SessionStorageManager` interface for the use of the Amazon S3 as the storage media requires the use of Amazon AWS SDK. The SDK can be installed as any normal SDK or as an toolkit when using Eclipse. To access an Amazon S3 instance a `AmazonS3Client` object is used. This is instantiated by giving an `BasicAWSCredentials` object as parameter, which contains the access key and secret key for the Amazon S3 service. As explained in the design Chapter these credentials are configured in the *web.xml* deployment configuration file as properties.

When the `SessionStorageManager` implementation gets a call to either store or fetch a `VaadinSession` object to S3 the `AmazonS3Client` object is instantiated, meaning that each thread (client) has an own such object. According to the Amazon S3 documentation [4], having multiple connections to an S3 instance does not affect performance of S3.

When a service makes a `storeSession` call to the `AmazonS3SessionStorageManager`, the connection to S3 is made. Since the S3 service allows storage of files the `VaadinSession` object is serialized into a temporary file. The file is named after the identifier provided from the service, which in this case is the unique identifier assigned to a client by Tomcat. Once the serialization is complete it is uploaded to the bucket configured in the deployment descriptor.

When a service makes a `loadSession` call a connection to S3 is first made. An attempt to download the file containing the `VaadinSession` object is made based on the unique identifier, which is the name of the file containing the session object. Once the file is downloaded it is de-serialized back into `VaadinSession` format and returned to the service. If for some reason the file can not be found or de-serialized the value of `null` is returned which makes the service create a new session object for the client as per normal Vaadin session management.

5.1.3 Local Storage Implementation

The implementation of using local storage as storage media for `VaadinSession` objects is a bit more complex than the Amazon S3 implementation even though the design is similar.

Vaadin does not by default incorporate functionality to enable communication with HTML5 local storage from the server. This lead to the need and usage of the `ClientStorage` add-on. The `ClientStorage` add-on is a wrapper allowing server-side access to data stored in local storage. The add-on is attached to `UI` objects as an extension. Since a `VaadinSession` object contains `UI` objects and has to be serialized to allow serialization, the add-on was modified to be fully serializable. Traditionally, a developer has full control over the `UI` objects and adds all extension themselves. In the proposed implementation the `ClientStorage` add-on is added to the `UI` object in the `storeSession` call if not already added previously. This is easily achieved since a `VaadinSession` contains the `UI` that made the call. Once done the object is serialized and converted into a `Base64 String` object and stored into the local storage memory with the identifier `v-session`.

The link between the server and local storage when storing `VaadinSession` objects is handled by the `ClientStorage` add-on. Due to it being a extension to an `UI` object this approach is not possible when wanting to access the session object when a load call is made. When a `BROWSER_DETAILS` request is received by the service there does not yet exist any session object for the client. If no session exist there can also not exist any `UI` object to add an extension to. This is why the loader page *vaadinBootstrap.js* had to be modified so that the local storage data is sent when a `BROWSER_DETAILS` request is made. The listing 5.2 shows the *JavaScript* code added to the loader page code. This way when the request reaches the service and a `loadSession` call is made the session can be accessed by looking at the request. The session is then transformed back from `Base64` representation and de-serialized into a `VaadinSession` object by the helper method `SerializeToBase64String`. By default Tomcat has a restricted header size for requests that has to be increased to allow the contents of the local storage to be added to a request. This is done by changing the connector for the mapped port for the request and adding `maxHttpHeaderSize` (E.g `maxHttpHeaderSize="5242880"`).

Listing 5.2: The added code to the loader page to include contents of Local Storage in request

```
try{
        if(('localStorage' in window && window['localStorage'
           ] !== null)==true){//Check for localStorage
           support
          url += "&v-session=" + localStorage.getItem('v-
             session'); // Add the session-data to header
        } else{
          //No native support for HTML5-LocalStorage
          url+="&v-session=" + "null";
        }
      }
      catch(error){url+="&v-session=" + "null";}
```

5.2 Implementing Changes in Vaadin

5.2.1 Modifying the ServletService

As mentioned in the design Chapter, one of the main parts of the development was modifying internal Vaadin session management. After researching the internal parts of the framework, it became evident that the best way to do this is to override the default service implementation. Vaadin does not provide any way to configure a web application to use a custom service implementation, meaning that this had to be achieved by overriding the default servlet implementation. Listing 5.3 shows the implemented `PersistenceStorageServlet` servlet that specifies that the new service implementation should be used.

Listing 5.3: The PersistenceVaadinServlet implementation

```java
package com.vaadin.server;

public class PersistenceStorageServlet extends VaadinServlet {

  protected VaadinServletService createServletService(
        DeploymentConfiguration deploymentConfiguration)
        throws ServiceException {
     VaadinServletService service = new
        PersistenceStorageService(this,
        deploymentConfiguration);
     service.init();
     return service;
  }

}
```

The actual implementation for the service class is somewhat more complex. As previously stated, the proposed session manager will be used in the `getExsistingSession` method, since this is where the actual call to fetch a session from memory or create a new one is made. The `getExsistingSession` method is the method in the service class where the actual logic resides for creating and fetching sessions.

