IT 08 043
Examensarbete 30 hp
November 2008

Mobile Widget Architecture

Lars Vising

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Mobile Widget Architecture

Lars Vising

Driven by the vision that mobile computing devices will constitute the vast majority of the near future computing devices, new technology and frameworks arrive to facilitate the creation of highly interactive mobile web applications. Those applications are called rich internet applications. This thesis concentrates on a subset of those. Namely the internet enabled widget. Widgets are small computer applications that perform a single task and lowers the threshold of knowledge required for creating them. As the evolution of technology and internet use proceed, this leads to a number of rivaling technologies and more or less obsolete devices. This thesis investigates the realization of widgets on devices already considered to be of a past generation. Such a widget system consisting of a mobile widget engine and a server side engine can be more or less isolated from contemporary widget enabling frameworks. This thesis highlights the suitability of using the Representational State Transfer (REST) architectural style for constructing both widgets as web resources and addresses the contrast between the original large system, of internet scale, use of REST with its adoption of comparatively small scale widgets.

Handledare: Leonid Mokrushin
Ämnesgranskare: Justin Pearson
Examinator: Anders Jansson
IT 08 043
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

Desktop computers as well as their smaller relatives, mobile devices are steadily evolving into more and more capable platforms. As the capacity to deliver graphics and video over broadband connections increase the role of internet services and internet users are changing. As the rich internet application tries to narrow the gap between a traditional computer application and the web browser this is changing the expectations and the view of the internet. The change in view and use of Internet is part of a set of core ideas commonly referred to as Web 2.0, indicating the transition from a old way to use the internet to a new set of business models and placing more importance to internet content generated by individuals. Widgets are one of the parts of this *new Internet* [1] and this thesis is about the role and place for widgets on mobile devices.

1.1 Problem Description

The observed trend in widget systems and contemporary usage of widgets is twofold. First there is drive towards finding new uses of widgets and use them for introducing new ways of accessing the Internet. Second the creation of rich Internet applications are departing more and more from the development of traditional computer software. As new tools and supporting frameworks become available for both running and creating widgets this means that widgets may spread rapidly to mobile devices. However, this evolution places demands on already limited mobile devices as they may not have the support for each and every introduced item of technology. The first problem addressed by this thesis is thus to lay out the foundation for the implementation of a widget engine for mobile devices which are as of the time of writing not considered being state of the art. While new technologies and practices for widgets and rich Internet applications emerges for mobile devices they may not support access to built in functions on the device, for example the embedded camera. Thus, it is considered important to retain the capability to use the hardware of the mobile device.

The main research goals of this thesis are the following:

- Briefly investigate some common platforms for widget development and various ways to realize them on current and near future mobile devices. Even if those platforms are not fully compatible with the target mobile devices it is considered important not to isolate the proposed widget engines capabilities from other similar observed work. This as to provide support for any observed emerging user scenarios for widgets of today and the near future.

- Find a suitable structure for realizing widgets on mobile devices which are as of the time of writing not considerer to be state of the art. Specifically this means the Sony Ericsson JP-7 series phones. In addition to the inherent limited capabilities of the mobile device there exist other obstacles in realizing widgets on those devices according to emerging main stream techniques. A number of widget systems exists, both for mobile and desktop computers, that are based on using a web browser. Choosing a browser based and/or Java Script enabled solution may not be problem

free for every mobile device. The browser and Java Script capabilities of the JP-7 series phones are limited and motivate the search for alternate ways of widget realization. Specifically this means laying out the foundation for a widget run time engine that hosts widgets that is able to function as rich internet applications without the support of a mobile browser.

1.2 Purpose

The purpose of this thesis is to provide guidance and answers on issues that regard the creation of a widget engine that allows widgets to function as rich Internet applications on a mobile device and simultaneously be able to access both the phonetop and device hardware, such as the embedded camera. Access to the phonetop means applications will be able to run at the startup of the device. The phonetop can be said to function and look like a desktop computers traditional desktop interface. This to facilitate the provision of a prototype widget engine implementation which may serve as a proof of concept widget engine for the selected legacy devices.

1.3 Delimitation

A capable user interface in the areas of good layout and interaction capabilities is considered vital for the kind of rich internet applications this work is focused on. Investigation and the study of different layout and interaction techniques are however considered to be outside the scope of this thesis.

2 Background

2.1 Introducing the Widget

This report does not strive to achieve a very precise definition of what a widget is. The English language commonly uses the term widget to denote something you do not have a name for or do not really care to make up a name for. In the context of information technology a definition of widgets, is best given by illustrating some typical uses of a widget. The magazine Newsweek in one article asked the question *Will 2007 be the year of the widget?* [2]. This article can be seen as the result of an increased spread of widgets and that widgets are appearing in more and more places. Widgets are found on the computer desktop, on web pages and on mobile phones. Widgets are an integral part of popular operating system like Microsoft Vista and OS X for Macintosh computers. Widgets appear as part of web pages and widgets can also be constructed to display a certain or combination of web pages.

2.2 Widget Examples

In the following two illustrations two sample uses of widgets are shown. First shown is a wide variety of widgets as they can look when appearing on the Windows Vista operating system, figure 2-1. Here among others a weather widget, a clock and calendar widget are shown.

Figure 2-1 Vista Widgets

Figure 2-2 A widget station displaying a weather widget

The second example is a from a dedicated widget device, figure 2-2, which has the purpose of running widgets that can be downloaded from the Internet.

Widgets are also found on mobile phones, and as previously stated the widgets on mobile phones are the focus of this thesis. Mobile devices have different capabilities for utilizing contemporary technologies for enabling widgets. More advanced phones will be running some variant of an more advanced operating system, offering much the same functionality as on desktop computers and widgets on those will work like widgets on desktop computers. For the rest of the mobile phones the situation is different when it comes to running widgets and what role a widget has on the mobile device. There exist broadly two different approaches for enabling widgets on a mobile device. Devices which have a web browser capable of JavaScript can use this combination for running widgets. The other approach consists of creating a widget engine that consists of an application that is installed on the device.

There are more users with a mobile device than there are people with a desktop computer. But the current situation is that those phones can not be said to constitute a common technical platform except for services such as normal voice calls and SMS. Widgets installed on mobile devices not only offers a way to create new services but may be viewed as the means to create common, and unifying, elements for different mobile devices [3].

2.3 Defining the Widget

For the purpose of this work the conclusion is made that a widget has three distinct properties. First it provides users with increased functionality for their system. The

widget accomplishes this by adding new functionalities and may provide for alternate ways of using the system. The widget may work as a replacement or enhancement of the phones built in interface with underlying operating system. Secondly it gives widget developers the necessary tools to create them. This is accomplished by working within a single architecture framework for the chosen widget system. And last widgets may also offers new revenue models for companies and new channels for marketing and advertisement. While this may be said to be true for each and every computer application the widgets introduce some very specific properties when installed on a mobile device. Because of the limited screen size and the very number of applications a mobile device can host the widgets are the tool and medium to shape the role of the device. This as the purpose of the device and the capabilities are mediated trough widgets.

Widgets are reusable, self contained visual idioms that provide interaction capabilities [4]. This statement together with the limited set of examples leads to a first somewhat loose definition of a widget. A widget is a computer program that is comparatively easy to distribute and use. To this it is possible to add a second criteria for a widgets, namely how it is created. The widgets are typically very small computer applications, both in terms of memory footprint at runtime and the amount of resources, such as CPU time they require. The widget creation process contrast sharply with traditional computer application development performed with general purpose programming languages. Many such languages have to be used in conjunction with compilers and special purpose libraries and are aimed at creating system of greater complexity than widgets. Widgets are said to be relatively lightweight also in the demands of computer programming skills required. The widgets are typically not run by themselves on the system but instead are under control of some execution environment like a web browser or a dedicated widget engine.

2.3.1 Widgets and the Internet

Widgets can be viewed as part of an undergoing process to make the internet more accessible. This can be achieved by widgets that let people arrange their favorite web sites and for example collect several information feeds from news broadcast into a single easily accessible place, a widget. Widgets are parts of a process to structure the practicalities of how the internet is accessed and used.

The relationship between internet and widgets are enhanced by the work in process to achieve a standardized definition of widget functionality [5]. The work is undertaken by the World Wide Web Consortium and the present draft version of this document indicates that it is essential that the widgets are capable of internet access and that there should be a unified way of packaging and distributing them. Thus according to the proposed standard a widget can be said to be defined as a sharable application, that should be able to run with no or non significant modification on different systems and capable of internet access.

2.3.2 Representational State Transfer

REST is an acronym that stands for representational state transfer. It is an architectural style that emphasizes abstraction of both data and services. It does so by as relying on exploiting existing web technology and protocols. REST is proposed as a suitable replacement for creating service oriented architectures on the web. This space is shared with other techniques, for example SOAP and XML RPC.

The origins of REST can be traced back to the dissertation by Roy Fielding from the year 2000[14]. The author concludes that the success of the Web relies on a few fundamental properties. Namely that the web can be viewed as a very large collection of resources and that every one of those resources can be uniquely distinguished from another resource. Furthermore the Web relies on a unified way of retrieving and modifying those resources. This is accomplished with URLs, uniform resource locator's, and the HTTP protocol. A URL is capable of giving each resource a unique location. The HTTP protocol is simple in the way that it only allows limited set of action to use on the resources. Those actions are enough to support the basic operations of reading, creating, updating and deletion of a resource. The main concern for the HTTP protocol is not to focus on and supply a wide variety of commands for consuming web services. Instead it focuses on the fact that there in theory can be a unlimited number of resources of any type. The strength of the REST approach and the success of the HTTP protocol for the Web is that this protocol gives all resources a single unified interface. This interface is rather constrained and limited compared to other interfaces for accessing web services.

All actions on resources are atomic and stateless. This is because the REST style is describing how a well designed web application should function and behave. The user navigates through the vast space of Internet resources (web pages) by selecting hyper links (state transitions) which results in the application changing its state (displaying a new page). The resources themselves are viewed as an abstract concept. A resource is viewed as a holder of different representations of the resources contents. The type of representation can be specified in the request message by for example asking for an image or text representation. As shown in the related works section, these resources can be modeled to provide data for viewing by widgets.

The original dissertation proposing REST presents the theory as not only an architecture but also as a way to evaluate different architectures level of adherence to the REST principles [15]. The originally proposed REST is a style derived from analyzing several architectures for network based systems style [14]. According to the author the REST architectural style is made up of a client server relationship. The interaction between those two must be stateless. This means a request from a client must be sufficient for the server to respond to the request. The client server relationship further means that a clear separation of basic responsibilities can be achieved. The messages sent should always be self descriptive. The HTTP messages are packaged in such a way that the client and server always know what to do with them. Transmitted data must be labeled as cacheable or non cacheable. All interactions must be performed using a single unified interface. The constraint that it should be possible to download code for local execution is considered not essential but optional.

What is not optional is that a REST system must be able to function as a composition of several layers. Each layer should only have knowledge of the layer it communicates with. This to reduce complexity of large systems and make it easier to add layers that can perform load balancing over multiple networks and servers. Since the REST style is describing how a well behaving internet application functions its target is to provide a solution for internet scale size systems and their needs. The main challenge is to interconnect multiple information networks spanning national and organisatory boundaries. The author argues that those constraints given that form this architectural style are enough to give an efficient solution to such problems. The need to have a large amount of different types of resources is handled by the fact that the REST style does not limit the actual implementation to certain fixed predefined models. Each application is free to choose its own way for implementing resources and their representations.

3 Related Work

In this section three examples of related work are given. They were chosen because they all have provided details to this thesis view of widgets and they have provided guidance for the design of the prototype widget engine architecture. The interested reader can look into Appendix A which contains links to more information about these and other mobile widget solutions.

3.1 Available Mobile Widget Systems

3.1.1 Widsets

This widget engine is based on a JavaME platform. A software development kit as well as the compiler and emulator are provided. The widgets are developed and compiled and then uploaded to a server for distribution to actual phones. An on line visual editor for developing widgets is provided. However, these widgets are essentially limited to picking up RSS feeds and for displaying text and images. To create more functionality the developer needs to use a programming language. A scripting language called WidSets Scripting Language, formerly known as Helium, is used. The language itself bears some resemblance to Java and most of all JavaScript. One technical motivation for a custom made language is the lack of class loader functionality for Java ME. On MIDP 2.0 devices there is no support for adding a new class, that is a new widget, to the existing set of widgets already installed on the Widset widget engine. The language comes with a limited set of native data types.

Widset API and Widget Structure

An API is provided for such functionality as adding menus to screens, responding to the phones soft keys and using the camera. Access to the built in record memory store (RMS) are wrapped in function calls such that each widget has access to its own dedicated part of the RMS. Standard Java ME threads are not exposed or made available to the widget

developer, instead support is given in the language for timers and timed callback functions. The API includes several functions for creating and manipulating graphics. The syntax of those function calls most often resembles their Java ME counterparts. The strategy appears to be that a range of standard Java API calls have been wrapped into custom made API calls; either to simplify use or to group several Java ME API calls into a single Widsets API function.

A configuration file is used and includes meta data of the widget. In the configuration file location of resources such as images are declared. A widget can be uploaded in a package consisting of widget code, configuration file and necessary resource files. Created widgets can be shared by means of uploading them to a public widget library.

