Uppsala Student Thesis in
Computer Science 311
2007-06-01
ISSN 1100-1836

A database interface for Java-based AJAX applications

Vasilij Savin

Department of Information Technology

Uppsala University

Box 337

S-751 05 Uppsala

Sweden

Supervisor: Kjell Osborn

Examiner: prof. Tore Risch

Abstract

Google Web Toolkit (GWT) is a Java-to-Javascript compiler, that allows to greatly simplify development of interactive web applications. Amos II (Active Mediator Object System) is an object-relational mediator database system that allows queries and views over different kinds of back-end data sources and also provides own main-memory database servers. The purpose of this work is to create a simple API between GWT-based web applications and Amos II servers. It enables developers to create web client applications using GWT that use Amos II as the back end database without necessity to create any server side code.

Table of Contents

1 Introduction

Recent years showed an explosive growth of activities over the Internet. Businesses are moving into this lucrative market, offering various services to users. Nowadays it is essential for a system to be not only functional, but also attractive to users: it has to be easy to use, responsive, and highly interactive.

This is where AJAX comes into play. It stands for Asynchronous Javascript And XML [8] The main feature of this technique is to allow creation of highly interactive web applications. However it comes at high development cost, because JavaScript development, testing, and debugging are extremely difficult and time-consuming processes.

In May 2006, Google introduced new tool – Google Web Toolkit (GWT) [1], that makes development less painful for AJAX developers. The developers can now write AJAX code in Java, making it easy to draw upon the huge pool of Java programmers. GWT provides tools to compile this Java code into Javascript for deployment. Since the code is initially developed in Java, all Java tools can be used that have been developed through the years for support of coding, testing, and debugging.

AMOS II (Active Mediator Object System) [2, 3], is a lightweight, object–relational database, running on the Windows NT platform. The purpose of the Amos II project is to develop and demonstrate a database mediator architecture for supporting information systems where applications and users combine and analyse data from many different data sources. A data source can be a conventional database but also text files, data exchange files, web pages, etc.

This project implements an interface, called GWT-AMOS, between GWT-based AJAX web applications and Amos II database servers. This simplifies development of interactive AJAX applications using Amos II.

2 Background

First this chapter briefly describes the mediator-wrapper approach, on which the Amos II system is based. Then the Amos II system is described followed by a description of the principles of AJAX and GWT.

2.1 The mediator-wrapper architecture

Today a lot of applications are developed that need to access databases. On-line travel agencies for example must be able to access databases of flight companies to gain information of empty seats, of airports to see arrival and departure timetables and of hotels to search for free rooms. Often such information won't be stored in one database, but each flight company, airport and hotel will have their own database. Thus access to several databases is necessary.

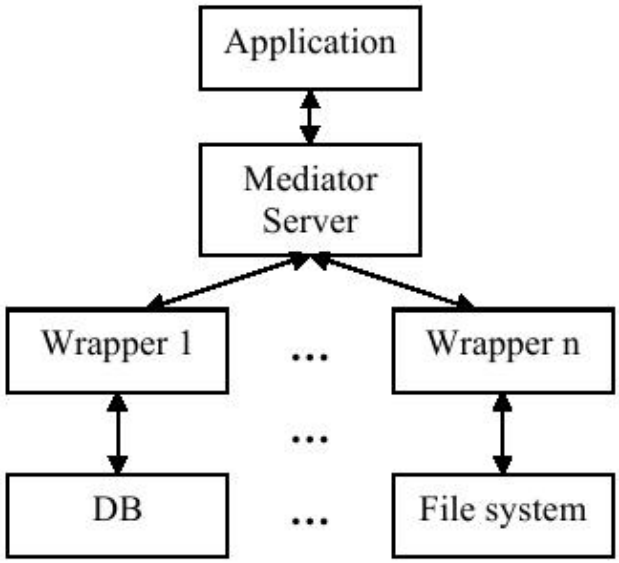

Figure 2.1: Mediator Wrapper Architecture

Furthermore there exists a large amount of possibilities to store data. One company may have one kind of database and schema, while another company might have a completely different way to store data. Thus an application needs the possibility to access different kinds of data sources.

The mediator/wrapper approach [18] supports queries and views over heterogeneous data sources. A mediator system consists of *mediators* each having one or several *wrappers*. A mediator is a software module that presents views of data in other systems in terms of a common data model (CDM). The task of a mediator is to process queries over these views. The queries are automatically

split depending on the data and capabilities of the target data sources. A mediator can regard other mediators as data sources, i.e. a federation of mediators can be created. The wrappers [3] define query processing interfaces to different kinds of data sources, for example relational databases, XML files or object stores. Queries, sent from the mediator, are translated by the wrapper to a data source specific format and thus hide the heterogeneity of that data source. After that the wrapper retrieves the query result, which has to be translated again into the common data model of the corresponding mediator. The translated data is passed to the mediator. There all results from all data sources are integrated and returned to the application.