As shown in listing 5.4, the request is identified through request parameters to a specific type so that correct action can be taken. If the parameter identifying the request is of type `BROWSER_DETAILS` it means that a `VaadinSession` should either be created or persisted from storage. A `loadSession` call to the `SessionStorageManager` object is then done, which returns a session based on a identifier. In this case the identifier is the default cookie provided by the servlet container Tomcat. Once a session object is recieved from the session manager, a check is made that the current thread is allowed to use the `VaadinSession` before finally setting the transient variables and loading the `VaadinSession` object into main memory.

As previously stated, a decision was made that the state of a client was to be persisted each time a state changing request is made, in other words an `UIDL` -request. When a `UIDL` request is received by the `getExsistingSession` method, a `storeSession` call is made to the `SessionStorageManager` object. The `VaadinSession` is passed along with necessary information for storing the session to some storage media. Finally, if the request is any other than the two mentioned, the request is handled as normal by the default service implementation `VaadinServletService`.

Listing 5.4: The modified getExistingSession metod implementation

```
protected VaadinSession getExistingSession(VaadinRequest
    request,
      boolean allowSessionCreation) throws
        SessionExpiredException {
    String uiIdRequestParameter = request.getParameter("v-uiId
        ");
    String browserDetailsParameter = request.getParameter("v-
        browserDetails");
    VaadinSession vaadinSession = null;
    if(browserDetailsParameter!=null){ // A BROWSER_DETAILS
        request has been made
      vaadinSession = sessionStorage.loadSession(request);
      if(vaadinSession!=null){
        WrappedSession wrappedSession = request.
            getWrappedSession();
        if(((ReentrantLock) wrappedSession.getAttribute(
            getLockAttributeName())).isHeldByCurrentThread()){
            //Current wrappedSession and current thread
            belongs to lock. Add lock to the VaadinSession
          wrappedSession.setAttribute(getLockAttributeName(),
               wrappedSession.getAttribute(
             getLockAttributeName()));
        }
```

```
            vaadinSession.storeInSession(getCurrent(), request.
               getWrappedSession());
            return vaadinSession;
          }
        }
        vaadinSession = super.getExistingSession(request,
           allowSessionCreation); // Let default method take care
           of fetching from RAM and creating of session
        if (vaadinSession!=null){// If vaadinSession not in memory
            check persistenceStorage, if this is null the super.
           doFindOrCreateVaadinSession will create and register a
           new Session
          if(uiIdRequestParameter!=null){ //"UIDL-request":
             sending session to sessionStorage to be serialized
             and stored
            sessionStorage.storeSession(vaadinSession, Integer.
               parseInt(uiIdRequestParameter), vaadinSession.
               getSession().getId(), request);
          }
        }
        return vaadinSession;
      }
```

5.3 Handling Persisted UI Objects

As previously stated, the `UI` class is the main class used by developers to create a Vaadin application. A clients session contains at least one `UI` object. The session can though contain several `UI` objects, since every new window or tab is a new `UI` object. Instantiating `UI` objects is done in the implementation of the `RequestHandler` interface `UIInitHandler`. There are multiple handlers depending on the requests made. When a `BROWSER_DETAILS` request is made it is the `UIInitHandler` that handles the creation and management of `UI` objects. This class was overridden by creating the class `PersistenceUIInitHandler` to re-fire an `UI` object from a persisted session object. This was done by overriding the `createRequestHandlers` method in the service implementation to instantiate a `PersistenceUIInitHandler` instead of the default `UIInitHandler`. Since the purpose of this thesis was to research the feasibility of persisting sessions outside the server environment, the `UI` object to be re-fired was defined to the first object in the HashMap containing all said objects. A proposed improvement would be to improve the session manager implementation to include the identifier of the `UI` object that is associated with the persisted session object. This way the correct `UI`, namely the one that made the `storeSession` call, could be re-fired.

CHAPTER SIX

EVALUATION OF MODULE

In this chapter the performance results of the developed module are presented and analyzed. This is followed by an evaluation of the results. The evaluation reviews the feasibility and usability based on the results and discusses possible optimization proposals and future work.

6.1 Results

In the design chapter the goals and requirements for the session management module was presented. Many of these were quite general and hard to measure due to subjectivity. Therefore, all the functional requirements were excluded. The main metrics that are of interest and examined are the size of persisted sessions and the time of the different processes that goes into persisting sessions.