3.1.2 Yahoo Mobile Widgets

Like the previous example this second example of a widget engine also relies on its own scripting language for developing widgets. This time the language is instead inspired by XML. The language is called Blueprint and can be viewed as an extended form of XML with added functionality for creating widgets. At the time of writing version 1.0 of this language has been released. Promised for future releases are model view and controller (MVC) programming based on xForms [6]. Blueprint is a declarative language and as such it contains no scripting or procedural code. The syntax is directly related to xForms. A software development kit is provided and it is free to use. The ordinary tag elements of HTML are retained and it is possible to use them in the same document as the Blueprint instructions. Compared to Widsets scripting language or Javascript the Blueprint language is not particular forceful in the area of offering logic control and data structures for the widgets.

Design Strategy for Yahoo Mobile Widget System

According to Yahoo a widget is an application with customized layouts. The widgets themselves rely heavily on server side code for processing and managing the control logic of a widget. The main focus of the design appears to be offering ways to create various layouts for pages and link page content to internet data sources. The main element available for the programmer is called page and is a XML data file, which specifies the layout and user interface. The blueprint XML syntax has tags for such functionality as adding buttons and menu items to the layout. A widget is foremost viewed as a hierarchy of graphical elements. Those elements and their relationship are visible and readable in the XML form. The engine in its present form relies on Java applet Yahoo Go and as no support for directly accessing the RMS or use the mobile for storing data. The design strategy behind this solution appears to have been focused primary on reaching as many mobile devices as possible [7].

The widget is packaged into a single compressed file and uploaded to a server where it is made available to all users of the widget engine. The widget must be checked and approved before it is made available on internet to the public. A web page exists that allows developers to test their widgets before upload. A mechanism to download a widget

from any web page is also provided. This consists of a link to ta download server were user made widgets are published.
This is reported as work in progress and the goal is to offer the ability to running the same widgets on both mobile and desktop components. Currently the widgets run on phones with an HTML browser and for some phone models, xHTML browser.

3.2 Creation of Web Mashup Applications

3.2.1 Telefonica RESTful Gadgets

The term mashup refers to the creation of web pages by means of rearranging and combining contents from different sources into a single web page. Creation of mashups with the developed prototype is discussed in chapter six. In [8] it is argued that a web service built upon the Representational State Transfer Architecture (REST) would support development of mashups in an effectively way. In this paper a classification of resources are suggested. The resources are grouped to provide functionality for enabling the creation of mashup applications of existing components. A system with the following resources is presented:

- *Data sources* feeding the mashup application with data
- *Operators* transforming the data sources
- *Gadgets*, which are responsible for providing graphics and simple and efficient user interaction mechanisms.

The visible component is the graphical human computer interaction resource called a gadget in the proposed terminology. It is pointed out that gadgets behave consistently and gives the appearance of a strongly cohesive interface. Thus the user does not directly see that the interface is made up of several gadgets. A mashup is built with one or more such gadgets. Each of those gadgets is linked to one or more data sources, that supplies data, and operators that can modify data. Those components can be linked into chains of arbitrarily length, where each component provides the input for the next one. This work has resulted in a developed prototype with the Spanish operator Telefonica. Note that this work does not directly addresses mobile widgets as such but instead focuses on both the high and low level aspects of internet content delivery.

4 Rich Internet Applications

4.1 Defining the Rich Internet Application

A rich internet application (RIA) can be described as a browser you can use even if it is not connected to the internet and the look and feel of it mimics a traditional desktop application. The goal of the rich internet application is to provide users with highly interactive way to access web pages and give the application the means to be able to communicate in near real time with server side data. The rich Internet application is a part

of the effort of overcoming the basic limitations of existing internet protocols and the traditional workload distribution of the client server model.

The word “rich” in the name refers to the fact that these applications are capable of delivering relatively large amounts of multimedia contents from the internet without sacrificing the ability to interact with the application. The early internet experience relied on a server delivering pages whenever a user requested them, which, in most cases, will lead to long loading times. Especially when the user has to do a lot of navigation and loading of media contents the reliance on the client server approach is not comparable to the experience of using a traditional desktop computer application.

The driving force behind the appearance of the rich internet application is to close the gap between the browser experience and the desktop application experience. The strength of this concept is that it allows information to be updated at a rate high enough to allow the user to work more efficiently with the user interface presented [9]. Some common element of rich internet applications can be identified, namely whatever underlying technology used they all have the ability to achieve a combination of graphics rendering and communication power to be able to retrieve data and update the user interface. The common strategy is to break up the traditional client server relationship and move more functionality to the client.

The rich internet application may or may not require the presence of a server connection. They can be self contained and installed on the client and may as well permit offline capabilities. Offline capabilities means that the application is useful even if the device is not currently connected. The support for near real time collaboration between different users so they can work on the same task can be considered a good indication of the capabilities of a rich internet application compared to a traditional web application.

4.2 Developing Rich Internet Applications

Rich internet applications (RIA) relies heavily on their graphics output and their ability to retrieve data from external sources. So modeling such a system is to work simultaneously in the field of multimedia engineering and distributed computing.

The development of rich internet applications are constrained by the fact that the underlying technology of the World Wide Web was never designed for highly interactive multimedia sessions. Development must rely on tools and methods for overcoming those constraints. In practice this will mean waiting for new additions to the Java scripting language, new plug in modules and the solving of various incompatibility problems that may arise when trying to overcome the constraints.

Since the rich internet application strives to merge the interaction possibilities of a traditional desktop application with the normal web page navigation. There seldom is clear way exactly how to do this. The current situation of user interfaces are described as a landscape where new interfaces and interaction methods constantly are introduced.

This situation are the result of applications constantly are in a beta release phase and new interaction methods are tested [10].

The introduction of the rich internet application creates a number of questions that need to be addressed by the chosen system architecture. It is possible to have the business logic run on the client, the server or as a combination of both. This is also complemented with the new problems of data distribution that is created. The underlying system design question of where the data should be located now has more answers. Since the client is more capable it can store and manipulate more data. This data can be persistent as well as non persistent. The problem of maintain consistency between duplicate data sets on the client and server are addressed as well as maintaining the security of the data in a relatively weakly protected sandbox runtime [11].

The lack of established procedures and methods to follow for achieving a coherent architectural style for rich internet applications are addressed in a study by [12], which highlights the difficulties in evaluating different rich internet application development frameworks. A suitable architectural style for modeling a rich internet application is proposed in [13]. In this paper the benefits of using the representational state transfer (REST) architecture are highlighted.

4.3 Mobile devices as Rich Internet Application platforms

4.3.1 Device Runtime Environment

Ever since the introduction of the first graphical web browser the quality and interaction capabilities have increased as a result of new tools and techniques have been made available. This is also true for mobile devices. In figure 4-1 the evolution of Java ME graphics are shown. The first box corresponds to the graphical capabilities of the intended platform for the widget engine. Thus this may be seen as the base line capabilities available on nearly all mobile phones. Note that the later additions to the graphical capabilities directly addresses the point of improving and providing support for developing rich internet applications.

Mobile devices are capable of accessing internet and doing so by the use of widgets. But mobile widgets for internet access can not simply be a direct translation of their desktop counterparts, mainly because the mobile device has some limitations. Not only is the screen size of much smaller size, the processing power is less and the capabilities of available device browsers varies. As more capable mobile devices are introduced there is also an increase and refinement of technologies available for creating mobile widgets. As this paper is intended to propose a way to realize widgets on phones not suitable for these emerging technologies it is useful to remember that emerging technologies sometimes also are rival technologies and there is no certainty what standards will prevail. A brief summary of some of those emerging technologies is given in appendix B.

While the intended target phones for this work can be said to be not of the highest standard and graphic capabilities, as of the time of writing, it is worthwhile to remember that the past generation of phones will have a chance to constitute the bulk of the total phones in use. There is a balance between use of technologies and the total number of different phones a widget engine can be deployed on.

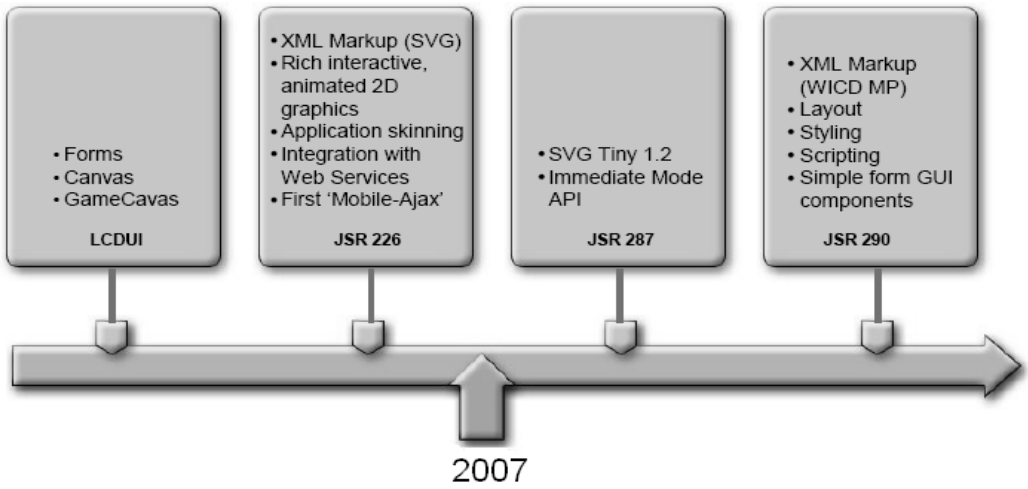

Figure 4-1 Evolution of Java ME graphics. Based on image from [17]

4.3.2 Mobile Widgets as Rich Internet Applications

This section is intended to shed light on the role of a mobile widget when the widget engine is not dependent on a mobile browser or third party technologies for internet content delivery and presentation. The intention is to lay out a base for showing that widgets on a mobile device without browser or JavaScript are a viable approach. This as the intended target phone is assumed to have no built in support for JavaScript. It should be noted that this scene could possible be viewed as placing the work as somewhat off center when compared to current mobile and desktop widget trends.

The first observation made is that there is a difference between widgets as a rich internet application and the use of a browser for consuming internet information. The main difference is that if the widget is designed to provide a single function, for example displaying weather forecast information, the widget may do so by processing and combining internet resources rather than displaying the information directly. A rich internet application running under the control of a browser may also do this but it is

limited to what the browser environment allows. A widget running inside a widget engine may on the other side have access to the underlying system resources, like the file system and ability to access other applications on the local system. The effect of this is that the widget becomes a container for processed data and that this data may be stored on the local system running the engine. The widgets can access locally stored data, process it further and offer it as a new internet service.

The second observation is that since the widgets are comparatively small and lightweight they are not only suitable for fast development of rich internet applications but are also suitable candidates for being the essential building blocks of rich internet applications. This means that a single rich internet application can be made up of several widgets working together.

There exist many definitions of what a widget is and what they can be used for. These first two observations add three additional remarks to the definition of a widget. This provides the starting point for define the main goals of the widget in the proposed widget engine prototype.

- The widgets may be the primary way for the user to interact with the mobile device. This may be accomplished by for example widgets for making phone calls and taking photographs with the embedded camera.

- The widgets are applications that consume internet content away from its original source and may in turn offer this content in a processed way.

- If the processed data the widget uses are located externally then there exists a very strong tie between the widget and its remote data sources. This means that the widget is a way to gain control of what parts of the vast free internet data is consumed.

It can be argued that the role of this widget offer is nothing new compared to a widget made of JavaScript and displayed by a browser. The basic shortcomings of a widget running in a browser can be said that browser based applications are inherently limited to be used online and basically are designed for communication with server side applications [18]. Although JavaScript is used for widget creation on web pages the role of JavaScript is not entirely problem free on a mobile device. Apart from lack of, or the degree of support for, JavaScript the parsing of the script tends to increase processor load and thus battery consumption. However this has to be viewed in light of the flexibility the JavaScript has to offer in terms of widget creation. Adam Sha, the architect behind Google Gears, have stated that the mojority of current mobile devices lack support for JavaScript and other enabling technologies [19]. The real mobile widget explosion will happen in the next few years, following 2007, when support of open standard have increased. The conclusion is that widget without JavaScript is not only possible and may have additional advantages but current trends are to put a sign of equality between widgets and JavaScript.

5 Prototype Architecture

5.1 The Requirements

Indisputable the sheer number of information published on the internet has grown rapidly and can be expected to do so in the future. However, a change in the roles of producer of content, the editor and the consumer, the reader can be observed. What has traditionally been a one way communication process from editor to reader is now gradually loosing ground. Instead the consumer may now be more involved in themselves also producing content. The editorial documents are mixed with a stream of user generated content and recombined into new information.

The territory for the widget engine system is set of soft concepts commonly known as Web 2.0. This is a set of principles and guidelines. As said in the introduction widgets can be viewed as a direct descendent from this viewpoint. The core functionalities of a widget directly address two key principles of Web 2.0. Two of the key concepts in this new way of looking at internet is the idea of the web as a repository for collective intelligence and that internet content more and more tends to be consumed from other locations than the original source.

