

# Seminar Paper

## Design and Implementation of an Internet based Calendar System

December 2000

Thomas Jungmann, Reinhold Klapsing



University of Essen  
Information systems and software engineering  
Univ. Prof. Dr. Rony G. Flatscher

---

# Table of contents

## 1. Introduction

- 1.1 The subject and scope of this work
- 1.2 Existing standards regarding this work
- 1.3 Definition of terms

## 2. The requirements analysis

- 2.1 Overview
- 2.2 Phase 1 – Users view
- 2.3 Phase 2 – Software engineers view

## 3. The system architecture

- 3.1 Overview
- 3.2 Classes diagram
- 3.3 The basic operation principle

## 4. The database design

- 4.1 The entity relationship model
- 4.2 Transformation to relations
- 4.3 Normalisation

## 5. Usage of the system

## 6. Conclusions and future work

- 6.1 Interoperation
- 6.2 Conceptual improvements
- 6.3 Object orientation
- 6.4 Security

## 7. References

---

# 1. Introduction

## 1.1 The subject and scope of this work

The purpose of this work is to design and implement an internet based, distributed calendaring and scheduling system. The user can perform all operations on the calendar with a simple Internet web browser, while the actual calendar service and the database are maintained on a central server.

This system does neither support 'groupware'-like functionalities, e.g. free/busy-time planning between users, nor interoperation with other calendar systems yet.

The implementation of the services and interfaces to the web server and the database will be done in the scripting language 'Object Rexx' (IBM) [8].

## 1.2 Existing internet standard regarding this work

The 'Internet Engineering Task Force', IETF [2], has set up a working group for 'Calendaring and Scheduling' (calsch) [7], in order to provide a framework of standards when implementing such systems. They have already released the following RFCs (Request for Comments) and 'Internet drafts':

RFC 2445: Internet Calendaring and Scheduling Core Object Specifications (iCalendar)  
Specifies the objects and data types and defines also a special MIME-Type for Calendar Objects (text/calendar).

RFC 2446: iCalendar Transport-Independent Interoperability Protocol (iTIP)  
Specifies how calendar systems use iCalendar Objects to interoperate with other calendar systems.

RFC 2447: iCalendar Message-Based Interoperability Protocol (iMIP)  
Describes the binding of iTIP to email-based transport over the internet.

RFC 2739: Calendar Attributes for vCard and LDAP  
Describes how calendar systems exchange the addresses of the attendees of an event.

Internet draft: Implementors' Guide to Internet Calendaring  
This document describes the relationship between the above mentioned various internet calendaring and scheduling protocols.

Internet draft: Calendar Access Protocol (CAP)  
This describes an internet protocol for communication between Calendar User Agents and Calendar Stores.

It was mentioned before that the interoperation between calendar systems is beyond the scope of this work, therefore there is no need to follow the proposal of these Internet Standards until an interface to other systems is desired. Nevertheless the definition of terms supplied in the Internet Draft 'Calendar Access Protocol (CAP)' is still useful for the context of this work, and will partly be used (see: Definition of terms).

Other standards used:

ANSI-SQL 92:

The database is queried according to this standard, so it can easily be exchanged for any other database that meets ANSI-SQL 92.

HTTP 1.1 [4]

The web browser and the web server communicate via Hypertext Transfer Protocol (HTTP)

HTML 4.01 [6]

All output of the calendar will be formatted in HTML, which is then rendered by the web browser to display the information to the user.

CGI 1.1 [5]

Used for passing user data to the calendar.

### **1.3 Definition of terms**

Calendar:

A collection of calendar events associated with a specific user.

Calendar Event:

An entry in a calendar that represents an event for a specific user.

Calendar User (taken from CAP [7]):

An entity (often biological) that uses a calendaring system.

Calendar User Agent (from CAP [7]):

The client application that a Calendar User utilizes to access and manipulate a calendar.

Calendar Service:

The collection of programs that receive and interpret the Calendar Users commands and also generate and format the output for the user.

Calendar Store:

The database that stores the calendars.