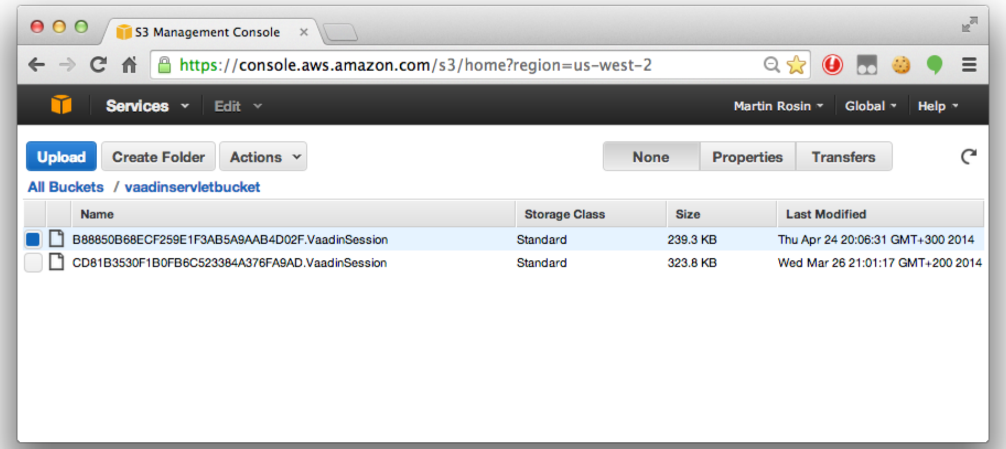
6.1.1 Data Collection Procedure

The data collection procedure was done by using the example application Addressbook provided by Vaadin [18]. This was done by running the application locally with specifications presented in table 6.1. The requests were made manually from the browser and consist of simple `UIDL` requests to trigger the storing of a sessions and `BROWSER_DETAILS` requests for loading said sessions. The time profiling was done by introducing timers for each task in the code.

The main tasks to measure were defined to be: 1) initialization of storage media, 2) serialization of `VaadinSession` objects, 3) uploading serialized objects to S3, 4)downloading serialized session objects from S3, 5) de-serialization of objects from S3. Size profiling was done by examining the size of session objects for each storage media. When using S3 this was done using the AWS management console as shown in figure 6.1. When using local storage this was done by using the developer console of the Chrome web browser as shown in figure 6.2.

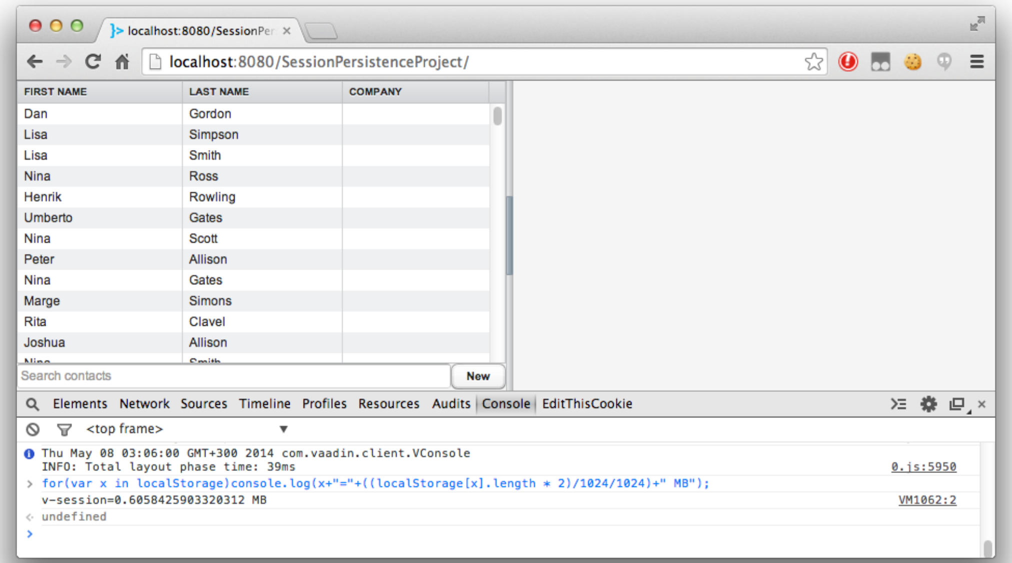
Hardware	Macbook Pro 2013 early 2011 *Processor: 2.3 GHz Intel Core i5* *Memory: 8 GB 1333 MHz DDR3* *HDD: Samsung SSD 840 Series* *OS: OS X 10.8.5*
Vaadin	Vaadin 7.1.1
Servlet Container	Apache Tomcat 7.0.39
Connectivity	Ping: 9 ms DL: 382.94 Mbps UP: 74.62 Mbps
Browser used	Google Chrome 32.0.1700.107

Table 6.1: Table of specifications used in the data collection procedure



Name	Storage Class	Size	Last Modified
B88850B68ECF259E1F3AB5A9AAB4D02F.VaadinSession	Standard	239.3 KB	Thu Apr 24 20:06:31 GMT+300 2014
CD81B3530F1B0FB6C523384A376FA9AD.VaadinSession	Standard	323.8 KB	Wed Mar 26 21:01:17 GMT+200 2014

Figure 6.1: The AWS management console



FIRST NAME	LAST NAME	COMPANY
Dan	Gordon	
Lisa	Simpson	
Lisa	Smith	
Nina	Ross	
Henrik	Rowling	
Umberto	Gates	
Nina	Scott	
Peter	Allison	
Nina	Gates	
Marge	Simons	
Rita	Clavel	
Joshua	Allison	

```
Thu May 08 03:06:00 GMT+300 2014 com.vaadin.client.VConsole
INFO: Total layout phase time: 39ms                                   0.js:5950
> for(var x in localStorage)console.log(x+"="+((localStorage[x].length * 2)/1024/1024)+" MB");
  v-session=0.6058425903320312 MB                                      VM1062:2
<- undefined
>
```

Figure 6.2: The chrome developer console

6.1.2 Results of Using S3

Table 6.2 shows the measurements gathered regarding total execution time for storing and loading. The table also shows the size of the session object in S3 after each request. Table 6.3 shows the time distribution of each of the defined tasks. From looking at the table it is clearly evident that the tasks of uploading and downloading objects to and from S3 are the most time consuming tasks. This is followed by the tasks of serialization and de-serialization. The task of initializing S3 is small in contrast to the other tasks. Notable though is that the time to initialize S3 is at the first request much greater than the rest. This is due to the thread creating the object into memory whereas the variable at later requests already exists.