2.2 AMOS II

Amos II [3] is a distributed mediator system with an object oriented and functional data model. Queries to that data model are written in AmosQL [2,3], a functional query language. The system can consist of several autonomous and distributed Amos II peers. These peers can interoperate through its distributed multi-database facilities. Each mediator peer offers three possibilities to access data:

- Access to data stored in an Amos II database.
- Access to wrapped data sources.
- Access to data that is reconciled from other mediator peers.

Especially the second point makes Amos II extensible. New application oriented data types or operators can easily be wrapped and queried by AmosQL. Thus a powerful query and data integration is offered by the system.

The core of Amos II is an open, light-weight and extensible main-memory database management system (DBMS).

2.2.1 Data model

The basic components of the data model of Amos II are *objects*, *types*, and *functions*. In object oriented programming languages these concepts approximately correspond to instances, classes, and methods.

Objects

Everything in Amos II is represented as objects, independent whether the object is user-defined or system-defined. Literal objects are primitive objects like integers, strings or even collections that

represent arrays of other objects. In addition to this there are *surrogate objects* which are created by the user or the system and have explicit object identifiers (OIDs). An object of type *Person* in Amos II can be created with the following command:

```
create person instances :thisisme;
```

When a query requests the object *:thisisme*, the returned result will be displayed in this format:

```
#[OID 1101]
```

Types

Types are used for classifying objects. Each object is an instance of a type. Types associates properties with objects. If an object inherits from another type it gains all properties from the supertype. The following command creates a person and an assistant type in Amos II:

```
create type Person;
```

```
create type Assistant under Person;
```

Amos II offers a basic type hierarchy that can be seen in figure 2.3. The root element is named *Object*. All (system and user) defined type names are represented as instances of a type named *Type*. User defined types are always a subtype of type *Userobject*

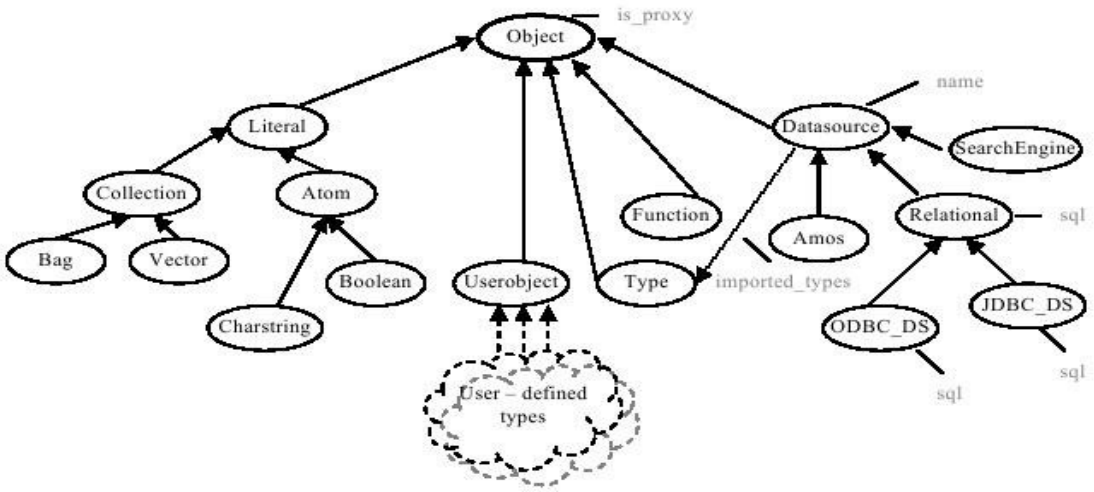

Figure 2.2 System type hierarchy

Functions

Functions model the relationship between objects, model properties of objects, queries and computations over objects. The function's *signature* consists of the types of its arguments and its

result. All functions are instances of a type named *Function*.

2.2.2 External AMOS II interfaces

There are two ways to interface Amos II with other programs, either an external program calls Amos II through the *callin* interface[4], or Amos II calls external functions through the *callout* interface[4]. Interfaces to several programming languages are developed, e.g. C, Java, and Lisp.

In this work only the *callin* interface to Java is used. It is similar to JDBC where embedded queries are provided through Java methods. Strings containing AmosQL statements are passed as method arguments to Amos II for dynamic evaluation. Methods are also provided to iteratively access the results of AmosQL statements. The embedded query interface is relatively slow since the query statements have to be parsed, optimized, and compiled at run time.

In the *fast-path interface* predefined Amos II functions are called as Java methods, without the overhead of dynamically parsing and optimizing query statements. The fast-path is significantly faster than the embedded query interface. It is therefore recommended to always make Amos II derived functions and stored procedures for the various Amos II operations performed by the application and then use the fast-path interface to invoke them directly

2.3 AJAX

AJAX stands for Asynchronous Javascript and XML [8]. It is web development technique for creating interactive web applications. The intent is to make websites feel more responsive by exchanging small amounts of data with the server asynchronously behind the scenes, so that the entire web page does not have to be reloaded each time the user requests a change.