## 2. The requirements analysis

### 2.1 Overview:

The requirements analysis will be performed in two steps, based on usability engineering methods, to ensure that technical considerations do not dominate the actual needs of the user right from the start. Of course, the limits of technical possibilities have to be kept in mind, so there is most likely always a trade-off between both perspectives:

1. Analysis of the users needs and their expression in ‘plain English’, without the use of technical terms and without already having a technical solution in mind. In this phase, the problem is only described from the users point of view.
2. Refinement of the needs and search for appropriate technical solutions that match each of the users needs, this time from the software designers point of view. In this phase, we are aware of all the technical possibilities that are at our disposal.

### 2.2 Phase 1 – Users view:

- The user wants to uses the system like a real calendar, where he can enter, modify, delete and query events. The access to the system shall be possible from any computer that is connected to the internet (e.g. from internet cafés), without the use of any special, proprietary software.
- The system shall be able to store separate calendars for an (almost) arbitrary number of users.
- As personal calendar data is from its nature rather private, there has to be a certain mechanism that ensures at least a minimum degree of confidentiality. So each user has to authenticate himself with a combination of an unique name and a password.

### 2.3 Phase 2 – Software engineers view:

Based on these user requirements and their implications, we find that the following needs and their solutions have to be met:

- As the client software (calendar user agent) must not be any kind of proprietary software, we are going to use a simple standard web-browser (e.g. Netscape Navigator, Opera, Internet Explorer) that uses the standard ‘Hypertext Transfer Protocol’ HTTP to transmit the servers HTML output and receive the users input via the ‘Common Gateway Interface’ CGI. This is a very well established technique and is available on most platforms.

- The web server has to be capable of CGI as well. For portability reasons it is recommended to use a web server that is available on many platforms, like the Open-Source-Webserver 'Apache' [3].
- The system is going to be a multi-user system, so it is possible that the amount of data that is to be stored, is rather large. Due to the fact that also some meta-information about the actual data is needed, e.g. for administrative purposes, it is recommended to use a powerful and reliable database system (calendar store) that can be queried by the standard-database language 'SQL', like 'MySQL' [10], which is also available for many platforms.
- For the actual implementation of the calendar functions (calendar service), a programming language is needed. For CGI-Programming a scripting language like Object Rexx [8] with powerful string-parsing functions is generally preferable. An interface to the database is also required. Rexx/SQL [9], is a widely database independent interface for the Rexx language. Both are available on many platforms as well.
- There is a need for user authentication and as HTTP is a stateless protocol, it is also required to implement some kind of 'artificial' session management, that is capable of keeping track what user wants to access which data, after he has logged on. A session management can be introduced in three different ways: Hidden Fields in HTML-Forms [6], CGI PATH\_INFO mechanism [5] or by the use of 'cookies' [11]. As this system shall also be implemented in a WML Version, the first alternative is preferable.

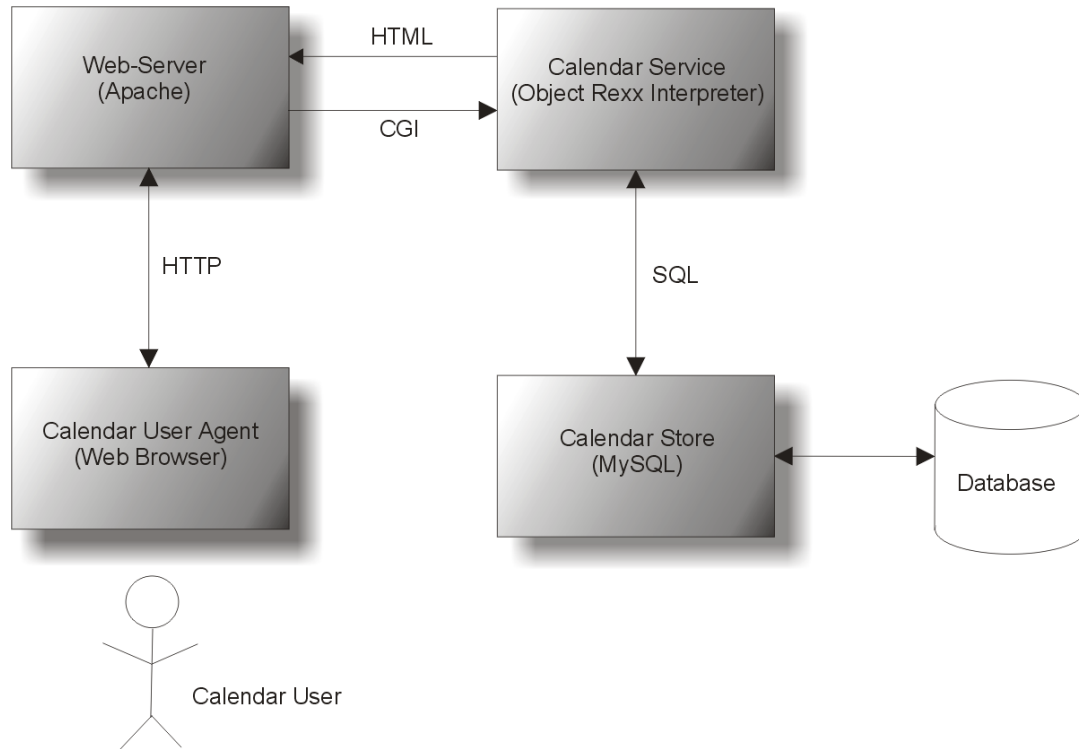
The information about a session is stored in the database together with a certain expire time (e.g. 20 minutes) after the last user action. This minimizes the risk of abuse of a session identifier, that has accidentally become known to a third party.