Request #	Size (KB)	Total store time (ms)	Total load time (ms)
1	205.4	1664	719
2	227.2	911	697
3	235.5	4867	1089
4	236.5	886	788
5	237.5	1218	865
6	242.3	842	963
7	247.2	951	1201
8	251.2	1102	1165
9	252.6	874	896
10	253.1	872	902

Table 6.2: Size of sessions and total execution times for loading and storing when using S3

Request #	Initialization of S3 (ms)	Serialization of session (ms)	Upload object to S3 (ms)	Download object from S3 (ms)	Deserialization of session (ms)
1	457	446	754	409	310
2	7	165	739	398	299
3	12	136	4721	781	308
4	10	124	752	487	301
5	7	109	1102	576	289
6	6	117	719	661	302
7	11	121	841	897	304
8	9	122	971	853	312
9	8	135	731	597	299
10	10	109	753	601	301

Table 6.3: Specific execution times when using S3

6.1.3 Results of Using Local Storage

Table 6.4 shows the time and size measurements of storing a session to local storage. It is evident that the serialization process is much less time consuming in contrast to when using S3. This is due to the fact that the objects are serialized into `byte` arrays and then converted into `String` objects. This serialization is much more effective than serializing objects to a file. As to avoid localization problems the serialized session string is converted into `Base64` format. The table shows that this is not that time consuming. Though the serialization process is much more effective than when using S3 it is evident from the results that the stored object is of a much greater size. This is due to it being a `Base64`format `String` object. The results also show a rapid growth of the size of the object as state changing request are made. It is evident that it will, after a number of request, grow to hit the limit of local storage.

Table 6.5 shows the time measurements of loading sessions from local storage. The initialization task is not per se the task of initializing local storage, but more the task of getting the object from the request. The conversion of the `Base64 String` to a normal `String` and its de-serialization is clearly not that time consuming.

From these measurements the download and upload time has been excluded. This is due to that the testing process was made running the application locally. This thesis will not go into the effectiveness of using the `ClientStorage` add-on, used in part of the implementation of the local storage session manager. In a production environment one must remember that where the connectivity performance of using S3 is controllable it is not when using local storage. The connectivity of using local storage depends greatly on the connection of the clients, which vary tremendously.

Request #	**Size KB**	**Serialization (ms)**	**Base64 Conversion (ms)**
1	605.8	220	19
2	605.9	69	2
3	627.6	30	2
4	630.1	42	7
5	632.7	13	1
6	635.2	12	1
7	652.9	12	2
8	658.0	11	2
9	641.5	9	1
10	663.8	13	3

Table 6.4: Execution times and size results when storing sessions to Local Storage

Request #	**Initialization (ms)**	**De-Serialization (ms)**	**Conversion from Base64 (ms)**
1	130	14	41
2	658	14	25
3	218	12	34
4	125	11	22
5	138	11	22
6	148	14	31
7	256	13	28
8	295	12	28
9	220	12	20
10	246	12	21

Table 6.5: Execution times and size results when loading sessions from Local Storage

6.1.4 Results of Using Kryo Serialization

During the development of the module presented in this thesis, an alternative serialization mechanism to Java was tested. This was done by use of the Kryo framework. Even though the implementation of using Kryo broke when Vaadin was updated to version 7.1, the use of Kryo showed promising results when tested using Vaadin version 7.0.3. Since the results were promising, they are included in this Chapter as a potential optimization proposal for future work. Table 6.6 shows the results of serializing `VaadinSession` objects with default Java serialization and Kryo. From comparing the results it is evident that Kryo is considerably more efficient and could decrease the time spent serializing the state of clients.

Request #	**Java serialization execution time (ms)**	**Kryo serialization execution time (ms)**
1	267	237
2	199	146
3	131	43
4	118	36
5	113	37
6	115	36
7	124	24

Table 6.6: Execution time comparison of using Kryo versus Java serialization

6.1.5 Analysis of Results

There has not yet been defined any constraints of what is needed of the proposed session management module in terms of execution time and size management. Naturally it applies that lesser execution time and data size is better, but what is acceptable? Previously it was stated that the most optimal scenario would be that the clients would not be affected by the process of persisting their state. This means that the response times should be of an acceptable magnitude. According to the research by Nielsen [12], there are three critical time constraints for response times. The limit for a response time to fall under the category of a web application being said to be reacting instantaneously is 0.1 seconds. A users flow of thought is classified as uninterrupted if the response time is lesser than or equal to 1 second, even though the delay then is notable. The final constraint for response times is 10 seconds, which is the limit for keeping clients attentions. From results of the time profiling it is evident that for an application like Addressbook the total amount of time spent for persisting the state at each request falls in the range of acceptable limits of the flow of thought for users being uninterrupted. The results though vary and one must remember that the connection between the server and storage media is crucial and can give very varying results. Especially when using local storage if the clients connection is bad it may result in unacceptable download and upload times

The size metric is defined by the storage media used. The S3 has no constraints regarding size and will therefore theoretically never be an issue. One must though keep in mind that the upload and download times are related to size. This applies for both storage medias. When using local storage, size has a limit which is dependent on the browser used and will therefore be an issue when the session object grows in size. For most of the commonly used browsers the allowed size of the local storage is 5 MB. From the data it is evident that the state of clients do grow quite quickly and will reach the limit after a number of requests.