The AJAX term was mentioned first time by Jesse James Garrett in February 2005 [5]. Garrett invented the term when he realized the need for a shorthand name to represent the suite of technologies he was proposing to a customer.

AJAX is not a technology per se, but a term that refers to the use of a group of technologies.

- XHTML [11] and CSS [10] are used for marking up and styling information.
- XMLHttpRequest objects [12] are used to exchange data asynchronously with the web server.
- XML, JSON (JavaScript Object Notation) [13] or plain text are used for transferring information between server and client.
- DOM (Document Object Model) [14] is used with the client-side scripting language, e.g. Javascript or JScript, to dynamically display and interact with the information presented.

Most user actions in the interface trigger an HTTP request back to a web server in classic web application model. The server processes the request — retrieves data, performs some calculations, talks to various backend systems — and then returns an HTML page to the client.

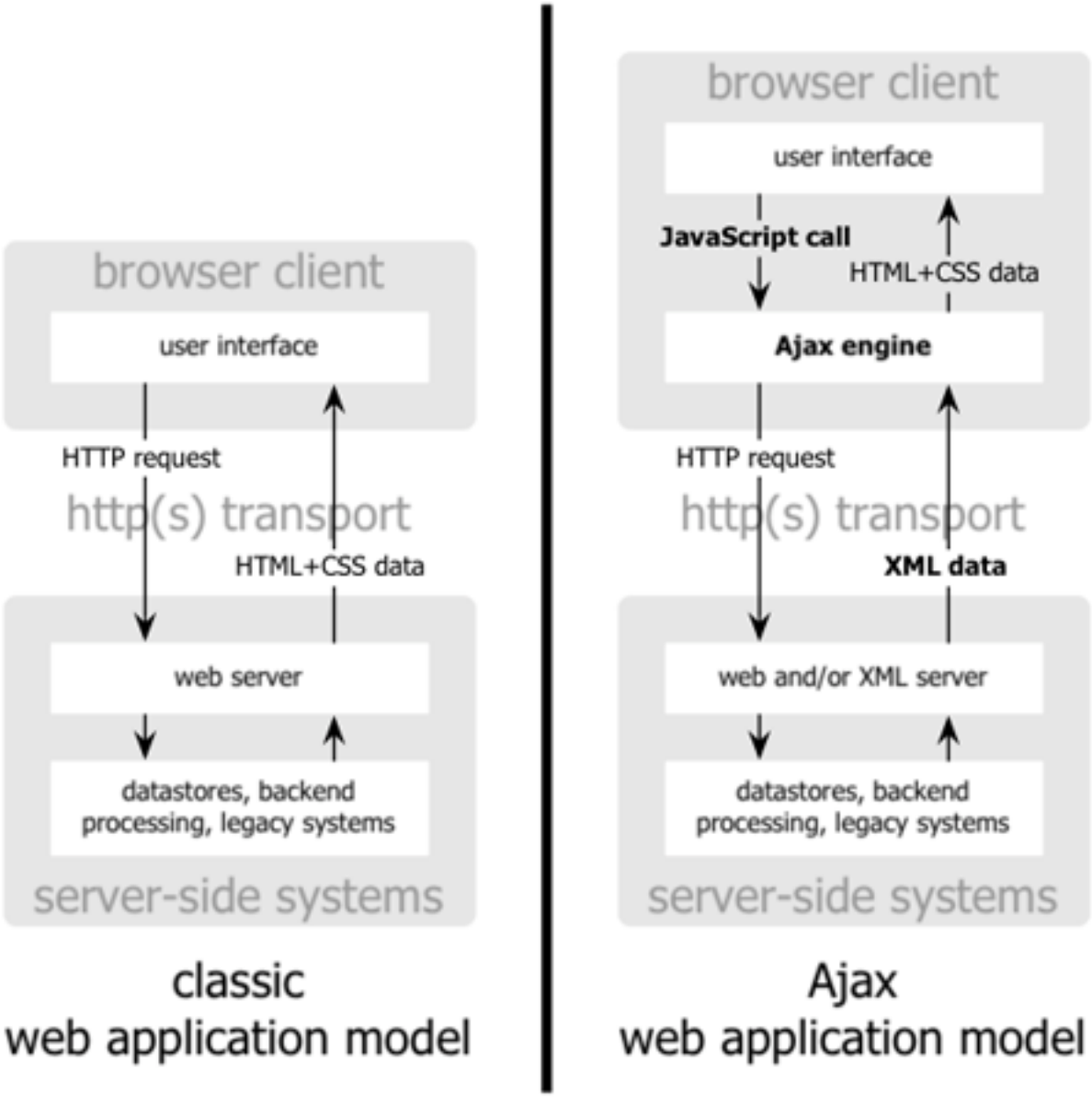

Figure 2.4 Differences between Classic and Ajax application models[5]

However, even though the classical model can be good for web pages, it is not necessarily suitable model for interactive applications. The main drawback of such an approach is that while the server is handling client requests, the user has no other option but to wait for the reply.