### Summary of needs

User needs (Users view)	Technical requirements (Software engineers view)
Internet calendar, easy access from everywhere	Client = Webbrowser, HTML+CGI for data exchange
Multiple users	Powerful database, users must logon to use the calendar. Necessity of session management
Confidentiality of data	Password check and session management with timeout.

## 3. The system architecture

### 3.1 Overview



System Architecture Overview

This graphic shows the interoperation between the different entities.

The calendar user uses his Internet Web Browser as 'Calendar User Agent' for all operations on the system. The communication between the browser and the web server is performed over the HTTP-Protocol. The web server receives the CGI-Requests and stores them in environment variables. These can be read by the Calendar Service Rexx-Script, that is automatically executed by the web server on receiving a CGI-Request.

The Rexx-Program interprets the users command and sends a Standard-SQL Query (via the Rexx/SQL Interface) to the database, if necessary. Possible query results will be passed back to the Rexx script.

These results will be interpreted and formatted as HTML-Output, that is then send back via the Standard Output to the web server. When the Rexx-Script is finished and its process terminates, the web server generates the necessary HTTP-Header information and sends the complete HTML-Page back to the Web Browser.

Once the user has logged on to the system, all subsequent generated HTML-Pages contain a reference to a session, a session identifier, that is always passed back to the calendar service via CGI, so it is possible to distinguish between different users working simultaneously on the system. The session is valid until the user ends his session explicitly, or a certain time out limit after the last user action is reached.