6.2 Evaluation of Results

This Section evaluates the developed module to ascertain if the use of such a module is a viable solution to achieve long-term session persistence. The evaluation consist of looking at the persistence session manager using S3 and local storage as storage media for the hard state of clients. Both medias are very different in nature and completely outside the server environment. The evaluation will be based on performance, reliability, cost-effectiveness and usefulness.

6.2.1 Evaluation of Using S3

In regards of performance the results show that using S3 as a storage media for persisting sessions is a possibility. Since S3 does not have any limitations in regard of size, it comes down to the execution time. The execution time when using S3 as developed in part of this thesis falls barely in range of being acceptable. Though there is a notable delay, a clients flow of thought stays barely intact when persisting the state after each time it is changed.

When comparing reliability S3 excels over local storage. The service is classified as extremely secure and reliable. If persisting the state of clients is of importance the service is truly a good option as storage media. S3 is a service requiring monetary resources and a thorough cost analysis of using it should be made to ascertain the cost versus benefits. Depending on the amount of clients and web application itself the number of requests can quickly grow to prove very expensive.

In terms of ease of use the S3 service excels. This is both from an application developers perspective as for maintainability. As shown the persistence manager is easy to use with any kind of application without compromising web application functionality. Since it is a pay-per-use service it is easy to follow and maintain. Furthermore S3 allows easy configuration in terms of session life time through the AWS management console.

Using S3 for persisting session is a viable option. During the evaluation there did though rise ideas for optimization in regards to performance and cost effectiveness. As explained in the design chapter, the developed module was designed to persist the session after each state changing request. This can fast become straining in regards of performance and cost. It is not in the scope of this thesis to research when a session should be persisted, but in regards of optimization there is a better solution. Instead of persisting the state of clients after each state changing request the state could be persisted only before it is removed from memory. The session object in a servers memory is destroyed once its lifetime is deemed at an end. It is possible to override the default implementation of the method that is called when a session is destroyed. This can be done by implementing the `HttpSessionListener` interface and overriding the `sessionDestroyed` method. A proposed improvement of the implementation done in part of this thesis would be to use the session manager in the `HttpSessionListener` implementation to persist the session before it is destroyed. This way a clients state data would be preserved by only storing it once right before it is about to be destroyed. This would mean that the number of session storing calls to S3 would not be as high as in the current implementation, therefore being more cost effective.

6.2.2 Evaluation of Using Local Storage

Even though the results presented in this thesis show that using local storage is a viable option for storing the state of clients there are a number of drawbacks. As mentioned, the performance of using local storage is very dependable on the connection capabilities of clients. This is something that is difficult to control and can pose as big problem. Before using local storage in a real production environment the impact of client connectivity should be researched.

Where S3 allowed the storing of objects of any size, local storage does not. As explained before, there is a limit to the amount of data that can be stored and with the current implementation it will be a limitation. The fact that local storage is limited to only storing the state of clients for a limited amount of request removes the viability of it being a solution to the problem of long time session persistence. This means that before actually being able to use local storage, research should be made into possible compression of the state data. This though is only a temporary solution, since it would only delay the growth of the size of the state data. Instead a proposed solution would be to develop methods to better manage the state data of clients, removing all overhead and only storing the most relevant information. This could be done by finding a way to store the soft state instead of the complete hard state.

In terms of cost the solution of storing state data of clients to local storage would be fantastic. It could be deemed free storage, since the state data would reside client side. Though in terms of reliability it would not prove as good as for instance S3. Since the data resides client side, it is prone for manipulation. Before using local storage in a production environment safeguards for tampering should be implemented. This could for instance be done by encrypting the data. Another factor hurting the reliability is that since it resides client side, it can be removed by accident by the client. In most common browsers for instance the contents of local storage is removed when the client removes cookies.

CHAPTER SEVEN

CONCLUSIONS

In this thesis a module for Vaadin has been presented to change the internal session management of the framework. The module modifies Vaadin by introducing a session manager, which stores and loads the state of clients from and to different storage medias located outside the server environment. The storage medias chosen were the *Amazon Simple Storage Service* and HTML5 *Local Storage*. Both are very different in terms of functionality and cost. Both have their benefits and drawbacks.

The goal of this thesis has been to research whether the state of clients can be persisted outside the server environment to allow long time session persistence. This was succeeded by the module. The results, however show, that there is further research needed to optimize the solution. The results show that persisting the state after each state modifying request is not effective enough in terms of execution time, since there is a notable delay. Besides optimizing the connectivity and the serialization process, which are the two most time consuming tasks, a proper study should be made into when it is optimal to store the state data. Also the financial value of clients state data should be researched and defined, especially when using S3, since the cost-benefit ratio may be negative. The presented module may not be effective enough to be used in an actual production environment, but it is a good starting point for developing one.