An AJAX application eliminates the start and stop nature of interactions on the Web by introducing a middleware — an AJAX engine — between the user and the server.

Instead of loading a web page at the start of the session, the browser loads an AJAX engine written in JavaScript. This engine provides user interface primitives and an asynchronous communication

protocol with the server. The asynchronous communication makes the user rarely having to stare at a blank browser window, waiting for the server to return results.

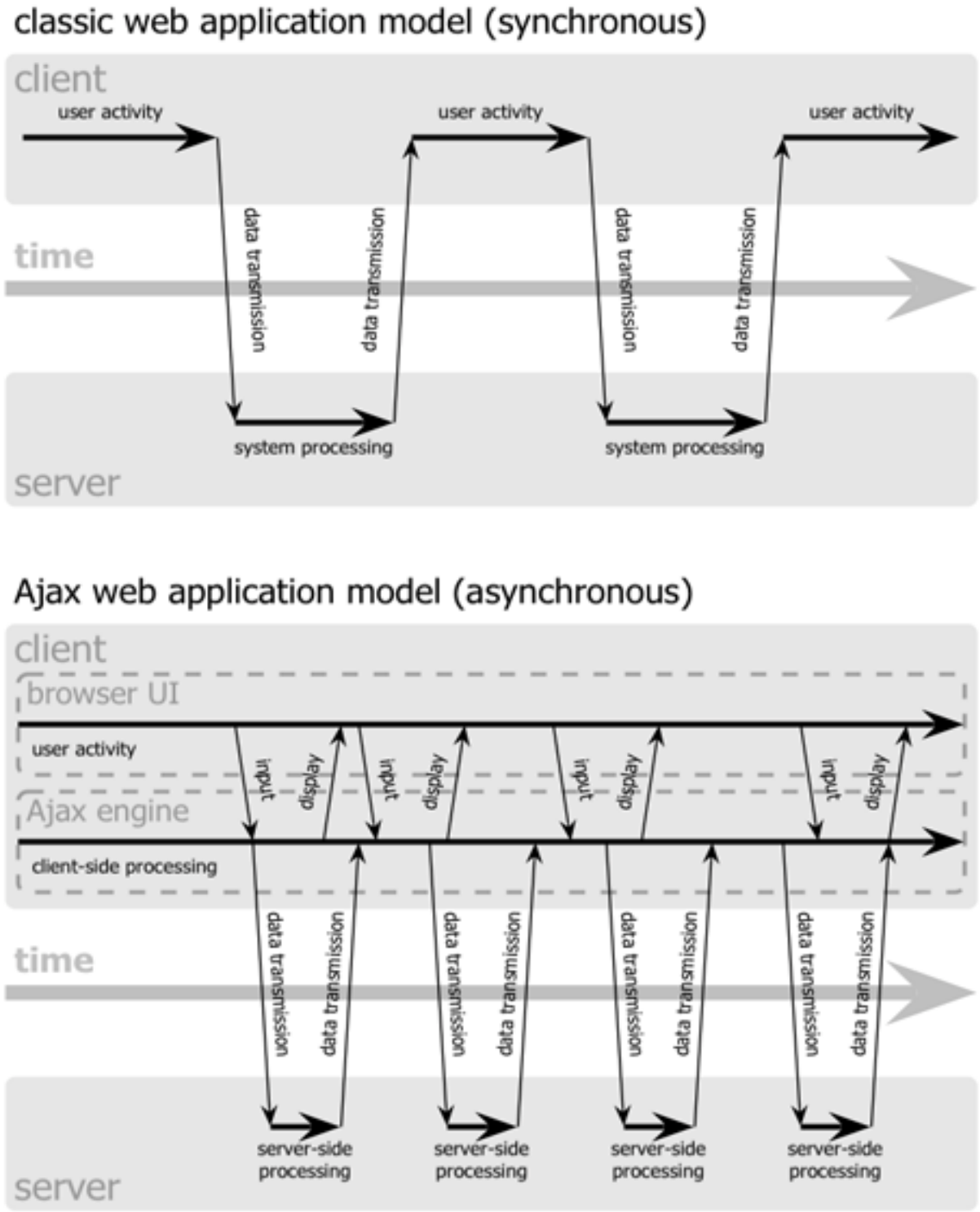

Figure 2.5 Differences in client-server interaction between Classic and AJAX application models [5]

In AJAX applications, every user action that normally would generate an HTTP request takes the form of a JavaScript call to the AJAX engine instead. Any response to a user action that doesn't require a trip back to the server — such as simple data validation, editing data in memory, and even

some navigation — the engine handles on its own. If the engine needs something from the server in order to respond, like submitting data for processing, loading additional interface code, or retrieving new data — the engine makes those requests asynchronously, usually using XML or JSON (JavaScript Object Notation) [13]

However, this approach does not guarantee smooth user experience for all applications. If most operations are data-intensive and require a lot of interaction with server, the gain from AJAX implementation of web applications would be very slim.