### 3.2 The classes diagram

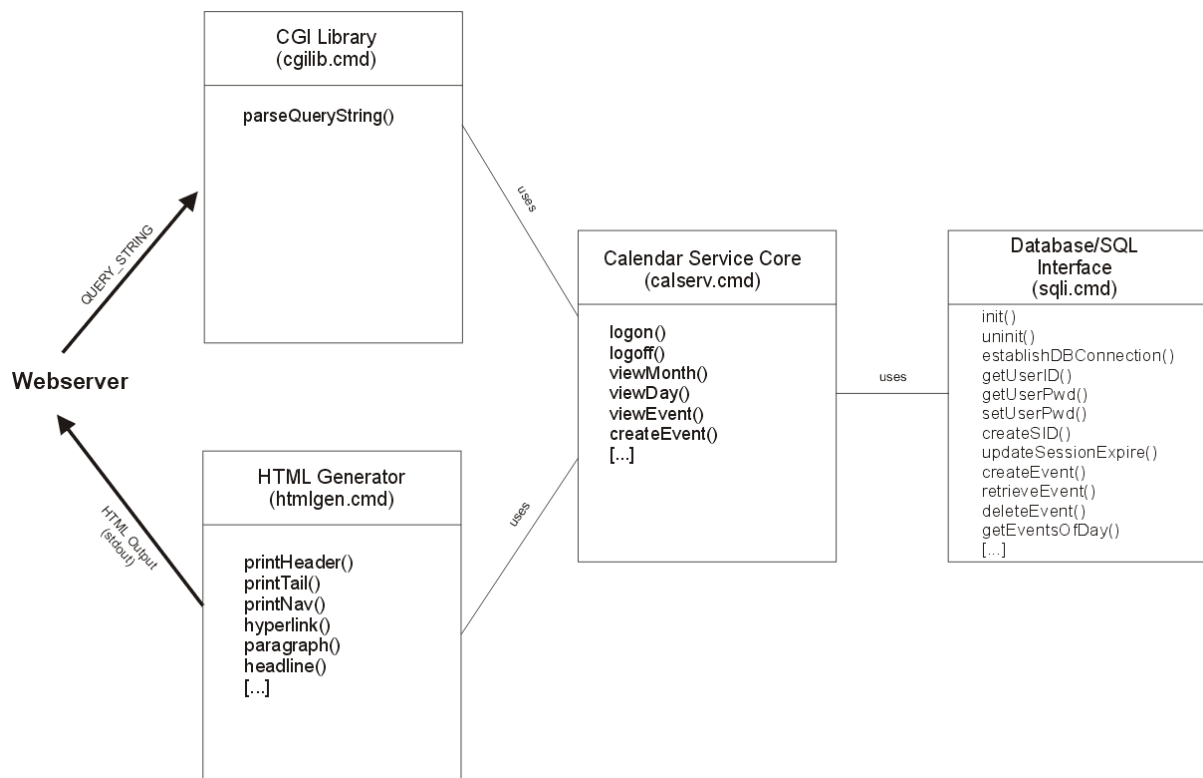


Diagram of classes (Calendar Service)

### 3.3 The basic operation principle

The Calendar Service is a system of several Rexx Scripts. The main part of the system can be found in the Calendar Service Core (calserv.cmd). This script is always started by the web server on receiving a CGI query. Although nearly all of the actual ‘work’ is performed by this script, it does not communicate itself with the other components. This is done by the supporting interface classes.

Right after starting the main script, it creates instances of the ‘CGIParse’-Class, the ‘HTMLGen’-Class and the ‘SQLInterface’-Class. These initialise themselves through their constructors, if necessary, and provide methods for the main script for interaction with the other components, like the database and the web server. This is the initialisation procedure of



these variables. There is no need to expose the parser handle-variable, as this interface is only once per runtime used to put the CGI Variables into the stem variable 'cgi':

```
initVars:  PROCEDURE EXPOSE db html cgi.
           db      = .sqlInterface~NEW
           html    = .HTMLgen~NEW
           parser  = .cgiParse~NEW
           cgi.    = parser~parseQueryString()
           return
```

The third line `db=.sqlInterface~NEW` creates an object instance of the `SQLInterface-Class`. The constructor of that class automatically loads the required library `Rexx/SQL` and starts the initialising process by sending the message `establishDBConnection()` to itself:

```
::METHOD INIT          /* Constructor */
  if rxFuncQuery("SQLLoadFuncs") then do
    call rxFuncAdd "SQLLoadFuncs", "rexssql", "SQLLoadFuncs"
    call SQLLoadFuncs
  end
  self~establishDBConnection()
```

By encapsulating the access to other components in separate interface classes, the advantage gained is, that it becomes on the one hand easier to adapt the main program to changing environments, because you have only to exchange the interfaces. On the other hand you can possibly reuse the interface code for other, similar applications.

The **CGIParse-Class**, for instance, has only one public method, called 'CGIParse()', that reads the CGI-QueryString from the environment, separates the variable pairs, and assigns them to a Rexx-Stem Variable, an array, which is returned to the main script, where they can be used conveniently.

The **SQLInterface-Class** is a collection of methods that are used for the interaction with the database. Most of the public methods prepare a text-string containing a standard ANSI-SQL-92 SQL Command/Query and send it to the MySQL-Server (by using itself functions of a shared library (written in C) called `Rexx/SQL`). The reply from the database, probably containing calendar data, is then returned to the Calendar Service Core Script.