Persisting the state of clients for a longer period of time allows web applications to be more alike traditional applications running on a native operating system. The module presented allows this, though not effectively.

Being able to persist the state of clients can give users of web applications value, since it can prevent doing request multiple times in vain. This in turn gives value to application providers and can prove beneficial. After all, the more satisfied a user is, the higher the probability is of a user returning and being a long-term customer.

BIBLIOGRAPHY

[1] *Book of Vaadin: Vaadin 7 Edition - 1st Revision*. Vaadin Ltd, 2013.

[2] Freeman Adam. *The Definitive Guide to HTML5*. Apress, 2011.

[3] Amazon. Amazon simple storage service console user guide, api version 2006-03-01, 2012.

[4] Amazon. Amazon simple storage service developer guide, api version 2006-03-01, 2012.

[5] Amazon. Amazon simple storage service (amazon s3). `http://aws.amazon.com/s3/`, 2013. Visited: 2013-08-26.

[6] Bryan Basham, Kathy Sierra, and Bert Bates. *Head First Servlets & JSP - Passing the Sun Certified Web Component Developer exam*. O'Reilly, second edition, 2008.

[7] Robin Berjon, Steve Faulker, Travis Leithead, Erika Doyle Navara, Edward O'Connor, Silvia Pfeiffer, and Ian Hickson. Html5 - a vocabulary and associated apis for html and xhtml w3c candidate recommendation 29 april 2014, 2014.

[8] Joshua Bloch. *Effective Java*. Addison-Wesley, second edition, 2008.

[9] Rusty Harold Elliotte. *Java ™Network Programming*. O'Reilly, third edition, 2004.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. 1999.

[11] Martin Grotzke. Kryo framework website. `https://github.com/EsotericSoftware/kryo`, 2013. Visited: 2014-05-12.

[12] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[13] Oracle. Interface httpsession api reference. `http://docs.oracle.com/javaee/7/api/javax/servlet/http/HttpSession.html`, 2013. Visited: 2014-05-10.

[14] Henrik Paul. Clientstorage addon website. `https://vaadin.com/directory#addon/clientstorage`, 2013. Visited: 2013-06-10.

[15] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.

[16] Christopher Schmitt and Kyle Simpsons. *HTML5 Cookbook*. O'Reilly Media, 2011.

[17] Lean Shklar and Rich Rosen. *Web Application Architecture - Principles, Protocols and Practices*. John Wiley and Sons Ltd, second edition, 2009.

[18] Vaadin. Vaadin example application website. `https://vaadin.com/tutorial`, 2013. Visited: 2013-04-28.

[19] A. Vukotic and J. Goodwill. *Apache Tomcat 7*. Books for professionals by professionals. Apress, 2011.

SPARNING AV TILLSTÅND ÖVER EN LÄNGRE TID I ETT JAVA-BASERAT UTVECKLINGSRAMVERK

Introduktion

Efterfrågan för mera komplex funktionalitet inom web applikationer har ökat avsevärt. Utvecklingen av dessa har rört sig emot funktionalitet som endast traditionella applikationer, exekverade på egen hårdvara, kunnat leverera. En stor del av detta har varit möjligheten att skapa tillståndskänsliga applikationer.

En av de vanligare sätten att möjliggöra tillståndskänslighet i web applikationer är att spara tillståndet av klienter i servern. Detta har dock en tyngande effekt på servern eftersom tillståndet av klienter sparas i minnet, som är begränsat. Därför måste tillstånden hanteras av servern på ett sätt så att minnet inte tar slut. Detta görs genom att tilldela tillstånd en livslängd som är beroende på klienternas aktivitet i web applikationen. Om klienten är inaktiv över en viss period kastas tillståndet från minnet och är förlorat. Detta är ett problem, eftersom det kan hända att tillståndet försvinner då det inte är meningen. En användare av en web applikation kan t.ex. bara ta en paus från web applikationen, vilket misstolkas och skapar därmed besvär när tillståndet inte mera finns. Vad om man kunde flytta tillståndet bort från serverns minne till ett mera kostnadseffektivt lagringsutrymme? Vad om detta kunde göras för vilken web applikation som helst utan att ställa krav på utvecklarna?

Att kunna spara tillståndet av klienter skulle vara lönsamt för både utvecklare såsom också för användare. Från en utvecklares perspektiv ger möjligheten att spara tillståndet över en längre period en mera intuitiv programmeringsmodell. Från en användares perspektiv kan detta spara tid, vilket betyder nöjda användare som kan resultera i ökad vinst för leverantören av web applikationen.

Denna avhandling undersöker om det vore möjligt att spara tillståndet av klienter utanför servermiljön över en längre tidsperiod. Detta görs genom att presentera en modul till programmeringsramverket Vaadin som ändrar tillståndshanteringen av ramverket. Mera specifikt är tanken med modulen att kunna spara kompletta tillstånd till *Amazon Simple Storage Service* (S3) och HTML5 *Local Storage*.