Widgets directly relate to those concepts by offering a platform for small computer applications that opens up windows to internet content. Thus widgets are candidates for both storing rearranged internet data and displaying the data.Through their interactive nature the also provide a vital communication channel for user generated content. Widgets may also be said to challenge the definition of internet access software. Creation of currently available widgets is typically simpler compared to traditional software development. This comes from the definition of a widget which states the widget should be a lightweight application with a small memory footprint. Widgets not only serve as the medium for user generated content but they also serve as a catalyst for this content by lowering the knowledge required for creating the widgets.

A set of requirements for the widget engine were given before the design work started:

- The widget runtime engine should use the phone top to render the dashboard.
- The widget runtime engine will not support hot pluggable widgets but rather be a monolith midlet compiled together with the installed widgets.
- Widgets should be ordinary java classes
- The widgets should be able to communicate with each other
- Provision of a mechanism for pushing data to the widgets
- Widgets installed on the server should be source code compatible with widgets running on the mobile device
- Support for media acquiring from the mobile device

5.2 Architecture Goals

In traditional building architecture the goal of the architecture is to enrich and support the

lives of the people who will use and be affected by the architecture. Architecture can not be understood without relating it to its context. This can be phrased as the work of an architect should be engaged in a dialog with the needs, beliefs and history of a certain time and place [20]. For the widget engine prototype time and place can be replaced with the concepts of Web 2.0. So the starting point for creating the foundation for the architecture is to relate the requirements to their context.

5.2.1 Elaboration of the Requirements

Here two requirements are highlighted and discussed since they were found either missing or contrasting from observed contemporary mobile widget systems.

The Phonetop

One major difference between a widget engine running on a desktop computer and on a mobile phone is the way you interact with the widgets. On the desktop the widgets are always present and it is even possible to run several different widget engines at the same time. On the other hand the mobile device not only has a smaller screen size but more important the MIDP 2.0 specification places limits on how many applications are running at the same time and how they are started. While the widgets on the desktop will be available upon system or browser startup a mobile application has to be manually started by the user.

In recent years a number of manufacturers have begun to include the possibility of automatic launching of midlets when the phone startup. Such midlets run on the phone idle screen, normally displaying carrier information and background wallpaper decorations and animations. Note that the support for running your own applications on the phone idle screen varies between phone models. The proposed next specification for mobile devices is currently under work and draft versions of it have been published. In the coming MIDP 3.0 there will be intrinsic support for background midlets [21]. The final draft version is destined to be finished at the end of year 2008. The current state of no real common way of accessing the idle screen will probably prevail until new devices have penetrated the market.

There is a number of reasons why it is desirable to run the widget engine in the idle screen mode. The idle screen is usually the first screen the user will see when the phone starts up. This means the widgets will instantly be available. One of the first attempts of using the idle screen is Motorola's Screen3 technology which adds among other things news feeds to the idle screen. The use of the idle screen will also enable creation of more operating system interfaces, like a traditional desktop, for accessing applications and the phones resources. The term idle screen is a bit misleading, since this kind of applications will make it all but idle by adding interactivity to it.

The drive to use the idle screen is not only about creating a new way of interacting and using mobile applications. It is also about finding new ways to increase data traffic to foster new business models centered on the use of the idle screen [22]. One way to do this is to create so called on device portals. The startup screen is used for accessing various

services and for delivery of for example commercial advertisement. What is characteristic about a on device portal is that is a native part of the device. Although exactly the same functionality could be implemented at different levels of the software stack for instance through midlets or browser application. But applications running on the idle screen, with present technology, are fundamentally separated from mobile applications not utilizing the idle screen [23].

Widgets as Java Classes

The requirement for widgets as Java classes should not be seen as a statement in an ongoing discussion of the suitability and the future of the Java ME platform. Instead for the purpose of this work it serves as a delimiter adding a constraint that affects the installation and creation of widgets.

As illustrated by the Widsets, overviewed in the related works section, a custom scripting language was developed for creating widgets. Even though the widget engine is a MIDP application written in JavaME. According to the creators of Widsets the custom language was partially motivated by the need to add new widgets to the engine at runtime. Since there is a requirement for not having this functionality in the prototype engine this creates the choice of either creating the widgets in Java ME or use a dedicated language and then compile them into Java ME.

Since AJAX is commonly used on desktop browsers for enabling rich internet applications the use of plain Java also opened up the possibility to base the widget engine on AJAX technologies without the use of a browser. This approach was not used for the prototype, but if it had been pursued it is noted that there exists several sources of downloadable code for AJAX in Java ME solutions, for example several code repositories like [24].

5.2.2 Requirements Analysis

There are two requirements that relate directly to the concepts of user generated contents. Namely the requirements for media acquiring, support for content type tagging.
The mobile device is not seen as a consumer of information but an active producer. This can be automated by having the phone generate tagged data such as time and location. The widgets themselves could be tagged by users to facilitate search and discovery of widgets and their functionality. The inclusion of the push mechanism is part of the increasing trend to push content out instead of having clients fetching it.

The requirement for widget communication requires some attention. There is no observed focus on widget intercommunication on the current market. However this requirement relates closely to the concept of mashups, introduced in the related works section. This because the communication can be viewed as the foundation for linking information sources together and support for mashup enabling and creation.

The requirement for server and client compatible widget source code can be viewed as an experiment in exploring the belief that applications which are limited to a single device

are less valuable than those that are connected. Therefore the system should be designed from the start to integrate services across mobile devices, desktop computers, and internet servers [25]. The widgets would in this case be able to move freely across phones, ordinary personal computers and servers.

The notion of the widget comes with the idea that the widget should be shareable. The widget should be relatively easily to install. The requirement for installation of the widgets as a single file is not common among contemporary widget engines. Typically the widget engine is downloaded once, and thereafter the user can select and download single widgets for it. The second requirement that can be said not to be in line with mainstream widgets is that they should be ordinary java classes. This does not mean the widgets is programmed in java but if they are not then they would have to be converted into java or in this specific case java code that can run on both Java EE and Java ME. Current mobile widgets both on the mobile and on the desktop are typically developed using some language of lesser complexity than Java.

5.3 The Architecture

5.3.1 Finding Suitable System Components

This section covers some of the more influential sources that has been evaluated for finding a suitable structure for the system and define the individual widgets roles within the framework given by the implemented prototype.
First an overview of the building blocks of a REST based system is given and then two available development frameworks are presented. The latter two has not only provided guidance for the overall system layout but also influenced the concrete implementation. Especially the second example, the Restlet framework, have been influential in modeling the widget as an application managing a collection of resources and introduced the router as a special component for directing requests to those resources.

The building blocks of REST
A survey of a subset of REST components as originally presented in [14] were made. The selection to include this set is based on the fact that these particular components all have played a contributing role both to the final design and served as the base structure for the complete system. Since the widget engine should contain a push mechanism it is worthwhile to pay some attention to the concept of push and its place within the REST style. Typically a REST system does not use push methods for distributing contents. It is pointed out that this style relates closely to at least one other style, namely the C2 style [16]. This architecture has a similar structure of resources, connectors and representations. But the key difference is that C2 is push based. The reason for not having a push based model in REST is that it would be unsuitable for the large scale of the web. Components, for example a client like a browser, typically instead pull representations from resources.

Unifed Resource Locators
The unified resource locator (URL) serves the purpose of giving each resource a unique identifier. URLs can also have a semantic meaning by themselves. Information can be structured into logical hierarchies. Those hierarchies can be searched, organized and compared both not only by software agents but also by humans. The latter is true if there exists an organized way of naming resources and structuring them. Such a standardized convention offers advantages when at least partial knowledge of the structure or similar structure is known. The information structure can be discovered and explored, possible by manually reverse engineering it [26].

Below are some examples of resources and a simple schema for their syntax is presented. Note that in the context of this thesis, resources are viewed as a widget:

Favorites/Cities/NiceCities/{name}
Favorites/Cities/HotCities/{name}
Favourites/Food/{dish}
City/{longitude};{latitude}
Blend/{color1},{color2}

In this example a schema suggested in [15] is used. The (;) and (,) characters are used to indicate that resources are in the same sub resource hierarchy level. Their purpose is to give a clearer view of the resources structure. The (;) means that the order matters and (,) that the order does not matter. If blend is a resource method that when posted to its return value does not rely on the order of *color1* and *color2*. The logical grouping and structuring of resources can be done in standardized ways to facilitate inter application communication. One such method is to use the Resource Description Format RDF. The binding between widgets and URLs means RDF and similar structuring strategies enables high level descriptions of widget capabilities.

A single resource may offer, and mostly does so, several representations of its data. This can for example be textual description, an image or a movie clip. Since REST targets distributed hypermedia systems [8], the representations also most often contain identifiers for other resources. This way an application can use the identifiers to navigate among related resources. The navigation between resources is done by the single and rather small interface of REST. This means that REST offers a ready made interface and that it is both comparatively small and standardized according to the HTTP protocol. The implementation of a resource would be a component that has a unique URL that can identify the representations and the uniform interface to it would be the basic HTTP verbs.

Actions
The full set of HTTP actions is used. This architectural approach is the result of an evaluation of several architectures suitable for network systems. The conclusion made is that the main feature of REST that is lacking in other network architectures is the focus on a single, not changing, uniform interface for all components. There are a number of advantages that comes with this fact. First there is no reason to go into details and

discussions of designing interfaces when creating a new system. The interface is given and ready to use. Secondly, the architecture may be simplified and the visibility of interactions is improved. On the negative side the adoption of the HTTP verbs as the fundamental interface gives a communication language that is optimized for the common case of internet use. This means the suitability for the specific case of widget communication is by no means guaranteed. Nonetheless it is determined that the HTTP vocabulary is sufficient to allow widgets to serve as components in a web mashup application.

The HTTP protocol has six major verbs: GET,PUT,POST,DELETE,HEAD and OPTION. Those verbs specify the basic instruction set for accessing and modifying Web resources as well as resources provided by a REST service. For example to fetch data the GET request would be sent and to modify data the PUT command would be issued. In fact, the first four of those commands are analogous to database operations with SQL[27], which also only relies on a small set of actions.

ACTION	***SQL***	***HTTP***
Create	*Insert*	*PUT*
Read	*Select*	*GET*
Update	*Update*	*POST*
Delete	*Delete*	*DELETE*

Those four HTTP verbs are enough to create, read from, change and delete resources. The last two verbs are used for gaining information about resources. The request OPTION is sent to find out what kind of methods (for example GET,PUT or DELETE) a resource will answer to. Finally HEAD delivers the meta data from the resource, not the actual representation.

To make a web service RESTtful means that it should be structured according to the foundation laid out in the REST dissertation and adheres to the rules prescribed there. Note that the dissertation is written at such a high level that it would be possible to design a RESTful system without the use of HTTP protocol. Such a RESTful service places restrictions on the effects of the verbs. Some actions are required to be safe and idempotent. The GET and HEAD request should be safe in the sense that it only does what its name suggests. That is those actions should read data from a resource. Reading from a resource once, many times or not at all should all have the same effect. A GET and HEAD request should never modify any data. The PUT and DELETE operations should in turn be idempotent. This means that the first, second and all following request all leave the resource in exactly the same state.

Those rules are designed to facilitate communication over unreliable networks. If the communication fails, then the application can retry and send the request again. The request should always to have the same effect.

For widgets and a widget engine the use of the HTTP protocols is seen as as valuable tool for offering capabilities to offer and combine data sources. Thus, this is the foundation

for enabling mashup creation. The REST communication rules are said to be designed for unreliable networks. This is of importance as a mobile device connection should not be viewed as particular reliable.

Data elements
The resource component in a REST system is a component responsible for managing a number of representations of its data. In its low level form the representation will just be a stream of bytes and some additional data that describes those data. This metadata description gives the flexibility to define several representations of the same resource. The actual data format of a particular representation is called media type in REST terminology. Such media types could, for example, be text, image or XML data. Some of those media types have their primary use as data for other software components while some are intended also for human use. According to the REST view of data modeling any piece of information that can be named is a potential resource. The resource provides the means to map a request to a particular set of entities. Those entities holds the values the resource represents and is called representations. Note that the REST design allows the mappings of resources to empty sets of representations. The original work describing REST points out that the only demands put on resources are that the semantics of the mapping to the representations never change. Requests for resources or responses sent back includes control data. This kind of data element is introduced to provide control over the purpose of the message sent or received. It specifies the action being requested or how the response should be interpreted. The question of weather to cache or not cache data is also answered by control data. The final type of data element is the resource identifier which is the location of the address specified as a URL. Together with representations resource identifiers may also be put into the set of entities a resource maps to.

Components
Components in REST terminology is typically more high level applications for initiating and responding to requests. The REST style declares four main components, each of them serving a specialized purpose. The origin server is the source for representations of the resources and it is important that this component is the final recipient of requests that modifies it resources. The gateway component may be used by the origin server for such purposes as data conversion, security, and provision of interface to services. The user agent initiates a request. It uses a client connector component for this purpose and must be the final recipient of the response sent back. User agents come in many types, such as web browsers like Firefox and Internet Explorer. The last connector type, the proxy, is used as both a client and a server connector. The proxy component is used by the client for such functions as forwarding, data translation and security enforcement.

While the REST theory is particular articulate about the clear division of system into clients and servers, the end result for the implemented engine does not clearly separate the roles into a strict client server relationship. This as the mobile and server engine share the same structure and have the same capabilities. The very widget itself constitutes a blend between a user agent and REST data elements. Widgets can acts as clients and simultaneously as containers for web resources.