This interface is also used for the WML-Version of an internet calendar for cellular phones, currently being developed by Ednan Masovic [12], so this is a good example for the easy reuse of code. Here is an example of a typical function (method), this one returns the user name for a specific user ID:

```
::METHOD getUsername
-----
/* Retrieves the UserName (Nickname) for a given UserID
   Arguments:  uID      = UserID
   Returns:    -1       on Error
               userName when uID exists          */
-----
parse arg uID
cmd = "select uniqueName from user where uID='||uID||'";
res = SQLCommand(SQLout,cmd)
if SQLout.uniqueName.0 then /* SQLout.uniqueName.0 = Resultcount of SQLCommand */
  return SQLout.uniqueName.1
else
  return -1 /* SQLCommand returned empty list => userID not found! */
```

The **HTML Generator-Class** contains the necessary methods to write the output of the calendar service in HTML to the standard output. It also generates the required headers and simplifies the use of HTML-Forms, Hyperlinks and other formatting structures. All output is passed through this interface, no other component of the system writes to the standard output. This is an example excerpt from the source code, a method that writes a HTML-Header to the standard output:

```
::METHOD printHeader
  parse arg title
  sq = d2c(39)
  say "content-type:text/html"
  say
  say "<HTML>"
  say "<HEAD><TITLE>" || title || "</TITLE></HEAD>"
  say "<BODY BGCOLOR=" || sq || "#D0D0D0" || sq || " TEXT=" || sq || "#006600" || sq || ">"
```

After the **Calendar Service Core Script** is started and initialised, it looks for a CGI-Variable 'action=xy', where 'xy' is the users command, e.g. 'logon' or 'overview'. It then calls the appropriate procedure for this command. These procedures get their necessary arguments from the other CGI-Variables, that have been stored by the CGIParse-Method in an array-variable.

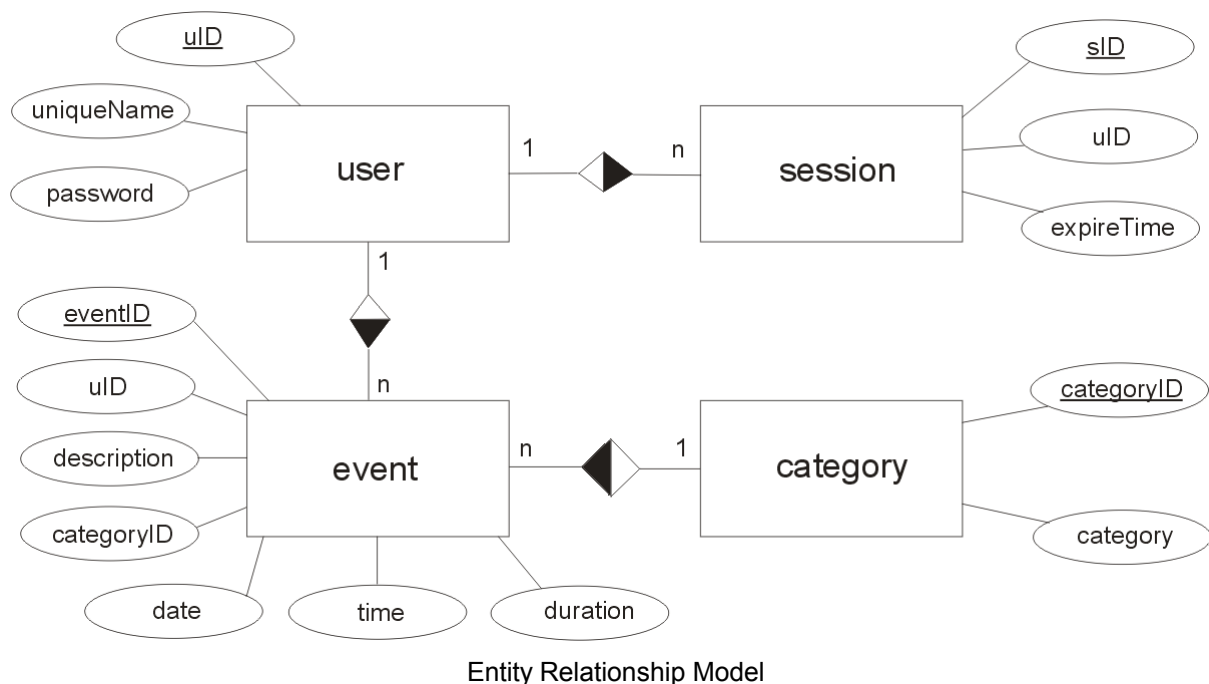
This is the complete main-procedure of this script:

```
call initVars
html~printHeader('The Web Calendar')
select
  when cgi.action = 'logon'          then call logon          --check username and password
                                     --and create session
  when cgi.action = 'newuser'       then call newuser         --create a new user account
  when cgi.action = 'logout'        then call logout          --invalidate session
  when cgi.action = 'overview'      then call overview         --display overview of this month
  when cgi.action = 'goto'          then call gotoMonth        --navigate to specific month
  when cgi.action = 'gotoform'      then call gotoForm         --show HTML-Form for gotoMonth
  when cgi.action = 'viewday'       then call viewDay          --display all events of day
  when cgi.action = 'view'          then call viewEvent        --display event details
  when cgi.action = 'create'        then call createEvent      --add event to database
  when cgi.action = 'newentry'      then call newentryForm     --show HTML-Form for createEvent
  when cgi.action = 'Edit'          then call editEventForm    --show HTML-Form for updateEvent
  when cgi.action = 'update'        then call updateEvent      --accept modifications for event
  when cgi.action = 'Delete'        then call deleteEvent      --delete event permanently
  otherwise call abort 'Unknown CGI-Action'
end
html~printTail
DROP db          --drop references to interfaces
DROP html
DROP parser
exit 0
```