Pros

AJAX has the following advantages over conventional web applications:

Bandwidth utilization

By generating the HTML locally within the browser, and only downloading JavaScript calls and the actual data from server, AJAX web pages subjectively load quicker since the payload is smaller in size.

An example of this technique is a large result set retrieved from a database where buffers of data are iteratively retrieved through a database scan. Instead of downloading at once the whole dataset that might not even be inspected by user, the AJAX engine fetches only one buffer at a time, which results in much faster display of the result.

In this project a demonstration program highlights this feature by introducing pagination of result scans. Web application programmers can specify the size of a scan buffer, i.e. how many elements will be read from the server at the time. It dramatically reduces required bandwidth and improves application response rate when handling big scans.

Responsive User interface

The main reason for using AJAX is an improvement to the user experience. Pages using AJAX behave more like a standalone application than a typical web page. By contrast, conventional clicking on links that cause the entire page to refresh feels like a "heavy" operation. AJAX allows a page to be updated dynamically in the background, allowing a faster response to the user's interaction.

In the demonstration program, only the retrieval of the first buffer of a scan takes time to render, the subsequent calls to fetch more buffers are next to instantaneous as there is no need to contact server since the page is cached on the web server.

Cons

The following are the main disadvantages with the AJAX approach.

Browser integration

The dynamically created page does not register itself with the browser history engine, so triggering the "Back" function of the users' browser might not bring the desired result.

Another important issue to be addressed is that dynamic web page updates make it difficult for a user to bookmark a particular state of the application.

Response-time concerns

Network latency — or the interval between user request and server response — needs to be considered carefully during AJAX development. Without clear feedback to the user, smart preloading of data and proper handling of *XMLHttpRequest* objects, users might experience delay in the interface of the web application, something which they might not expect or understand.

Search Engine Optimization

Websites that use AJAX to load data that should be indexed by search engines must provide equivalent data in a format that the search engine can read, because search engines do not generally execute the JavaScript code required for AJAX functionality.

Javascript compatibility

AJAX relies on Javascript, which is usually implemented differently by different browsers or versions of a particular browser. Because of this, web sites that use Javascript may need to be tested in multiple browsers to check for compatibility issues. It is not uncommon to see a Javascript code repeated for different browsers, e.g. for both IE and Mozilla compatibility.

2.4 Google Web Toolkit

"Google Web Toolkit (GWT) is an open source Java development framework that lets you escape the matrix of technologies that make writing AJAX applications so difficult and error prone. With GWT, you can develop and debug AJAX applications in the Java language using the Java development tools of your choice. When you deploy your application to production, the GWT compiler translates your client-side Java application to browser-compliant JavaScript and HTML."[1]

Java technologies offer a productive development platform, and with GWT, they can instantly become the basis of your AJAX development platform as well. Here are some of the benefits of developing with GWT [6]:

- ✓ You can use all of your favourite Java development tools (e.g. Eclipse, IntelliJ, JProfiler, JUnit) for AJAX development.
- ✓ Static type checking in the Java language boosts productivity while reducing errors.
- ✓ Common JavaScript errors (typos, type mismatches) are easily caught at compile time rather than by users at runtime.
- ✓ Code prompting/completion is widely available.

Java-based OO designs are easier to communicate and understand than Javascript, thus making your AJAX code base more comprehensible with less documentation.

GWT provides the following features:

- *Dynamic, reusable UI components*

  It is possible to create new widgets by composing other widgets and arranging widgets automatically in panels.
- *Really simple Remote Procedure Calls (RPC)*

  To communicate from your web application to your web server, you just need to define serializable Java classes for your request and response. GWT's RPC mechanism allows to throw exceptions across the wire.
- *Browser history management*

  GWT lets you make your site more usable by easily adding state to the browser's back button history.
- *Real debugging*

  In production, your code is compiled to JavaScript, but at development time it runs in the Java virtual machine. That means when your code performs an action like handling a mouse

event, you get full-featured Java debugging, with exceptions and the advanced debugging features of IDEs like Eclipse.

- *Automatic browser compatibility*

  Your GWT applications automatically support IE, Firefox, Mozilla, Safari, and Opera with no browser detection or special-casing within your code in most cases.

- *Internationalization*

  It is easy to create efficient internationalized applications and libraries.

- *Interoperability and fine-grained control*

  If GWT's class library doesn't meet your needs, it is possible to mix handwritten JavaScript with Java source code using JavaScript Native Interface (JSNI).

- *Complete Java environment on server-side of application*

  GWT allows to perform any logic required on the server, because it is uses plain Java language and is similar to Java Server Pages in this sense.

  However, it requires to have a JSP (Java Server Pages) capable web-server or application server (Tomcat, Glassfish, WebLogic, WebSphere, Resin or any other) to work properly.

- *Completely Open Source*

  All GWT source is released under the Apache 2.0 license.

2.5 GWT application development cycle

GWT allows to utilise quite efficient and comfortable way of developing web applications. While it is possible to use a simple text editor and command-line compiler, it is highly recommended to use some Java development IDE (Integrated Development Environment). This project was developed using the Eclipse open source Java IDE.