Connectors
The role of those components is to provide an interface for enabling component to component communication. The main components are client and server. The client component initiates contact while the server contacts waits for and responds to HTTP requests. Any component may include any number of client and server type connectors. A third type of connector is the cache connector. Its purpose is to save responses indicated as cacheable for reuse in future requests. Fielding locates the cache connector as part of either the client or the server connector components.

When resource identifiers need to be translated into usable network addresses this is done by the resolver connector type. The last type of connector is the tunnel. It is pointed out that this type of connector typically handles low level communication across firewalls and lower level gateways typically not needed for describing a REST system. But the tunnel connector is included in the REST since rest components may at any time choose to behave as a tunnel. The tunnel and resolver connector type are not directly used in the implemented widget engine. Though they would have been ideal canditates for solving handling address translation for connected mobile phones, which lacks a proper static address. Instead this functionality have been mapped to a separate standalone gateway component that both mobile widget engines and server engine interfaces with.

RESTLET
The Restlet framework has been around since the year 2005 [44]. It consists of two main parts. First there is an API geared for creating RESTful services. This part is considered standalone from the underlying Restlet implementation. Together with the framework a reference implementation is provided in the form of the Noelios engine. The use of a different implementation is as simple as removing the JAR file for the reference implementation and substitute it for another. Upon startup this new servlet engine will be loaded automatically. The reference implementation package is distributed under open source license.

The creators of Restlet states that this separation can be compared to the use of the Java Servlet Api and servlet containers like Tomcat [4]. The applications created with this framework can be deployed on a servlet container supporting the restlet API or they can also run standalone without a restlet container. This similarity between ordinary servlets has its roots in the fact that the restlet project was initiated as a replacement for the standard servlet API. It appears that the goals were twofold, first to create a framework for facilitating the creation of rest based web services and also to offer an replacement for the client server communication and data handling offered by the existing servlet api.

The restlet framework is organized around a set of components that is used to build up the application. Those components are named after the building blocks from the original REST dissertation [14]. If one has a basic understanding of the original REST terminology and concepts this naming strategy is helpful since the components are generally designed to work in the same way as originally described.

In addition to those components prescribed by Fielding's dissertation [14] the REST framework also introduces a few new ones. The purposes of those are to tie the others, like resources, components, and connectors, into a restlet application. Some of those added components are:

Restlet
The restlet is the implementation of the interface provided for methods responsible for dealing with requests directed to resources. As expected, protocols supported include HTTP but also FTP, SMTP and communication with databases.

Application
The application object is the manager of one or more restlets and is intended to provide common methods and data needed for those restlets which are placed under the control of an application component.

Router
The Router concept was not part of the original REST dissertation. The purpose of this class is to map each incoming request to the right handle method of one object of restlet class. The router can also be used for forwarding calls to proxy servers for dynamic load balancing. The specification of URI´s for the resources is made flexible with the use of parameters that stand for a segment part of the URL that can vary. The parameter names and their actual value can then be read when the resource processes an incoming request and choose what action to take. The concept of a special router class have been used in the widget engine. First the engine use the router class to direct requests to the receiving widget, then each widget does contain their own router which finally maps each request to the right web resource within the widget.

Notably in the architecture provided by a system created with the Restlet framework is that it abandons the notion of separate HTTP client and HTTP server parts. While the API provides a high level language for creating REST systems, the details of the requests and responses are never hidden. It is possible to have access to and manipulate the raw HTTP headers. The Restlet project is a work in progress and one recent addition to it is the support for creation of applications according to the JSR-311 specification. This brings it closer to the other framework covered in the next section.

JSR-311

The JSR-311 is also an effort to create a dedicated framework for creating rest services with the Java programming language [45]. This project aims at a more high level support compared to Restlet. This is a much more recent proposal than restlet and in some ways has been inspired by it but it is stated [28] that the goals are not to replace restlet but offer an alternate, more standardized high level frame work.

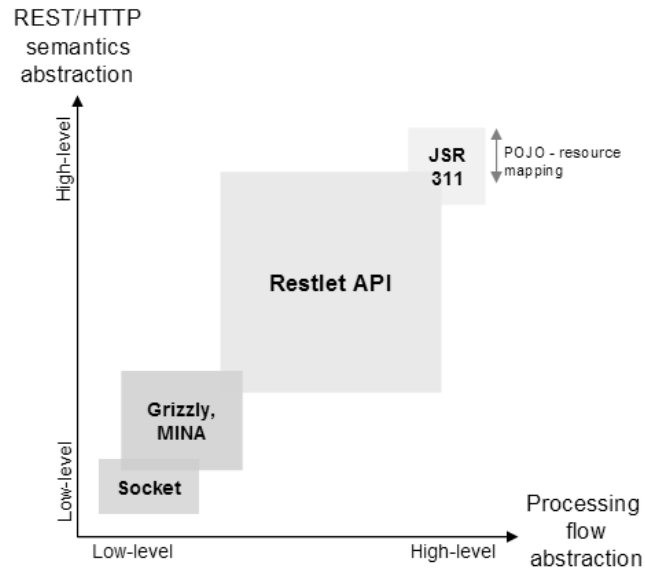

Figure 5-1 JSR-311 & Restlet comparison

In figure 5-1 the difference between JSR-311 and restlet is illustrated. The level of abstraction regarding the REST/HTTP semantics ranges from raw HTTP headers to high-level representation negotiation. The processing flow abstraction, from the raw network sockets Restlet builds upon to the framework for describing resources offered by JSR-311 [31]. Note that while Restlet is a complete solution in the case that it also provides a container adapter to run the framework in the goals of JSR-311 is to provide an API and the gap towards the servlet have to be bridged by an servlet adapter.

A reference implementation, like Restlet published under an open source license, is provided and is called Jersey. This reference implementation follows the direction and evolution of JSR-311 and is at the time of writing not considered finished.

The creation of resources that would respond to a GET request back to a caller would with Jersey be written like this:

```
@Path("/helloworld")

public class HelloWorldResource {

    @GET
    @ProduceMime("text/plain")
    public String sayHello() {
        return "Hello World\n";
    }
}
```

The JSR-311 relies heavily on Java annotations and the full description is given in [5]. In the example shown the resource contains three annotations. On that specifies it´s path, one that instructs it to return the message in the *text/plain* format and finally the @GET annotation directs a HTTP get request to be handled by the method *sayHello*.

Resources are declared as plain Java objects that has at least one path annotation or a request method designator. The Jersey implementation offers life cycle management for resources, such as per request life span. Like Restlet this framework also offers the possibility to specify path parameters as illustrated by the following snippet:

```
@PATH("Persons/NicePersons/{id}"
```

Here *NicePersons* is a sub resource of Persons. For specifying subresources this is directly done inside a class annotated as the parent resource.

In addition to describing the annotations the specification also covers the details of the algorithms for matching a request with a resource. Those matching algorithms are needed because several parameterized URI´s could match the same request, for example @PATH(*User/{id}*) and @PATH(*User/{name}*) both matches the URI *User/John*. This information is included to provide implementations with the information needed for translating URI templates to regular expressions in a consistent way.

The annotations are inherited by a corresponding subclass or implementation class method. To override inherited annotations a subclass simple specify its own annotations which then take precedence over the super class annotations. As previoulsy mentioned the design and implementastion have been largely influenced by the Restlet framework. The JSR-311 and the corresponding reference implementation Jersey proved somewhat difficult to model on the mobile widget engine version. This as annotations are not available on for Java ME. The end result was that it was not very straightforward to mimic the structure of an Jersey application on a mobile phone.

As can be seen in the examples the use of annotations provides a high level mapping between the REST concepts and ordinary Java objects. A similar strategy is used to create a bridge between the underlying server engine Jersey would be deployed on. Here an annotation defining the actual context the application is deployed in can be used to inject HttpServletRequests to a resource, so the request passing through the servlet are streamed to the resource.

The JSR-311 specification is aimed at providing a standardized process for not only creating RESTful applications in Java but also specifying how they are deployed. There is only one allowed way of inserting new resources. Resources are defined at startup and since they are given their URL by annotations their paths can be seen as hardcoded at the system startup. There seems to be no standardized way of inserting user defined resources at runtime. This may be a limitation when designing system that would allow users to freely add widgets at runtime, if each widget corresponds to an annotated resource in Jersey.

Conclusions drawn from component analysis

Two of the concepts should be retained and are suitable both for fine scaled component as an individual widget as well as part of a larger supporting framework. The concept of

URL and the prescribed use of the HTTP verbs could be used for such roles as giving each widget their unique identity. The HTTP action verb is seen powerful enough and can form the base for the interwidget communication. It is worth pointing out that for the context of unreliable networks and application of special rules for the use of the basic verbs is applicable. Communication with a mobile device should not be viewed as reliable.

The study of JSR-311 and the reference implementation Jersey as well as Restlet were initiated to find suitable server side components. An investigation was also made in order to evaluate the feasibility of porting parts of one or the other to the JavaME platform. A component that is not used, but would have been very useful for mashup creation is the filter component found in Restlet. The concept of HTTP stream modifiers are presented in [14] and this is modeled as the filter component by the Restlet framework. This component is seen as valuable in the mashup context as it enables built in support for modifying contents when redirected to resources.

The resulting engine does not constitute a port of any of those two frameworks. Instead the design and functionality have been influenced by the REST components and the implementation of Restlet. Especially the structure of a Restlet application, which is one of the components provided, has been particular influential. As Restlet is closely modeled after the original REST paradigm a large part of the design work has been to partition useful components into widgets and surrounding supporting engine structure. This and the question of how to expose REST concepts or if they should be exposed is the topic of the next section.

5.3.2 Analyzing REST Effects on a Widget Engine

A system based on the representational state transfer architecture will naturally embrace the concept of the resource as one of its central design blocks. However the resources themselves are not enough to make up a fully functional widget engine that allows widgets to function as a rich internet application. The resources need to be maintained, accessed and displayed graphically. The resources are an abstract notion and this contrast with the notion of a widget as an small application displaying evoking graphics and provides the means of using the phones capabilities for sharing user generated content on the internet.

When is the REST architectural style useful? Some suggestions given in by Sameer Tyagi in [27] and are as follows:

1. The interaction between the client and server can survive the restart of the server. This means the web service is completely stateless.

2. Producer of and consumer of content must have an understanding of the content sent. This because there is no formal way to describe the actual interface since the interface is the same for all rest web services.

3. There is a need to conserve bandwidth. REST typically adds very small information for headers and additional information compared to XML RPC.

4. The caching mechanism of REST can be used for improved performance.

How do these guidelines relate to a widget engine, suitable for a mobile device? Number three and four directly addresses mobile device considerations. The inherent caching mechanism could be used to provide off line capabilities for the mobile widgets. The need to conserve bandwidth is ever present for mobile devices. The second argument is determined to be neutral for the widget engine. The mutual understanding of the content passed is considered to be part of either system documentation or practices evolved within widget developer communities. The critical topic is the first because it addresses the client server roles in the system. Furthermore it imposes an important constraint on the system. The interaction needs to survive a restart of the server. This constraint is applicable if the system is modeled around a RESTful web service with widgets as clients.

The role of the resource in a widget engine is not directly clear either. There exist a number of choices for the meaning of a resource in a widget system. First it is possible to view the resource as something that either the widget engine consumes on behalf of the widget or the widget consumes itself directly. If this is the case then the resource means the widgets are clearly separated into a client role as consumer of information from a server offering a REST interface for its data and services. This would give an overall system architecture based on the client-server model.

If the widgets or via the widget engine are also capable of delivering representations of REST resources upon requests then the former separation into client-server roles becomes less distinct. This also means that the widget themselves are able to offer resources, to actually deliver internet content to consuming clients. This option could be realized by letting each widget engine beeing able to be a REST resource web service. The resources would then be the widgets and they could in turn act as client to a widget engine. Thus the mobile widget engine is a functional replica, although limited in performance, of a typical server side REST web service. This approach was chosen for the prototype.

The widgets as resources

Widgets as REST resources means that they all will have unique URLs that identify them and can be used for communication. This is determined to be a reasonable approach in combining the needs for widget communication with the components offered by the style of REST. This will also mean that the widgets are an active part of the system design that also encompasses the components that makes up the REST system. Parts of widget code will in this case contribute to the functions of a traditional RESTful web service.

If the widgets are REST resources then their usage is somewhat different from widgets deployed on contemporary widget engines and this raises the question whether widgets should be true REST resources or a system component that can act on and manage REST

resources. This is the question of the REST resources as originally described in [14] are suited for performing the kind of work normally associated with an computer application. The needs of an widget can be described as need for accessing mobile keyboard, camera graphics and loading, sending of data. The need of a REST resource is that it should have no side effects. There could be a possible conflict in this, together with evaluating what developing framework to present the programmer with, a REST based or not that motivates a discussion of how to use REST for widgets and what parts of it to expose to the programmer. If the widget, viewed as an application, would have to obey that rule it would impose a limitation of its usability The solution would be to separate the concepts of a widget and REST resource in the architecture. Some components are widgets and some are REST resources.

Following the layout of a Restlet application this approach was chosen for the widget engine. The widgets are implemented as a collection of classes working together. One class is responsible for implementing the REST resource interface and is related to separate widget classes that are responsible for connecting the resources with specific widgets.