## 4. The Database Design

### 4.1 The Entity Relationship Model

The design of the entity relationship model is based on the considerations of the requirements analysis. The first step is to transform the participating 'real life'-entities to abstract data constructs/entities. These data-entities have descriptive attributes that distinguish them from each other. They also interoperate in between each other in a certain way, that has to be well described in order to find an appropriate model that does not contradict the considerations of the requirements analysis.



### 4.2 The transformation to relations

Two entities can be found very easily, as they are the central objects in the multi-user calendar system: The 'User' and his 'Events' that are about to be stored.

A 'User' has the following minimum attributes:

- Name: a system wide unique name (nickname), that distinguishes him from the other users and that is therefore used during the logon procedure
- Password: a secret password, that allows only authorized access to ensure confidentiality

- 
- User identifier: this is not absolutely necessary, as the username is already unique and would therefore be enough to distinguish the users, but it is added to simplify administrative operations.

An 'Event' should have at least these attributes:

- Event identifier: used for administrative purposes as a unique identifier of an event
- User identifier: a reference to the owner of the event
- Description: this is the actual data of the event
- Time of event: the date and time of the beginning of the event
- Duration: duration of the event
- Category: events can belong to categories that can be used for searching specific events (e.g. birthdays, meetings etc.)

It was mentioned before that there is a need for a session management, i.e. we have to store somewhere what users are currently using the system, and how can they be distinguished. So we create an extra entity, that has no obvious 'real-life' association. This entity is going to be called 'Session' and has this attributes:

- Session identifier: a unique identifier for a session
- User identifier: a reference to the user who initiated the session
- Expire time: a session is only valid for a defined period of time after the user did his last action on the system. This guarantees a certain security improvement.

The last entity 'Category' seems to be already part of the 'Event'-Entity as an attribute, but since this is going to be used as a search parameter, it is better to use a set of predefined categories that are common for all users. The 'Event'-Entity has to be modified in that way, that only a (numerical) reference (categoryID) to this table of categories is stored:

- Category identifier: unique identifier for a category
- Category name: a plain text name for the category, e.g. 'Birthday', 'Meeting', 'Call' etc.

The next step is to analyse the relationship between these entities. Three binary relations (binary relations=relations between two entities) describe the interdependencies between the four entities:

- Entities 'Event' and 'Category': An event does always belong to exactly one category, while many events can belong to the same category. So the connectivity of these entities is 'one to many'.
- Entities 'User' and 'Event': Each event belongs to exactly one user, but one user can have many events, so this is also an 'one to many'-relation.
- Entities 'User' and 'Session': This is not as obvious as for the other relations. Each session belong to a certain user, but should it be possible that one user has more than one active session at a moment? Considering the possibility that one session has not ended properly with a clean 'logout', which is possible if, for instance, the web

browser crashes, it should be possible for the user to start a new session without waiting for the old session to timeout. So this is a ‘one to many’-relation as well.