It should be noted that there are two distinct parts of a web application – the client and server-side code. GWT generates the client-side code by compiling Java to Javascript. It can be deployed easily to any web server and used without any hassle. However, since the final code is Javascript, which is more limited than Java, only subset of Java language can be used in client-side code.[7][9][17]

On the other hand, the server-side code is not translated into JavaScript. Therefore it is here possible to use Java without limitation. However it comes at expense of deployment complexity, as it requires a JSP enabled application server.

There are also experiments to utilise other technologies than JSP on the server side.[15]

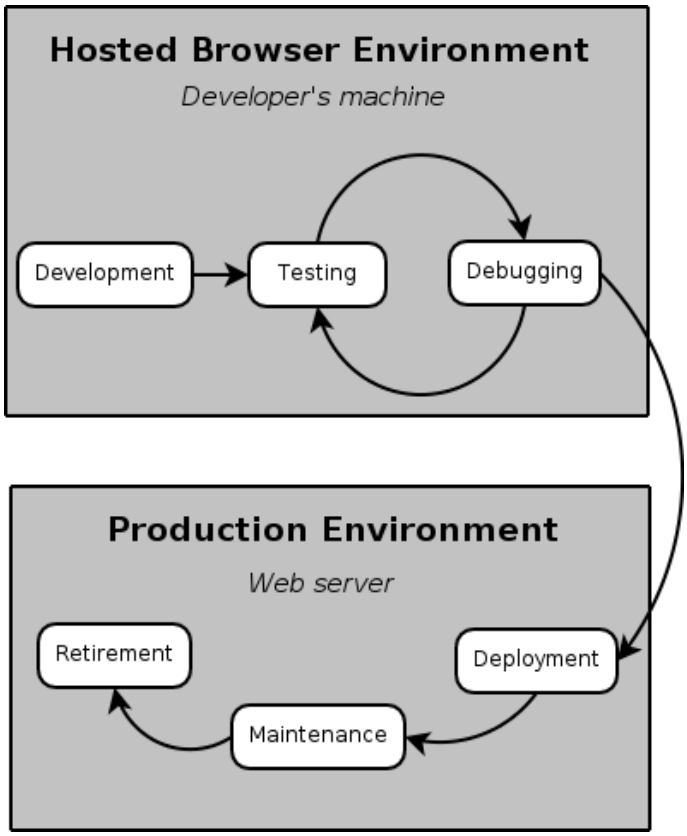

Figure 2.3 GWT Application Development Cycle

In GWT-Amos it is necessary to have Amos II installed and configured properly in order to be able to use the API. More specific instructions on how to configure working environment are covered in installation notes that come with the system.

During development the application is tested and run in 'hosted mode' where GWT provides a special environment that allows running applications without setting up any web servers locally. Technically, the hosted mode runs a stripped-down embedded version of Tomcat that is preconfigured to run GWT applications.

Once the necessary functionality has been implemented, the application is deployed on JSP-enabled application server. The trickiness of this operation depends on selected server. The most popular choice among GWT developers seems to be Tomcat.