The chosen approach is not free of possible conflicts. The REST paradigm reveals no suggestion how to structure the individual running components that uses and produce REST resources. The REST architecture is aimed at describing more large scale interactions than a few widgets interacting and offering a limited set of custom designed interfaces, based on the HTTP actions. The question is if there are any constraints imposed on the actual widgets by integrating them so closely with the REST architecture. Most important is how they should behave at runtime so they do not violate the REST principles to the point that the REST architecture is no longer motivated. Possible problems may appear as construction a system prone to race conditions and deadlock situations. This as the struggle to combine typical needs of widget engine, lifecycle control and access control with a need to program parts of the widgets as REST resources obeying the REST design principles. Special attention may be required to manage possible resource access conflicts and dependencies.

Since the caching mechanism is an essential part of any REST system this motivates a discussion of the relationship between the cache and the widgets. Since the widgets are envisioned to be linked together in chains where one provides the input for one other it is not directly clear how the cache should be structured to provide maximum performance. This is because there is a choice between caching raw data that may be used by many widgets for processing and caching of the processed data.

Illustration of how components, individual modules of the application, can achieve violations of a RESTful architecture are introduced in [32] and further elaborated in [33] In the latter paper the authors states that violations arose because web services tries to offer much far more fine grained services than delivery of hypermedia content originally envisioned. This paper also discusses the relationship between common AJAX mashups and the REST style. The definition of a mashup is formalized in [33]. With this formalized definition it is shown that the AJAX mashup can be viewed as a resource

conforming to the REST principles. Since the effects of the mashup is to move the computational context from the server to the client, latencies and server load are decreased. The study of the relation between resources and linked applications has motivated the introduction of a new architectural style. The authors name this style Computational REST (CREST).

The CREST style relates to the widget engine in two areas. First it indicates that achieving a system that conforms to the REST principles to the degree that it may be denoted RESTful may also easily at runtime behave in ways that contrast against those principles. Thus the REST principles are used for solely for design guidance and inspiration in the prototype design. Secondly, as CREST addresses the issue of REST and mashup applications at the large and in detail propose a way to also let resources contain computations it is viewed as a base for further investigations for finding a suitable programming language for the widgets. This paper makes an contribution in the area of relating a relatively new type of application, such as the mashup, with existing design theory. While CREST introduces new topics it should be noted that it is also possible to hold the view that the mashup represents a traditional facade pattern [34] as its role is to offer a simplified common interface towards a group of data sources.

Relying on REST interface

Early on the decision was made to assign a URL to each and every widget. This decision has evolved into making parts of the widget corresponding to REST resources and also exposing the REST interface to the widget programmer. This decision is based on the assumption that it would simplify the implementation by creating a single structure instead of modeling widgets as only clients accessing resources offered by a separate system. Thus the widgets makes use of the HTTP vocabulary and offers the resource programmer direct access to raw HTTP headers and data streams. The decision not to hide the underlying implementation was motivated by the HTTP verbs, or actions, being enough powerful to enable useful interwidget communication.

Furthermore it was concluded that this structure would be suitable for mashup creation, especially if combined with a widespread format for graphical content. This since it would not only allow linking existing widgets into chains for new or extended functionality but also creating a link between widgets and external web services offering a RESTful interface. Contemporary demonstrated mashup examples show a number of limitations. Once a mashup is created it is often impossible to easily adopt it to use new data sources and they are limited in the number of data sources they possibly can interact with [35]. The adoption of the HTTP protocol as the primary language of the widgets together with the concept of widgets as REST resources is seen as a base for overcoming such limitations.

If believed that the imposed REST style and terminology adds too much overhead by introducing concepts not directly related to contemporary use and development of widgets, then a more suitable framework would have to be developed around the base

framework. However such a framework, a suitable API for widget specific functions, would then be, isolated confined to this specific widget engine.

Mobile Devices as platform for Web Services

Observed existing widget engines on the mobile are rooted in a tight clearly client server bond, some relying heavily on the server for logic processing. A REST inspired mobile widget engine would not have to be restricted to the client role. Since the mobile engine would have the capabilities to act on itself as a platform for a web service. The limiting point here would be the difficulty for other to access the mobile since it does not have a fixed internet address. One solution to use a HTTP server on a mobile is presented in [36]. The conclusion in this report is that there is no major obstacles in preventing an implementation of an HTTP web service on a mobile device apart from possible cost issues for the user of the phone.

Placing a working replica of the serverside widget engine on the mobile version of the engine is possible. It is noted that there currently exists several obstacles that prevents a straightforward approach. The current uses of mobile phones are as clients.The mobile clients are not assigned a fixed IP address. Most operators configure firewalls in such a way so they prevent all traffic that is not initiated from inside the operators network. This has the effect that HTTP requests can not reach a HTTP server located on a mobile device [36].

6 Prototype Implementation

6.1 The Realized Widget System

The realized widget system consists of two core parts. The widget engine and a gateway connection component. The widget engine provides a framework for running widgets and provide support for widget intercommunication. Two versions of the engine have been implemented, one for the phone and one for the server. The widget engine's main roles are to serve as the platform for data traffic between widgets and manage the binding between a widget and its URL. The mobile version of the engine has been supplemented with graphic capabilities. This in the form of a HTML renderer component and a few widget associated classes for interfacing with the renderer and Java ME graphics.

The other main part shown in figure 6-1 is a gateway that enables communication to and from a widget engine which resides on a server to the widgets hosted by a mobile widget engine running on one or several mobile devices.

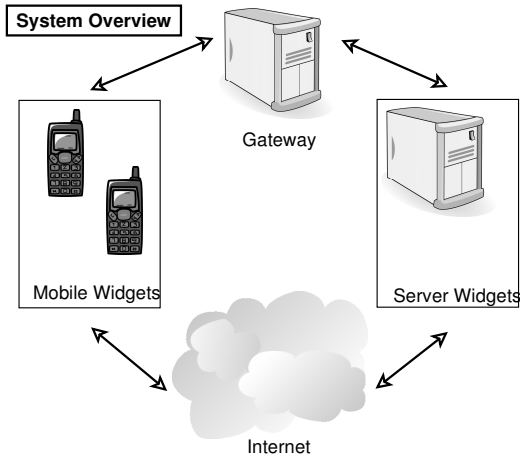

Figure 6-1 System Overview. Widgets are able to run on both mobile phones and the server.

The gateway's primary task is to enable the push mechanism prescribed by the requirements. As a beneficial side effect of the design it also enables the use of a URL for adressing a widget. This means it is possible to use a web browser on a desktop computer for accessing a widget running on any connected mobile device. The server engine and the gateway are deployed on the server as servlets. Two different servers have been tested, the gateway are deployed on the Glassfish server and makes use of its Comet module for providing push functionality. The engine located on the server share much in common with the mobile version and is injected on the server with the use of an HTTP servlet adapter module.

The widgets have evolved into a distinct architecture themselves. The definition of a widget contains in the server, the router and possible client components. As well as the widgets are the containers for information content, delivered trough resources each having their own URL. While not clearly visible in the illustration 6-1 is the fact that the structure is composed and implemented in away such that it blurs the definition of client and server roles. The implementation of the mobile widget engine contains also a workable HTTP server component making it possible to say that the system is designed to work in a peer to peer mode instead of a strict client-server hierarchy. Also not directly visible is the fact that the implementation has lessened the definition of widget role and widget engine role. This is further discussed in chapter six. Due to the fact that the widgets have evolved into a quite sufficient structure in itself for containing communication and data storage capabilities their need for support from the widget engine components have been reduced. This invited to the experimentation of making the system as a system of widgets. Thus widgets have been tested and set up to provide an illustration of how widgets themselves can constitute the interface to the phone functions such as memory storage and built in browser.

The Mobile and Server Widget System

Two versions of the widget engine have been developed. As it was desirable that the widgets should be able to run on both the server version of the widget engine and the

mobile counterpart, the engines share much in common. The system may as a whole be viewed from the outside as a platform for web service both on the server and from the mobile. However, this is not the main task of the system. The structure can be said to be inspired by the theory behind REST. Notably it contains components such as client, server, resource and the concept of a router borrowed from the Restlet framework. The role of those components have been combined in a way it might be argued is not fully aimed at creating a RESTful system, and this was not the goal of the prototype. Instead the use of those components have served as a starting point for exploring possible design paths and define capabilities of the widget engine and the individual widget.

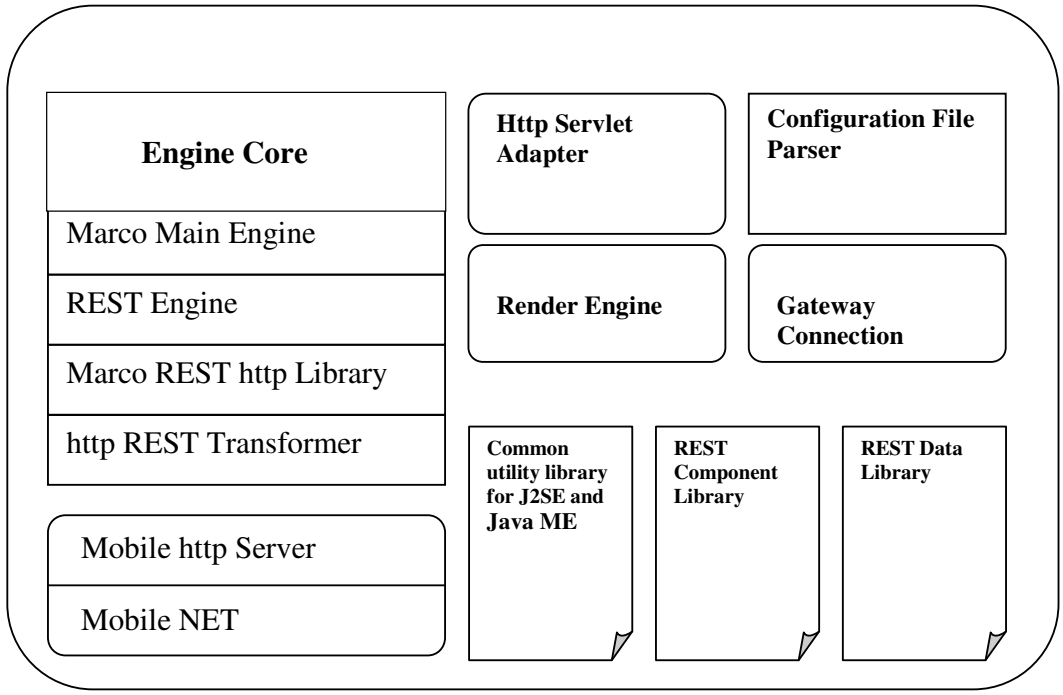

Figure 6-2 System Overview

The core of the widget engine, as illustrated by figure 6-2, is constituted by an implementation that provides a conversion between HTTP requests and a corresponding internal request format. The HTTP REST transformer receives an request and makes use of the classes in Marco REST Http Library for converting the request into the internal format used. Thus the implementation relies only in its endpoints of standard Java classes for HTTP communication. The introduction of this internal HTTP representation is motivated by the need to have a common interface for both JavaEE and JavaME. In addition to implementing a common HTTP interface that could be used on both the server and the mobile version there is also a small set of utility classes for lists and hash map functionality. Again those were needed to provide some minimal support for widget development on both platforms.

On top of the base communication framework there is a module for loading and initializing widgets as well as managing the HTTP communication between them. This is implemented by the class REST engine. After widget initialization the main functionality

of this class is to dispatch requests and let the responses go back to the caller. This class illustrates the spartanic capabilities chosen for the engine. The engine basically serves as a handler for HTTP request and delegates them further to classes which are part of the widget structure that contain individual server and resource components. Both the REST engine module and the individual widgets makes use of a set of components, given in figure 6-3, that is the result of the REST theory and REST framework analyses. The router and finder classes are borrowed from the Restlet framework and serves the purpose of matching a widget against a URL template patttern and direct the request to the proper destination widget.

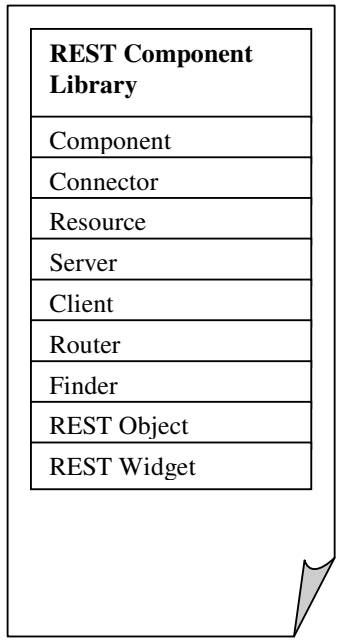

Figure 6-3 System classes inspired from REST theory and the Restlet framework

The rounded boxes in the system overview given by figure 6-2 represents those components that are only implemented on either the mobile or the server version. The Http Servlet adapter are needed on for the server engine, it allows the engine to communicate with a servlet. The server engine makes use of a servlet for passing incoming HTTP request to it. The other components are for the mobile version of the engine. The gateway component implements the connection to the server side gateway. This component enables the phone to establish a connection to the gateway and this connection is not closed. The server side gateway are then able to match segments of request URLs to the corresponding phone. Several phones may simultaneously be connected to the gateway and a table of connection and URLs are kept. The URL segment identifying a phone consists of the phones login identifier, for example a username.