Bakgrund

Web applikationer skiljer sig från traditionella web sidor vad gäller funktionalitet. Underliggande struktren för bägge är lika, dvs att en server är värd för web applikationen och skickar data till flera klienter som är anslutna till applikationen. Bägge bygger på HTTP protokollet, som är den grundläggande teknologin bakom webben. Dock stöder inte protokollet sparandet av tillstånd, vilket är en av de definierande egenskaperna av en web applikation.

Tillstånd är data över alla förfrågningar respektive svarförfaranden mellan en klient och en server. Mera allmänt kan tillstånd beskrivas som konversationen mellan en klient och en server. Tidiga implementationer av sparandet av tillstånd sköttes genom att spara det hos klienten i exempelvis kakor (*cookies*). Detta är dålig design, eftersom data då är modifierbart av klienten. Dessutom måste data skickas med varje förfrågan, vilket snabbt kan försämra prestandan. Nuförtiden är det vanligt att tillståndet för en klient sparas i serverns minne, vilket är bra design, eftersom klienten inte kan komma åt själva data. Det som sparas hos klienten är en textuell identifierare, som skickas med till servern vid varje förfrågan. Dock är detta inte heller helt utan problem, eftersom detta betyder att det måste finnas tillräckligt med minne i servern för att kunna spara alla tillstånd av alla uppkopplade klienter. Minnet kan snabbt ta slut, vilket kan leda till att servern kraschar. Detta är ett problem som modulen i denna avhandling strävar till att lösa genom att spara tillståndet till *Amazon Simple Storage Service* och HTML5 *Local Storage*.

Amazon Simple Storage Service (S3) är en tjänst som tillåter att spara data i Amazons datamoln. Tjänsten är en betald tjänst vars kostnadsmodell är enligt kostnad per förfrågan, samt kostnad per lagrad datamängd. Tjänsten är klassificerad som mycket pålitlig, säker och effektiv av Amazon, mycket tack vare möjligheten av redundans av att lagra data geografiskt utspritt. S3 fungerar enligt en ämbar/objekt (bucket/object) modell, dvs man sparar objekt i något de kallar ämbaren. Amazon ger tillgång till tjänsten både genom programmering, samt också via ett web gränssnitt.

HTML5 *Local Storage* är lagringsutrymme hos klienten. *Local Storage* påminner mycket kakor, men skiljer sig genom att ha störe datalagrings kapacitet. Medan det är endast möjligt att lagra 4 KB data i kakor, tillåter *Local Storage* lagring av data upp till 5 MB av typen String objekt. Eftersom *Local Storage* är belägen hos klienten kan detta inte sägas vara ett säkert utrymme för lagring av tillstånd, men det är gratis, eftersom det är klienten som står för lagringsutrymmet.

Serialization är en fundamental process i nätverksprogrammering. Man kan inte bara spara Java objekt till något lagrinsmedia, utan data måste först genomgå serializationsprocessen. Detta betyder att dataobjektet konverteras till en form som kan lagras och sändas över en länk i ett nätverk. Serialization är mycket aktuell för denna avhandling, eftersom tillstånds data objekten är meningen att skickas över länkar i nätverk.

Inom ramverket Vaadin representeras det kompletta tillståndet av klienter genom objektet `VaadinSession`. Detta objekt består av all data Vaadin behöver för att en Vaadin applikation skall fungera. Det är dessa objekt den utvecklade modulen strävar efter att spara utanför servermiljön.

Design

Modulen utvecklad i samband med denna avhandling designades att möjliggöra sparandet av tillstånd av klienter utanför servermiljön, samt hämta dessa tillbaka och återskapa dem. Detta gjordes genom att ställa upp krav för modulen. Exempelvis bestod dessa krav av att tillståndet måste vara det kompletta tillståndet och att modulen måste fungera med vilken Vaadin applikation som helst. Kraven ledde till identifieringen av delar enligt vilket utvecklingsarbetet uppdelades. Till dessa hörde för det första att kunna få tag på det kompletta tillståndet, dvs `VaadinSession` objektet. För det andra måste det vara möjligt att genom programmering kunna komma åt det utomstående datalagringsmediet. Den tredje och sista delen bestod av att man måste kunna spara och hämta dessa objekt från datalagrinsmediet. Dessa delar ledde till att själva ramverket måste modifieras och den interna tillståndshanteringen ändras. Förutom detta måste en hanterare utvecklas som är ansvarig för att skapa anslutningen till externa datalagringsmedierna, samt sköta om sparandet och hämtandet av objekten från dessa. Hela modulen designades att vara konfigurerbar via driftsättningbeskrivaren (*deployment descriptor*), *web.xml*.

Prestanda

Evalueringen av modulen utfördes genom att mäta exekveringstiderna vid användningen av S3 och Local Storage. Detta gjordes genom att introducera stoppur i koden. En annan metrik (*metric*) som användes var storleken av tillstånds objektet efter att de lagrats till bägge datalagringsmedierna. I användningen av S3 användes web gränssnittet för att erhålla storleken medan utvecklingskonsolen i chrome webbläsaren användes för att mäta storleken vid användningen av Local Storage. För att mäta metrikerna användes Vaadins exempel applikation *Addressbook*, utvecklad av Vaadin.