3 The GWT-AMOS system

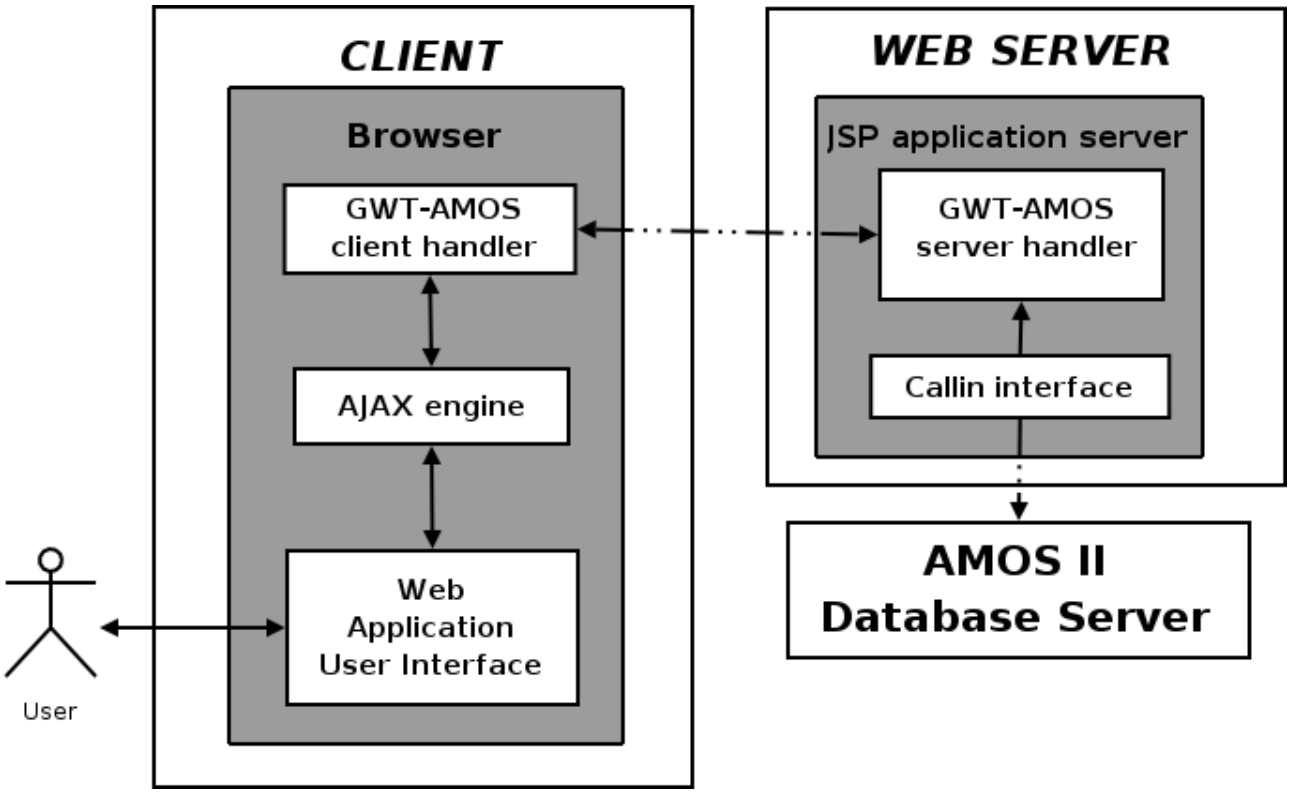

Figure 3.1 System component interactions

Figure 3.3 illustrates a web application client written with GWT that runs in a web browser. It accesses an Amos II database server through the GWT-AMOS API. GWT-AMOS connects three subsystems: a web *client*, a web *server*, and an Amos II *database server*.

Initially the user connects to a GWT-AMOS enabled web server. The client-side GWT generated code is automatically downloaded to the browser. The download includes the GWT *AJAX engine* that determines the browser in use and launches the client side code. The client side code running in Javascript includes the AJAX engine, the *web application*, and the *client handler*. The client handler manages the client side of the API. It communicates with the GWT- AMOS *server handler.* The server side code running in Java under Tomcat is never changed and can be launched once and for all. It runs the server handler that receives messages from the client manager, calls an *Amos II database server* through the *callin* interface, and manages database connections and scans. Both the client and the server handlers are written using GWT.

When a user requests information from the server, the client handler sends a *request* to the server handler using GWT's asynchronous RPC mechanism over HTTP [16]. When the server handler has

processed the request it sends the result back to the client as a *request reply* message. The reply message is registered as a *callback object* where the method *onSuccess(result),* the *reply handler*, is called when the request reply arrives. While waiting for the reply request from the server the application continues to run and the user can interact with it. The application is thus event driven and does not poll the server.

If some error is detected by the server manager it sends back an *error notification* to the client where an *error handler* is invoked in the application code. If the database server throws an exception, it is caught by server handler. Error handlers are implemented as user provided *onFailure(exception)* methods of the callback objects.

The first GWT-Amos application request is to connect to some Amos II database server using the *connect(dbname)* method. When a request to connect to the database server arrives from the client to the server handler it opens a connection to the designated database server through the Amos II callin interface. Once the connection to the database server has been successfully established, the server handler saves the *connection handle* received from the database interface in a JSP session variable and sends a request reply back to the client confirming successful connection to the database server. An error notification is sent if the connection fails.

Each client can connect to only one database server. However, it is possible for different client applications to connect to different Amos II database servers through the server handler since the connection handle is stored in a session variable that is private to each client.

Once the connection is successfully established the application can send GWT-AMOS *database requests* to the server. There are two kinds of database requests: AmosQL *queries* and Amos II *function calls*. All database requests use the client's established connection object.

The queries are sent to the server handler by the *execute(query)* method. The server handler sends the query to the database server for evaluation. The result from Amos II queries are always returned as *scans* by the callin interface [21]. The *scan handle* is stored on the web server in a session variable. Each client can have only one such *current scan* stored on the server. It is the result of the latest query or Amos II function call.

The contents of the current scan is sent back to the client as request reply messages containing *scan blocks*, which are data tables (objects of class *DataTable*) containing an application specific number of rows (e.g. 10) to be returned at the time. Normally the number of rows in a scan block is the same as the number of data values to display on the screen. The reply request handler displays the returned scan block. If the client calls the *next()* method the server handler sends back the next scan

block of the current scan. If the scan has finished an error notification is raised. If the client issues a new database request the current scan is replaced with the new result scan.

Amos II function calls are handled in a similar fashion as queries by calling the method *callFunction(function,arguments)*. It takes a function name as a string argument and an argument list represented as a data table with one row.

Due to limitations imposed by client side GWT code, which supports only a subset of Java, it is not possible send complex Java objects to the client. Therefore the data exchanged between the client and server handler are marshalled using a special data structure represented by class *DataTable*. It encodes all data as strings.