### 4.3 The normalized relation scheme

A formal examination of the relations for functional dependencies between attributes that do not belong to the primary key is needed to determine the level of normalisation that applies to this database, but since this is not within the primary scope of this work and since it is a rather simple database, it will be skipped. A functional dependency between the category and the category identifier in the ‘event’-entity has been anticipated and already avoided by creating a separate ‘category’-entity.

These are the table definitions actually used in the implementation:

```
mysql> describe user;
```

Field	Type	Null	Key	Default	Extra
uID	int(11)		PRI	0	
uniqueName	char(25)		UNI		
password	char(25)	YES		NULL	

The ‘user’ table uses the user ID ‘uID’ as the primary key, the ‘uniqueName’ is equivalent to the user ID and is only used as an alias of the user ID, so people do not have to remember a number but only a nickname.

```
mysql> describe event;
```

Field	Type	Null	Key	Default	Extra
eventID	int(11)		PRI	0	auto_increment
uID	int(11)				
categoryID	int(11)			NULL	
description	char(100)	YES		NULL	
eventDate	date			0000-00-00	
eventTime	time	YES		NULL	
duration	time	YES		NULL	

The user ID ‘uID’ must not be ‘Null’ because it is a foreign key. Any event must belong to a user.

Category ID is also a foreign key, so it must be set to ‘not\_null’ as well.

The data of an event must not be ‘Null’ because this would not make any sense at all. It is the nature of an event to happen sometime.

---

```
mysql> describe category;
```

Field	Type	Null	Key	Default	Extra
categoryID	int(11)		PRI	0	auto_increment
category	char(26)	YES		NULL	

The primary key is the category ID. The category name (category) may be 'null' to allow an empty default category.

```
mysql> describe session;
```

Field	Type	Null	Key	Default	Extra
sID	char(25)		PRI		
uID	char(25)				
expireTime	datetime			0000-00-00 00:00:00	

The 'session' table uses the session ID as primary key. The foreign key user ID is 'not\_null'. The expire time is 'not\_null' as well, because any session must end sometime.

## 5. Usage of the system

To use the system, the user has to navigate to the Logon-Page where he will be prompted for his username and password, in order to authenticate himself to the system. The 'Logon'- and the 'Newuser'-Page are in fact the only static HTML-Pages, because a session ID has not been generated yet, and will be needed only for the subsequent pages.



Logon Page

Once the user is successfully authenticated he will be shown the overview for the current month. Days that contain events will be shown with hyperlinks, that link to a page that shows an overview of all events of this day.

For navigation to other month the user can either follow the '<' and '>' links next to the calendars' headline, or directly enter the desired date into the from just beneath the calendar.



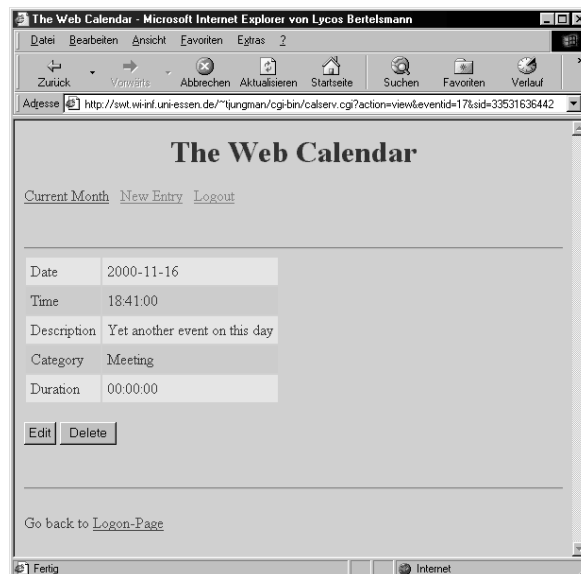
Month Overview

A hyperlink in the navigation bar displays the form for creating new events. Most of the values can be entered conveniently by the use of dropdown boxes.