Resultaten visar att de mest tidskrävande processerna är uppladdning och nerladdning av tillstånds objekt. Efter detta var serializationen den mest tidskrävande processen. Vid användningen av S3 visade det sig att tiden för att lagra och hämta tillstånden var inom acceptabla gränser att hålla en användares tanke obruten vid användning av en Vaadin applikation. Detta gällde också vid användningen av Local Storage, dock så utelämnades processen av uppladdning och nerladdning, eftersom test applikationen kördes på en lokal server. Vad gäller storlek har S3 inga restriktioner, vilket betyder att tillstånd kan teoretiskt sett växa utan att ha inverkan. Dock måste man komma ihåg att uppladdnings och nerladdnings tiderna korrelerar med objektens storlek. Vid användningen av Local Storage visade resultaten att objekten snabbt växer och kommer att fylla datalagringsmediet.

För att kunna ta i bruk den utvecklade modulen i en produktionsmiljö måste modulen förbättras. Detta gäller för exekveringstiden vid användningen av bägge datalagringsmedier. Vid användningen av Local Storage är kapacitet ett problem som måste lösas. Vad gäller förbättring av exekveringstiden är ett förslag att man inte sparar tillståndet av klienter vid varje förfrågan som ändrar tillståndet. Vid användingen av S3 kunde detta göras precis innan tillståndet tas bort från serverns minne. Vid användningen av Local Storage är detta inte möjligt, eftersom då detta händer är anslutningen till klienten redan bruten.

Slutsats

I denna avhandling presenterades det en modul för att ändra Vaadin ramverkets interna tillståndshantering. Modulen modifierade ramverket att spara en klients tillstånd vid varje förfrågan som ändrar tillståndet för att sedan kunna återuppliva tillståndsdatan i servern vid senare tidpunkt. Tillståndet sparades till externa datalagringsmedierna *Amazon Simple Storage Service* och HTML5

Local Storage.

Målet med denna avhandling var att undersöka om det är möjligt att spara tillståndet av klienter utanför servermiljön för en längre tid för att sedan kunna uppta dessa igen. Detta visade sig möjligt med den utvecklade modulen. Resultaten visade dock att modulen inte är så effektiv och borde förbättras förrän den kan tas ibruk i en produktionsmiljö. Modulen är inte meningen att tas i bruk som den är, men i samband med denna avhandling kan den hjälpa utvecklingen av en modul duglig för produktionsmiljö.

Att kunna spara tillstånd av klienter över en längre tid gör web applikationer mera likt traditionella applikationer. Att kunna spara tillståndet ger värde för användarna av en web applikation, eftersom detta kan förhindra att samma förfrågningar måste göras flera gånger. Detta kan också visa sig ge mervärde åt leverantörerna, eftersom en glad användare är en återvändande användare.

APPENDIX A

EXAMPLE DEPLOYMENT DESCRIPTOR WEB.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4" xmlns="http://java.sun.
   com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance" xsi:schemaLocation="http://java.sun.com
   /xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.
   xsd">
  <display-name>SessionPersistenceProject</display-name>
  <context-param>
    <description>
    Vaadin production mode</description>
    <param-name>productionMode</param-name>
    <param-value>false</param-value>
  </context-param>
  <servlet>
    <servlet-name>Sessionpersistenceproject Application</
       servlet-name>
    <servlet-class>com.vaadin.server.PersistenceStorageServlet
       </servlet-class>
    <init-param>
      <description> Amazon S3 Access Key </description>
      <param-name>accessKey</param-name>
      <param-value>AKIAIAISAG7CFPUTZ6VQ</param-value>
    </init-param>

    <init-param>
      <description>Amazon S3 Secret Key </description>
      <param-name>secretKey</param-name>
      <param-value>pB9G2MAY2+VsG54+0fQuLIlvpCphuY2YRxY+jFA0 </
         param-value>
    </init-param>

    <init-param>
      <description>Amazon s3 Bucket Name</description>
      <param-name>bucketName</param-name>
      <param-value>vaadinservletbucket</param-value>
    </init-param>

    <init-param>
      <description>What serialization mechanism to use</
         description>
      <param-name>serializationmechanism</param-name>
      <param-value>java</param-value>
    </init-param>

    <init-param>
      <description>Storage media to use</description>
      <param-name>SessionStorage</param-name>
      <param-value>com.vaadin.server.sessionstorage.
         AmazonS3SessionStorage</param-value>
    </init-param>

    <init-param>
      <description>
      Vaadin UI class to use</description>
      <param-name>UI</param-name>
      <param-value>com.example.sessionpersistenceproject.
         SessionpersistenceprojectUI</param-value>
    </init-param>

    <init-param>
      <description>Application widgetset</description>
      <param-name>widgetset</param-name>
      <param-value>com.example.sessionpersistenceproject.
         ClientStorageWidgetset</param-value>
    </init-param>
  </servlet>

  <servlet-mapping>
    <servlet-name>Sessionpersistenceproject Application</
       servlet-name>
      <url-pattern>/*</url-pattern>
    </servlet-mapping>

    <welcome-file-list>
      <welcome-file>index.html</welcome-file>
      <welcome-file>index.htm</welcome-file>
      <welcome-file>index.jsp</welcome-file>
      <welcome-file>default.html</welcome-file>
      <welcome-file>default.htm</welcome-file>
      <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>

        <async-supported>true</async-supported>
  </web-app>
```