The render engine component are discussed in the following section and consists mainly of an HTML page render and a mobile HTTP server. The HTTP server have been modified so it interfaces with both requests originating from other widgets and also from

requests sent through the gateway component, this to achieve a channel for push based data transfer. The use of an HTTP server on LCDUI mobile devices requries some attention. Because the widgets should be able to communicate with each other while they are located on the serve or on a phone the case where two widgets located on the same phone demands a loop back connection. The implementation of Java ME connection classes most often assumes that the connection is between the device and a external server. For this implementation it was necessary to both handle the cases were connection is between external server and an server deployed on the device itself. Connecting to the server on the device requires a loopback connection. Use of standard Java ME loopback connectivity are device dependent. It is not guaranteed to work on every device. Because of this fact a set of classes, the Mobile NET component in figure 6-2, were developed. The purpose of those are to enable either standard network connectivity or loopback socket connections based on piped streams.

6.1.1 The Graphics

As previously stated, graphic capabilities have only been implemented for the mobile version. The widgets use a simplified version of standard HTML. For testing and developing the system and widgets a standard web browser have been used for displaying contents both from mobile and server based widgets, but implementation of graphics for the server side component have not been pursued. The decision to use HTML is rooted in the fact that a HTML render engine, developed within Ericsson, was made available. This render engine has been added to the base widget engine structure as a standalone component for the mobile version.

Figure 6-3 View from example widget. This view consists of a single HTML page.

Figure 6-4 The Widget Dashboard, Browser and weather widgets are shown

The render engine was from start essentially a page renderer, displaying and processing input from individual HTML pages. Parts of the render engines code have been modified so it allows the rendering of several HTML pages on a single displayed screen. In figure 6-4 a sample screenshot from the mobile dashboard is shown. Here two widgets are

displayed, the start button for the phones browser and a weather widget. Both those images are separate HTML pages.

A slight modification is introduced in the HTML format; the widgets may use variables in the HTML document that are substituted with current values at time of rendering. Note that it is not only possible to bind page elements to current data contained in data storage widgets. But this is also a mechanism for generating dynamic HTML contents by the widgets at runtime.

The version of the render engined used for the prototype did not have HTML DIV tag functionality so this was implemented to be the primary mechanism for enabling mashup graphics. This by enabling HTML DIV elements to have a transparent background so pages could be superimposed on each other. A method for substituting the background with an ordinary JavaME canvas class was tested. Substituting the HTML DIV background with a canvas class made it possible to evaluate the combination of HTML layouts with the standard mobile Java graphics.

The original input mechanisms of the HTML renderer were retained. Widgets have essentially those input mechanisms available in a page displayed by a web browser, see figure 6-5. This was determined to be enough for a demonstration purpose.

6.1.2 The Widgets

Widgets in for example Widsets, as described in the section of related work, or the Macintosh OS X to name a desktop example are very much complete in themselves in the sense that one or two documents of specialized language is sufficient to produce a widget that runs and is visible on the screen. From the very start the widgets where envisioned to be very small components which would require many more widgets in combination and cooperation with each other to achieve the same result. The requirement for widget intercommunication was seen as direct enabler for making a traditional widget that consists in itself of many small scale widgets, each of them only performing a simple task.

The proposed widget can be used for such purposes as data storage, interwidget communication, display of graphics which provides for interactivity with the widgets. Firstly a description of the general widget will be given and thereafter examples How the widget is used for graphics and mashup creation.

A complete widget consists of three parts together with any number of files describing the graphics. The widgets themselves have come to be structured around the components describing REST and also inspired by the implementation of Restlet.
The core components are the classes Widget, RestWidget and Resource. Those are linked together in the way that a Widget has a RestWidget which in turn may consist of any number of resources. Any resources as well as server, router and any number of client components are grouped together into a single component, see figure 6-6, and constitutes

the major part of the widget structure. As previously mentioned the very widget structure in itself are given server, router and client components.

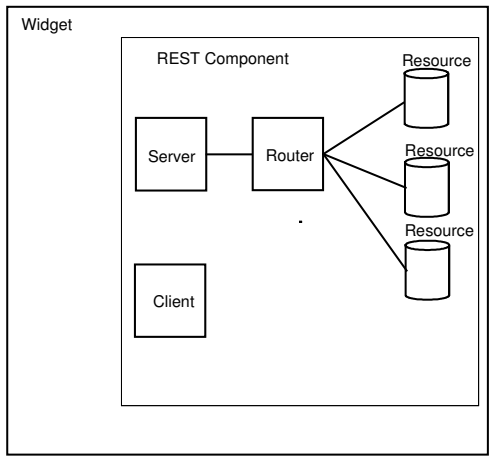

Figure 6-6 The structure of a widget

The individual parts of the widget have been structured to serve specific purposes. The REST components purpose is to let the widget function as part of a web service. Here specific resources are bound to URLs. It is possible to specify template patterns in the URLs. More specifically this class will set up the necessary information needed for the underlying system implementation to be able to match a incoming request to a specific resource. Example of source code for one weather widget is given in appendix E.

The place for actual logic code of a widget can be in either the resource or the widget component part. The widget developer needs to be aware of a fundamental difference between those two classes. The widget class is instantiated only once by the runtime. The

resource classes are instantiated every time a request is sent to it. If many requests are made for the same resource then many instances of the resources classes will exist at runtime. The developer will need to be aware of this fact to properly address synchronization of shared resources and concurrency implemented with Java threads. The developer needs to handle the situations were resource content are updated and simultanously accessed. The runtime engine allows for several requests for a single resource to be processed concurrently. The runtime will instantiate as many resource classes as needed. A possible scenario is that the engine are processing several GET requests and a POST or PUT request changes the resource contents. This has the effect of generating several instances, at runtime, of a resource. The problem with this is that the actual content of the reosurce may vary between the instances.

In the example given below, figure 6-7, the RestWidget that is part of a weather widget is shown. This particular widget is responsible for managing five individual resources. Note how a URL can be bound to one or more Java classes. The first two resources *setNewCity* and *city/{name}* both points to the same class. In the second case the URL is parameterized so it accepts any city name.

```
public class GoogleWeatherRestWidget extends RestWidget {

                public boolean enableRest() {
                        return true;
                }

                public void initRest() {
                  //Widgets server & router component
                  Server server = new Server(getWidgetComponent());
                  Router router = new Router(getWidgetComponent());

                //REST resources offered by this widget
                router.attachResource("setNewCity", ForecastIconResource.class);
                //Get weather forecast from default city or by parameter name
                router.attachResource("city/{name}",
                ForecastIconResource.class);
                //Page to show when widget is selected from dashboard
                router.attachResource("index", GoogleWeatherIndex.class);
                //Used to receive messages from SystemWidget, if this widget
                //subscribes to any messages from aparticular widget those
                //messages will be directed to this resource
                router.attachResource("receiveNotify",
                ReceiveNotifyResource.class);

                server.attachRouter(router);

                this.addServer(server);
                }
        }
```

Figure 6-7. An example of server, router and resource binding to a widget

Visible in figure 6-7 is also how the widget itself mimics the structure of a web server. First the widget is given a server and to this a router is added. The resource URLs are relative to the root of the widget. Every widget will have its root specified in a configuration file which the engine uses for starting up and initialize all widgets.

Supporting the Model, View and Controller model

The widgets can be used as the parts of the model, view and controller (MVC) paradigm. The first step towards this was to introduce an addition to the HTML format. The addition allows to directly display data from a resource. This creates a direct link between the data held in the resources and the view. A widget serving as a data repository can send a message to the view part for updating the display. The actual update of the display is accomplished by the developer with the following lines of code:

```
//how to update a page
getWidgetRef().getIndex().render();
this.getMarcoEngine().getDashboard().getIndexPage().repaint();
```

The developer has the choice of updating the widgets index page or the dashboard icon. The graphical elements of the widget consist of *icon* and *index*. The first is responsible for displaying the widget on the dashboard and the second is for the large display when the widget is selected. The icon class is responsible for displaying the widgets icon on the

dashboard and the index icon display widget content when the user selects the widget from the dashboard.

Following the idea that the engine could be augmented by widgets offering services a sample inter messaging widget were developed. This widget augments the support for MVC development as it allows widgets to register as subscribers to other messages originating from a given widget. The message widget the acts as a distributor of messages from the destination to all receiving widgets. This is illustrated in figure 5-6 where a widget implements the *receiveNotify* resource meaning the messaging widget will deliver any messages to this resource.

7 Evaluation

7.1 Enabling Widget Web Applications

The term mashup in the context of widgets refers to the technique of creating new content by means of combining one or more existing sources of information into a new service. This process usually involves taking existing components and extracting information and combines it without modifying the original components but instead using them as building blocks for a new application.

The requirement for widget intercommunication is a direct enabler for mashup creation. The developer are able to combine widgets and intercommunication widgets lend themselves naturally to be linked in chains. A widget is instructed to fetch data from one widget and then send it to another for further processing. The mashup strategy works in the way that data are fetched, processed, combined and overlaid with other data. Intercommunicating widgets have themselves the structure to accomplish this.

The technical foundation for enabling widget intercommunication as well as support for widget communication is fulfilled with the adoption of a framework centered on HTTP communication and the use of URLs for identifying and accessing widgets.
The REST architecture has means of sending messages, retrieving data and it divides the system broadly into data sources and channels for data flow. The next question is how the implementation may make use of this foundation?

The prototype has technical means of support for creating mashups of different combination of widgets and data sources. Relying internally on the REST principles for storing and accessing data is not viewed to be enough. For enabling creation of mashups there also has to be a way of reusing different widgets in any combination. Now the question is raised how should the application support mashup creation? The prototype system currently supports this directly trough direct use of REST resources and manipulation on them. Separation into data and logic creates two distinct components for later reuse. The logic parts are implemented in a separate class and it is possible to

structure a widget as a two layer system. Thus both resources and the logic classes can be viewed as separate entities for reuse.

Direct exposure of resources and HTTP protocol details has increased the number of classes needed to make one functional widget. There is a tradeoff between increased developer work and the possibilities of reuse. Note that with our current system there is a difference between actually programming for reuse and not to program for reuse. A widget intended for reuse would try to split its data and views as much as possible into different resources, possibly implementing them into a number of classes to reduce code dependencies, so parts of the widget structure can be reused.

If the underlying mechanism of REST had been hidden from the programmer then an API for offering the same transfer and processing of data and media information would have been needed. This would have made it possible to increase the level at which the developer would write the code.

The choice of basing the view data format on HTML means at least that the support for reuse of exiting templates and fetching data from other HTML sources are increased. But as illustrated by the need to actively introducing new techniques like AJAX to overcome limitations of HTML pages in web browsers, this also means the base format as such does not have particularly good support for neither rich internet application nor mashups. The customization of the HTML format was introduced to create a connection between the view and data residing in widgets. With this method it is possible to insert data, or other HTML pages, into an HTML page directly from the Java code. The prototype is capable of supporting structuring of the widgets according to the widely used model view controller principle. As the widgets that makes up a mashup may be located on any combination of a number of mobile devices and servers, this provides the flexibility for having the data and view part on separate physical locations.

The described flexibility comes at the price of increased complexity when it comes to develop the mashups. While a relatively complex mashup can be created with Google Map API in a single or few documents [37] this is not true for the developed widget engine. As previously noted the making of several small components that make up a workable widget means that a mashup of the same complexity as in [37] would consist of far more parts. The example widget given in Appendix E consists of several classes, and would only constitute one part of a possible mashup. It is clear that the number of classes needed for creating mashups with this structure would be quite large. Though the length of the widgets source code could be reduced with a proper API for sending and processing data such as JSON and common image formats. The number of Java classes required for a functional mashup type application is quite large. There exist a limit on the number of Java ME classes that may be installed on Java enabled phones and the chosen implementation does not help to keep the numbers of classes low.

7.2 Testing the Phonetop

The prototype is tested on Sony Ericsson´s JP-7 series phones. This series of phones have the ability to run an application on the phone's background screen. The normal wallpaper

background can be set to a midlet of choice. This accomplished by adding a property in the applications JAD descriptor and the application is available as background wallpaper. This method allows the widgets to display themselves upon startup. The dashboard is visible and the widget icons will change the content when updated with new information. However while the application is in background mode there is no way to interact with it. JP-7 background applications have some limitations. The user has to activate the midlet before interacting with it, and the midlet only has a short time to initialize itself and display something before the phone automatically enters energy saving mode and dims the screen. In addition, background midlets are assigned a lower priority by the phones application manager so while the user is not interacting with the widgets the runtime will run at a reduced speed.