New Entry Form

When an event is displayed, the user can easily modify or delete it, by using the corresponding button.



Detail View for Event

When the user has done his work, he should click on the 'Logout'-Hyperlink, which immediately invalidates the current session. No action on the calendar can be performed until the next logon.



## 6. Conclusions and future work

### 6.1 Interoperation

The calendar system as it is, is a standalone system. Although it can be called ‘distributed’, because it can be used from everywhere with a simple web browser and the database can easily be separated from the web server machine and placed virtually everywhere, and although it is a multi-user system, as users can use it simultaneously, it misses features to interoperate with other calendar systems and even to allow e.g. free/busy scheduling between its own users.

It was mentioned in the introduction that the Internet Engineers Task Force IETF has already released standards, that define in a very detailed way data types and protocols for the exchange of calendaring information. It will certainly be a challenging task to implement all suggested functions, but there is surely a need for this as there are often proprietary solutions to be found.

### 6.2 Conceptual improvements

The concept of separating interfaces from the calendar core functions and its advantages have already been stated before. If this idea had been preserved throughout the whole implementation, it should be easy to exchange the HTML-Interface e.g. for a WML-Interface, but the interfaces have not been designed clearly enough. The calendar service core script contains a lot of HTML specific code, and it is difficult to divide these functions now.

### 6.3 Object orientation

Although the program code was written in Object Rexx, there are little concepts of the object oriented paradigm used. Many parts are based on procedural thinking but it would also be possible to think of a completely different concept. Events could, for instance, be objects that have their data attributes, but also methods for creating, altering and deleting themselves. User object could contain methods for verifying their passwords and so on. But these fundamental conceptual alterations would require an almost complete rewrite of the code.

### 6.4 Security

The security of this system is based on user passwords and session management with timeout. However all data is transmitted completely unencrypted in plain format over the internet. A third party, being on some network node in between the calendar user agent and the web server, could easily read the transmitted data.

All data is transmitted via the CGI-‘GET’-Method, which makes it even easier to listen to the communication, because log files of caches store the information as well. A first step would of course be to use the CGI-‘POST’-Method (which would also allow to transmit more than 1kbyte per request), but much more security would be gained, if the whole communication between user agent and web server was encrypted, for instance by using the ‘Secure Socket Layer’ Protocol SSL.

---

# References

- [1] 'W3C – Technical Reports and Publications'. <http://www.w3.org/TR>
- [2] 'Internet Engineers Task Force IETF'. <http://www.ietf.org>
- [3] 'Apache HTTP Server Project'. <http://httpd.apache.org>
- [4] 'HTTP 1.1', RFC 2616, June 1999, T. Berners-Lee, P. Leach, L. Masinter, H. Frystyk, J. Mogul, J. Gettys. <http://www.w3.org/Protocols/rfc2616/rfc2616.html>
- [5] The WWW Common Gateway Interface Version 1.1", June 1999, Ken A L Coar, D.R.T. Robinson, INTERNET-DRAFT. <http://CGI-Spec.Golux.Com/draft-coar-cgi-v11-03.txt>
- [6] 'HTML 4.01 Specification', W3C Recommendation December 1999. <http://www.w3.org/TR/html401>
- [7] 'IETF Working group for calendaring and scheduling (calsched)', RFCs 2445, 2446, 2447, 2739, Internet drafts "Implementors' Guide to Internet Calendaring", Internet draft "Calendar Access Protocol (CAP)". <http://www.ietf.org/html.charters/calsch-charter.html>
- [8] 'IBM Object Rexx'. <http://www-4.ibm.com/software/ad/obj-rexx/>
- [9] 'Rexx/SQL', Mark Hessling, <http://www.lightlink.com/hessling/rexxsql/index.html>
- [10] 'MySQL', <http://www.mysql.com/>
- [11] 'HTTP state mechanism', RFC 2109, D. Kristol, L. Montulli, February 1997, <http://www.ietf.org/rfc/rfc2109.txt?number=2109>
- [12] 'Entwurf und Entwicklung eines Web-basierten Terminplaners (WML-Version)', Ednan Masovic, December 2000, <http://swt.wi-inf.uni-essen.de/~emasovic>