7.3 Graphics

The HTML format includes some inherent limitations of the format, especially when not used in conjunction with a scripting language to enhance the capabilities. The layout and interaction possibilities are limited. This means that there are also limitations in the area of mashup graphics. The inclusion of a binding between HTML and data in the widget Java code has been implemented. The most severe problem is the plethora of device screen sizes and lack of input control from the widgets. The situation could be improved by further extending the HTML format so the size of HTML DIV blocks would be set to the values related to the current screen size of the device. It is also relatively simple to implement a link between the position of selected items on the HTML page and Java code in the widgets. This would be beneficial for mashup creation as it would allow new content, that is a new HTML document, to be rendered at the selected position of the screen. The graphics consists of two main separate items. The dashboard is rendered as a collection of HTML documents but once a widget is selected the view consists of a single HTML page. Although it is possible to overlay that page with pieces of HTML code a better solution would be to let all graphics be composed of collections of HTML documents. This would give increased support for mashup graphics but not solve the problem of achieving a good layout across a range of devices. As no scalable graphics are available it would most certain be be needed to create custom specific pages for different phone models.

8 Conclusion

The design input to investigate the REST architectural style has come from many sources. Originally it was introduced by the supervisor as the idea of each widget should have their own URL and the communication between them could be based upon REST principles. As shown by the prototype from Telefonica, covered in the related work section, and numerous suggestions on various weblogs from this and the previous year there is currently a strive towards investigating the suitability of REST in relation with mashups and widgets. In chapter five parts of the structure and building blocks of REST

and two available frameworks have been introduced. The resulting widget engine prototype architecture is made of those components. As they have been slightly rearranged and combined in different ways it may be proper to call the prototype structure a structural mashup of those three examples.

The mashup strategy has served the purpose of enabling a framework for widgets inspired on the building blocks of REST and the implementation. As covered in chapter five there did not exist a straightforward way to merge the design of the widget engine with the REST principles. It should be noted that this is not really needed; it depends on the degree of adherence to the REST principles that is sought after. The prototype design has been based on the layout of a RESTful web service, this has meant bringing this structure on to the mobile runtime and thus creating a base for using the mobile as a provider of web services. This is the outcome of a process that started with associating each widget with a URL and thereafter applying REST concepts and layout both to the widget engine and the widgets themselves. There are some limiting factors for running a web service on a mobile device; one of them is cost and battery life [36]. This would imply that a web service on a mobile device is not useful for serving the internet but is instead more suitable for sharing contents with a group of defined size.

As seen by the given examples of widget engines the scripting languages like Javascript and custom designed scripting languages are a workable approach for defining and running the widgets. The use of Java classes for the widgets places restrictions on the installation of the widgets on a MIDP2 device. In line with the requirements the widget engine and the widgets are compiled and installed together. The use of Java and not relying on a browser environment for running the widgets has meant that there is no limitations or need to create a bridge between the widget engine and the phone hardware. However the use of Java, and especially the open structure aimed at making the widget engine mimic a web service platform has some implications. First there is no clear division where widget creation and widget engine module programming start. This because there is the possibility to create new widgets that provide functions normally associated with a runtime engine, for example access the phones file system and camera. The second issue comes from the fact that using Java as the programming language for the widgets made the inclusion of runtime life cycle control problematic. The basic options are to use a combination of threads and timers for simulating concurrent execution of the widgets. The thread control mechanisms are limited in Java and the execution of threads is considered to be resource intensive on the mobile device. The Widset widget engine does not allow direct access to threads and offers a solution with functions that execute periodically, with the use of timers. One solution have been tried to offer a standardized way of declaring a widget as a process and then use standard Java practices to pause and resume thread execution. In this way it is possible to control the proportion of time invested in running widgets and the runtime engine. However, such methods are only a superficial solution as the widgets are free to create any number of threads and timers themselves.

The widgets are aimed at being able to share their contents via their URL. In the prototype there exists no mechanism for defining the computer applications, or widgets,

which are allowed to do so. This and the lack of a proper widget programming language, as described in the previous paragraph, means that the engine in its present form is not suitable for the kind of widget engine where anyone can freely write widgets and upload them for others to download. This contrasts to the majority of available widget solutions, for example Widsets which were covered in chapter three. This and the direct use of HTTP headers and requests means that the prototype are more suited for use in a closed setting, were a single or co working developers will create the widgets. Alternatively the strategy adopted for the prototype may be viewed as a low level platform that can serve as a general platform enabling widget intercommunication by means of HTTP and URLs. This platform can be built upon, for example with the inclusion of a dedicated widget programming language that makes use of the provided REST framework. There is a trade off between the flexibility and richness offered by the Java widget programing language and the greater access and widget execution control offered by a dedicated widget programming language.

8.1 Suggestion for Future Work

The suggestions for future work concerns finding suitable implementations based on the REST architectural style and analyzing if a dedicated widget programming language would be useful. As previously discussed the use of Java for programming the widgets have consequences for the division of system into runtime and widget components and responsibility. The combination of resources as widgets means there is no proper division into client code that retrieves data from resources. Applications that are commonly referred to as mashup can be analysed and structured according to the principles that extend on REST theory [33]. As illustrated in this paper an implementation of an application can easily break fundamental properties of the REST style. Each component in a REST system must ensure that the REST principles are not violated by the component. Thus, the chain is as strong as the weakest link. As the widgets, in this implementation, acts as both clients and resources it would be beneficial to study how a system would could be based on the computational REST style introduced in [33]. This means resources themselves are also viewed as computations making it possible to view resources, in this case widgets, as containers of both logic code and data that can be transferred.

The programming language contained in the resources could possibly have a syntax that would facilitate operations on resources. This language could for example be inspired by the functionality of Yahoo Pipes! [38], a tool that enables the combination of contents from various internet resources. A widget engine offering widget code and data as internet resources should be suitable for mashup creation and since the widgets themselves are plain URLs the system would be very open and accessible. This leads to the question of investigating how to design access control mechanism. This means designing the system so it allows for independent creation of widgets by developers and at the same time provide mechanisms that guarantee the privacy and integrity of stored data.

9 Appendix

Appendix A Mobile Widget Frameworks

Bling
Uses an engine that utilizes a virtual machine to be able to run the widgets on both JAVA ME and BREW platforms. Uses the AJAX style but does not run the widgets in a browser. Widgets are declared in a XML document that can contain ordinary Javascript. Data exchange with servers are based on XML. Bling has custom graphics for mobile audio and video.
URL: http://www.blingsoftware.com/technology.html

Moblets
Has offline capabilities, widgets store their data and state in a dedicated object called Cache. Based on the AJAX model. Does not use a browser to run widgets. The widgets are created using a combination of XML and Javascript
URL: http://mojax.mfoundry.com/display/mojax/Overview

Opera
Offers SDK for mobile widgets. The widgets run in the mobile Opera browser. The SDK offers functions for creating AJAX style applications.
URL: http://www.opera.com/pressreleases/en/2006/02/14/2/

Plusmo
Exists in both Java ME and phone browser versions. This Widget Engine is essentially a XHTML based microbrowser which is Javascript enabled. Around this a framework has been added for installing and managing the widgets on the mobile The engine furthermore has support for background downloads, local cache mechanism so widgets can run offline. The widgets themselves are created with a graphical tool.
URL: http://www.plusmo.com/homepage/home.shtml

WebWag
Offers a standard set of widgets for download to a range of mobile phones. The concept of a WebWag mobile widget is based on a standard web page. Provide a function that makes it possible to create a web page that consists of parts of other webpages. This page can then be downloaded to the mobile.
URL: http://www.webwag.com/

Widsets
See the related works section for details.
URL: http://www.widsets.com/

Yahoo! Mobile widgets
See the related works section for details.
URL: http://mobile.yahoo.com

Appendix B Mobile Widget Technologies

FlashLite
FlashLite is the mobile version of the familiar Flash product, from Adobe Systems, found on desktop computers. The system is proprietary and although the system has been successful on the stationary web it has gained success in certain geographical regions on the mobile web [40]. Flash Lite is the Flash technology specifically developed for mobile phones and consumer electronics devices. FlashLite accelerates the delivery of rich internet content and browsing. The product allows for a customized user interfaces. The core component of FlashLite consists of a rendering engine, which compromises visual elements for display on the mobiles screen. This is supplemented by the script component, which is responsible for processing events such as key presses, enabling dynamic interactivity in the application. There can also be specialized components for processing and displaying specific data types, such as certain image or video formats, or device data such as network signal or battery level. The final class of components interacts directly with the processing capabilities of the device itself in order to optimize overall performance. FlashLite is not only market for use on phones and mobile browsers but also for devices like MP3 players and widget stations.A project initiated by Sony Ericsson is underway to enable the use of FlashLite as a front end to JavaME applications. The intended purpose is to combine the use of JavaME API with FlashLite and enable direct use of the devices hardware [41].

JavaFX
This is a development framework offered by Sun Microsystems for creating rich internet applications. As with FlashLite it relies on the fact that processing and networking capabilities in mobile handsets have increased potential to deliver higly rich interactive internet content rich. As such it can be viewed as a direct competitior on the market to FlashLite. The Mobile version, JavaFX Mobile, is build on top of the ordinary Java ME platform, and has direct access to the phones hardware. Content created for this technology is intended to be viewed and used on a range of devices from the desktop, mobile phones and other consumer devices. A special scripting language, the JavaFX Script has been developed. This language has among other things direct support for key frame based animation.

SVG Tiny
SVG Tiny is a standard for displaying vector graphics on mobile devices behavior. SVG Tiny can be used in JavaME applications and it can be used for graphics on pages displayed by the phone browser. The scalable vector graphics makes it is easier to achieve good results when graphics are displayed on a range of devices all with different screen sizes and resolutions. SVG Tiny is a standardized format [39]. Version 1.2 of this format includes support for creating data model, data renderer, setting up communication methods and provides a scripting environment [40]. The SVG Tiny format is strongly linked to JSR 226 and offers simple methods to load and display SVG Tiny files, as well

as manipulate SVG Tiny content or simply create SVG Tiny content from data in the application.

JSR 290

This JSR enables creation of JavaME applications which combine established web user interface markup technologies with Java code, see figure 9-1. The specification is targeted at creating Java ME applications that combine the authoring and graphical richness of established mobile web user interface technologies, such as SVG Tiny, with the capabilities of the JavaME platform.

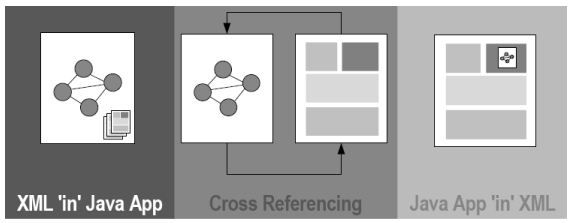

Figure 9-1 Binding between Java and XML [42]

This specification includes the use of the CDF (compound document format) specified by W3C. The CDF format has provision for screen size and form factor adoption as well as declaring interaction rules for the combined markup document. The specification is further intended to allow the use and combination of dedicated layout and content tools together with standard Java ME development methods, see figure 9-2.

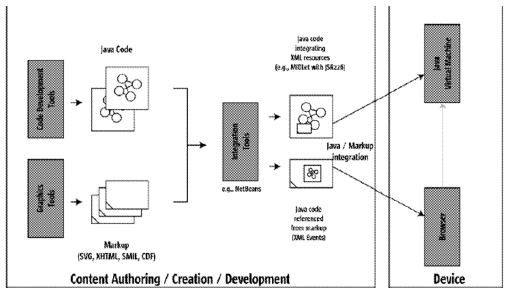

Figure 9-2 Creation process with JSR-290 [43]

Appendix D Widget View Definition File

This is the mHTML file describing the weather widget shown in the dashboard illustration from chapter five. Note the use of parameters for temperature value, image icon name and the name of the city. Those are marked in bold and constitutes the implemented binding between HTML and Java code. In appendix E the class *ForecastIconResource* is shown. This class contains a method `doGet(RestRequest request,RestResponseresponse)` and in this method, marked in italics the code responsible for providing this data are located.The icon size has to be manually set to a suitable value, disregarding the actual screen size which in practice varies between different devices.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.1//EN"
"http://www.w3.org/TR/xhtml-basic/xhtml-basic11.dtd">

<html>
<head><title>Weather Forecast</title>
</head>
<body bgcolor="#0A0A0B">

<div style="background-color:#AAABEF; position: absolute; left: 1px;
top: 1px; width: 98px; height: 98px; ">
<center><font color="#0000A">[city]</font></center>
</div>

<div style="background-color:#ABBBFF; position: absolute; left: 12px;
top: 20px; width: 75px; height: 65px; ">
<center>
<p> <img src="[picture]" alt="Some nice weather icon"> </img> </p>
</center>

<div style="position: absolute; left: 22px; top: 38px; width: 30px;
height: 20px; ">
<p> <font color="#000000"><h3>[temperature]</h3></font></p>
</div>
<div style="position: absolute; left: 21px; top: 37px; width: 30px;
height: 20px; ">
<p> <font color="#FA0202"><h3>[temperature]</h3></font></p>
</div>
</div>

</body>
</html>
```

Appendix E Example Widget Code

This example shows a widget complete with its dashboard icon. The intention of this example is partly to show a working sample of widget code but more importantly it illustrates the high number of classes that will be needed to create a widget with a number of resources and graphics. This particular widget downloads JSON weather data from a widget located on the server, responsible for XML to JSON conversion of the original weather data from Google.

There are three main classes for this widget. First there is the class *GoogleWeatherWidget* which is responsible for periodically updating the data. The class *GoogleWeahter RestWidget* sets up the resource structure and class *ForecastIconResource* delivers the graphics data for display on the dashboard. A resource class not shown is the *GoogleWeatherIndex* class, responsible in the general case for providing graphics for a selected widget and in this case it is also responsible for changing the default city for the weather forecast. Note that the widget is not complete, two more classes are needed and that is the widget Icon and Index classes that manages the actual drawing of the graphics on the dashboard and when the widget is selected from the dashboard.

```
public class GoogleWeatherRestWidget extends RestWidget {

	public boolean enableRest() {
		return true;
	}

	public void initRest() {
		Server server = new Server(getWidgetComponent());
		Router router = new Router(getWidgetComponent());

		router.attachResource("setNewCity",
		ForecastIconResource.class);
		router.attachResource("city/{name}",
		ForecastIconResource.class);
		// Delivers marHtml without html doctype & body
		// So it can be easily inserted into other pages, e.g. in a
		// html div
		router.attachResource("onlyMarhtml/city/{name}",
				ForecaseIconOnlyMarHtmlResource.class);
		// index page, used for changing default city name
		router.attachResource("index", GoogleWeatherIndex.class);
		router.attachResource("receiveNotify",
		ReceiveNotifyResource.class);

		server.attachRouter(router);

		this.addServer(server);
	}

	public class GoogleWeatherWidget extends MarcoWidget {

		private String cityName="London";
```

```
	public void init() {
		this.data.put("info", "Google weather data widget");
	}

	public void setup() {
		RefreshTask refresh = new RefreshTask(this);
		refresh.start();
	}

	public String getCityName() {
		return cityName;
	}

	public void setCityName(String cityName) {

		this.cityName = cityName;
		if(this.hasIcon()){
			// Update Dashboard Icon
			this.getIcon().render();
			this.getIcon().repaint();

		}
	}

	private class RefreshTask extends Thread {
		MarcoWidget parent=null;
		public RefreshTask(MarcoWidget parent) {
			this.parent = parent;
		}

		public void run() {

			while (true) {
				if (parent.hasIcon()) {
					parent.getIcon().render();

				this.parent.getEngine().getDashboard().re
				paint();
				}

				try {
					sleep(5*60*1000);
				} catch(InterruptedException e1) {
				}
			}

		}
	}

}

public class ForecastIconResource extends Resource {

	public boolean allowDelete() {
		return false;
	}
```

```
public boolean allowGet() {
      return true;
}

public boolean allowPost() {
      return true;

}

public boolean allowPut() {
      return true;
}

public void doDelete(RestRequest request, RestResponse
response) {

}

public void doGet(RestRequest request, RestResponse
response) {


StringcityName=((GoogleWeatherWidget)this.getWidgetRef())
      .getCityName();
      MarHtmlVarTable table = new MarHtmlVarTable();

      byte[] ba=null;
      String s="",temps="";
      int n=-1, i=0;

      try {
      // use helper function to download JSON weather data
      // from
      // widget on server
            ba=this.getData(new
      Url("http://127.0.0.1:8080/MarcoEngineServer/GoogleWe
      ather/City/"+cityName));
      } catch (IOException e) {
            e.printStackTrace();
      }
      // create JSON object
      n=ba.length;
      while (i<n) {
            s=s+(char)ba[i];
            i++;
      }

      JSONObject json=null;

      try {
            json = new JSONObject(s);
      } catch (JSONException e1) {
            // TODO Auto-generated catch block
            e1.printStackTrace();
      }
```

```
        String city="",humidity="",icon="";
        int temp=-276;

        JSONObject jo= json;
        try {
              JSONObject ja2
              =jo.getJSONObject("weather").getJSONObject("for
              ecast_information");

              city=ja2.getJSONObject("city").getString("data"
              );
              JSONObject ja
              =jo.getJSONObject("weather").getJSONObject("cur
              rent_conditions");
              temp=
              ja.getJSONObject("temp_c").getInt("data");
              humidity =
              ja.getJSONObject("humidity").getString("data");

              icon=ja.getJSONObject("icon").getString("data")
              ;
              } catch (JSONException e) {
              // TODO Auto-generated catch block
              e.printStackTrace();
              }

        StringTokenizer st = new  StringTokenizer(city,",");
        if (st != null) {
              city=(st.nextToken());
        }
        table.put("temperature", temp+"C" );
        table.put("city", city);
        table.put("humidity", humidity);
                          table.put("picture",
        "http://127.0.0.1:8080/MarcoEngineServer/GoogleWeathe
        r/getImage"+icon );

        RestMarHtmlRepresentation htmlRep = new
        RestMarHtmlRepresentation();
        htmlRep.setHTML( table,
        "//weather/weather2_icon.marhtml");
        response.setEntity(htmlRep);
  }

  public void doPut(RestRequest request, RestResponse
  response) {
        JSONObject
        city=((RestJsonRepresentation)request.getEntity()).ge
        tJson();
        String newCity="";
        try {
              newCity=city.getString("newcity");
        } catch (JSONException e) {
              e.printStackTrace();
        }
        response.setStatus(RestStatus.SUCCESS_OK);
```

```
                                ((GoogleWeatherWidget)
                        this.getWidgetRef()).setCityName(newCity);
                        if(this.getWidgetRef().hasIcon()){
                              this.getWidgetRef().getIcon().render();
                              this.getWidgetRef().getIcon().repaint();
                        }
                  }
            }
      }
```

10 Bibliography & Sources

[1] W3C *Widgets 1.0*, *The widget landscape*, 14 April 2008
URL: http://www.w3.org/TR/widgets-land/

[2] Braiker B., *The Year of the Widget?* Newsweek, 22 December 2006.
URL: http://www.newsweek.com/id/44320

[3] Jaokar A., *Mobile web 2.0: on mobile widgets, micro learning and intertwingularity*.
URL: http://www.slideshare.net/MicrolearningOrg/mobile-web-20-mobile-widgets-microlearning-and-intertwingularity/

[4] Rohs M.. *Visual code widgets for marker based interaction*. Distributed Systems Group, Eidgenossische Tech. Hochschule, Zurich, Switzerland, 2005.
ISBN: 0-7695-2328-5
URL: http://www.vs.inf.ethz.ch/res/papers/rohs-widgets.pdf

[5] Widgets 1.0: Packaging and Configuration, W3C Working Draft 14 April 2008.
URL: http://www.w3.org/TR/widgets/

[6] W3C, *Description and the role of xForms*
URL: http://www.w3.org/MarkUp/Forms/

[7] Taggert A., director of product marketing for Yahoo! Mobile.
URL: http://www.softwaredeveloper.com/features/yahoo-mobile-developer-platform-022208/

[8] Soriano, J.; Lizcano, D.; Hierro, J.; Reyes, M.; Schroth, C.; Janner, T.;*Enhancing User Service Interaction Through a Global User Centric Approach to SOA*. IEEE Conference Proceeding, 16 March 2008 Pages:194 - 203 Digital Object Identifier 10.1109/ICNS.2008.37

[9] Rogovski R., *Findings from Forrester usability study*: Web users like rich Internet applications because they are easy to use. How does the usability of RIA and HTML applications compare? 7 December 2006.
URL: http://www.forrester.com/Research/Document/Excerpt/0,7211,40566,00.html

[10] Urbieta M., Rossi G., Ginzburg J. *Designing the Interface of Rich Internet* Applications, 31 October 2007
URL: http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/4383145/4383146/04383169.pdf

[11] Preciado J.C, Linaje M., Comai S, Sanchez-Figueroa f.
Designing Rich Internet Applications With Web Engineering Methodologies.
October 5 2007 ISBN: 978-1-4244-1450-5
URL: http://www.webml.org/webml/upload/ent5/1/Comai_Preciado_WSE07.pdf

[12] Preciado J.C. *Necessity of methodologies to model RIA*
Proceedings of the Seventh IEEE International Symposium on Web Site Evolution
2005 Pages: 7 - 13
ISBN: 0-7695-2470-2
URL: http://portal.acm.org/citation.cfm?id=1092361.1092692

[13] Farrell, J.; Nezlek, G.S. *RIA Next stage of application development*
29thInternationalConferenceonInformationTechnologyInterfaces
IEEE25June2007Pages:413-418
Digital Object Identifier 10.1109/ITI.2007.4283806
URL:http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/4283719/4283720/04283806.pdf

[14] Fielding R. T., *Architectural Styles and the Design of Network-based Software Architectures*, University of Irvine, California 2000
URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[15] Richardson L., Ruby S., *RESTful Web Services*, O´Reilly 2007
ISBN 10: 0-596-52926-0

[16] Institute for Software Research, University of California, Irvine. *The C2 Style Rules*
URL: http://www.isr.uci.edu/architecture/c2StyleRules.html

[17] Nandini R., Hardy V., *Graphical, Scripted and Animated User Interfaces on the Java Platform,* Sun Microsystems 2007
URL: http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-5743.pdf

[18] Paal S., Bröcker L., Borowski M., *Supporting On Demand Collaboration in Web-Based Communities*, Fraunhofer Institute for Media Communication.
URL: http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/11152/35810/01698353.pdf

[19] Lawton G., These *Are Not Your Father´s Widgets*, Computer, Volume 40 Issue 77 July 2007 Pages 10-13
URL: http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/2/4287226/04287234.pdf

[20] Sharad A., *Architecture and Context*, 13 August 2008
URL: http://www.calpoly.edu/~arch/program/fifthyr/atre.pdf

[21] *MIDP 3.0 Public Draft Version*
URL: http://jcp.org/aboutJava/communityprocess/pr/jsr271/index.html

[22] Voulgaris G., Constantinou A., Benlamlih F., *Activating the Idle Screen: Uncharted Territory*, Informa Telecoms & Media 2007
URL: http://www.visionmobile.com/researchreports.php

[23] Jaokar A.. *On device portals - ODP, Widgets and the Phonetop*, 7 october 2007
URL: http://opengardensblog.futuretext.com/archives/2007/10/on_device_porta_1.html

[24] *Mobile AJAX For Java ME*
URL: https://meapplicationdevelopers.dev.java.net/mobileajax.html

[25] O`Reilly T., *What is Web 2.0?,* 4 April 2008
URL: http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html?page=1

[26] Ben R., *Designing RESTful Web Applications*, International PHP conference lecture, 2007.
URL: http://benramsey.com/media/talks/ipcse07-rest.pdf

[27] Sameer T., *RESTful Web Services*, 1 August 2006
URL: http://java.sun.com/developer/technicalArticles/WebServices/restful/index.html

[28] Louvel J., New JSR to define a high level REST API for Java, 14 February 2007
URL: http://blog.noelios.com/2007/02/14/

[29] Louvel J. *Restlet API and JSR-31*, 25 February 2007
URL: http://blog.noelios.com/2007/04/25/restlet-api-and-jsr-311-api/

[30] *JAX-RS: Java™ API for RESTful Web Services Editors Draft*, 20 March 2008
URL: https://jsr311.dev.java.net/drafts/spec20080320.pdf

[31] *The Restlet 1.0 tutorial*
URL: http://www.restlet.org/documentation/1.0/tutorial

[32] ISR-Connector, *CREST: When REST Just isn´t Enough*, Institute for software research, University of California USA 2007
URL: http://www.isr.uci.edu/newsletters/ISR-Connector-FW2007.pdf

[33] Erenkrantz J., Gorlick M., Suryanarayana G., *From Representations to Computations: The Evolution of Web Architectures,* Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2007
ISBN:978-1-59593-811-4
URL: http://portal.acm.org/citation.cfm?doid=1287624.1287660

[34] Gamma E., Helm R., Johnson R, Vlissides J., *Design Patterns: Elements of Resuable Object-Oriented Software,* Addison Wesley, 21st printing 2000. ISBN 0-201-63361-2

[35] Lathem J., Gomadam K., Sheth A., *SA-REST and (S)mashups: Adding Semantics to RESTful Service,* IEEE 2007. ISBN: 0-7695-2997-6
URL: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4338383

[36] Wikman J., Dosa F., *Providing HTTP access to Web Servers Running on Mobile Devices*, Nokia Research Center 24 May 2006
URL: http://research.nokia.com/files/NRC-TR-2006-005.pdf

[37] *Google Map Mashup Examples*
URL: http://code.google.com/apis/maps/documentation/examples/

[38] *Yahoo Pipes!*
URL: http://pipes.yahoo.com/pipes/

[39] W3C*, Scalable Vector Graphics (SVG) Tiny 1.2 Specification,* 10 August 2006
URL: http://www.w3.org/TR/SVGMobile12/

[40] Sledd A., Hambreaus T., *SVG Tiny 1.2 brings AJAX to the world of mobile SVG*, IKIVO AB, 2007
URL: http://svgopen.org/2007/papers/ajax/index.html

[41] Sony Ericsson, *Project Capuchin*, 30 April 2008,
URL:https://developer.sonyericsson.com/site/global/newsandevents/latestnews/newsapr08/p_project_capuchin_announcement.jsp

[42] Ramani N., JSR *290 and its relation to mobile AJAX*
URL:http://www.openajax.org/member/wiki/images/3/3b/JSR290_NandiniRamani_20061006.pdf

[43] *Java Language & XML User Interface Markup Integration*
URL: http://jcp.org/en/jsr/detail?id=290

[44] Restlet, *a lightweight REST framework for Java*
URL: http://www.restlet.org/

[45] *JAX-RS: Java API for RESTful Web Services*
URL: https://jsr311.dev.java.net/