3.1 GWT-AMOS API

The API is implemented by the classes *AmosRPCAsync* and *DataTable* where the methods of *AmosRPCAsync* are used for communication with the database server and the class *DataTable* represents data tables passed between the application and the server, such as parameters of Amos II functions and scan buffers.

3.1.1 Database communication

All methods in class *AmosRPCAsync* have a user provided callback object of GWT class *AsyncCallback* as the last argument. Class *AsyncCallback* has two abstract methods *onSuccess()* and *onFailure()* that implement specific reply and error handlers.

void connect (String db, AsyncCallback callback)
This method connect to an Amos II database server named *db*.

DataTable execute (String query, int size, AsyncCallback callback)

This method sends an AmosQL query to the Amos II database server. The *size* parameter is used to define size of the result scan buffer, i.e. how many rows should be returned at the time. The rest of the scan remains on the server until the method *next* is called. The result is always **null** since the call is asynchronous. However GWT still requires to specify a method return type [16].

DataTable next (int size, AsyncCallback callback)

This function returns next scan buffer of the current scan. If the scan is finished the server handler

throws a *GWTException*. The result is always **null**.

DataTable callFunction (String fname, DataTable data, int size, AsyncCallback callback);

This method fetches a scan buffer as for *execute*. The *size* parameter is used to define the size of the result scan buffer, i.e. how many rows should be returned at the time. The rest of the scan remains on the server until *next* is called.

If no function with provided name exists, *callFunction* will raise an error exception.

3.1.2 Data tables

The class *DataTable* holds data values passed between the application and the database. A table consists of a number of rows and each row has a number or elements. The rows and the elements are index by row and element numbers. The following methods manipulate data tables:

Table access

int getSize()

returns number of rows in the table.

int addRow ()

appends new empty row to the table.

void delRow (int nr)

deletes selected row from the table.

boolean isLast()

The method returns true if this data table was the last scan block.

Row access

int getLength(int nr)

returns the length of a row.

void delElem (int nr, int index)

deletes the selected element from a row.

void addElem(int nr, int value)

adds an integer to the end of a row.

void addElem(int nr, double value)
adds a double to the end of a row

void addElem(int nr, String value)
adds a string to the end of a row.

void addElem(int nr, boolean value)
adds a boolean to the end of row.

void addOidElem(int nr, int value)
adds an Amos II OID object to the end of row. Since AMOS OID objects can not be used on the client, only the OID id value is stored, which is used by server handler to construct the corresponding OID object.

Element access

Since it is impossible to return different types of data by one method, different methods are used to access elements in a row depending on their data types.

String getString(int nr, int index)
returns the selected element converted to a string.

int getIntElem (int nr, int index)
retrieves an integer.

double getDblElem (int nr, int index)
retrieves a float.

String getStrElem (int nr, int index)
retrieves a string.

boolean getBoolElem (int nr, int index)
retrieves a boolean.

String getOidElem (int nr, int index)

Retrieves a string representing a database object identifier on the format“[OID <no>] ” where <no> is the identifier the database server has assigned to the OID.

Element types

It is recommended to check for the data type of an element before accessing it. The following methods check for the data type of an element in a row.

String getElemType (int nr, int index)

returns a string representing the type of a selected element. This method returns one of the following values:

o – object ID (then its ID is stored)

b – Boolean

s – String

i – Integer

d – Double

v – Vector

The data type of an element can also be tested with the following methods:

boolean isInt (int nr, int index)

boolean isDbl (int nr, int index)

boolean isStr (int nr, int index)

boolean isBool (int nr, int index)

boolean isTpl (int nr, int index)

3.1.3 Exceptions

The class *GWTException* holds only AMOS error messages. GWTException is passed to the error handler on client side when an AmosException on server is raised. It has one method:

String getReason()

returns an error message, indicating a problem that happened during call to AMOS database

3.4 GWT-AMOS implementation

The system is implemented by six Java classes, of which four belong to the client handler and two to the server handler.

GWT Naming conventions

A s*ervice interface* is a set of methods that is used in client-server communication, for example the methods in Section 3.1.1**.** The relationship between a service interface and the corresponding asynchronous GWT-RPC method is straightforward:

- Assume a service interface is called **com.example.client.Service**,
- The corresponding asynchronous GWT-RPC is then called **com.example.client.ServiceAsync**.
  The asynchronous interface must be in the same Java package and have the same name, but with the suffix `Async`
- `The server side implementation of the same service must be called` **com.example.client.ServiceImpl**.

Client Classes

AmosRPC.class – class that contains the same methods as those defined for class *AmosRPCSynch*.

AmosRPCAsync –class implementing the API as explained earlier.

DataTable implements API data structure representing data tables, as explained earlier.

GWTException is a class derived from Java's *Exception* class that is used to notify clients about errors detected on the server. These exceptions can be serialised and sent back to the client.

Server Classes

AmosRPCImpl.class implements the server handler.

Amos.class implements connections to the AMOS II database system in the server handler.

3.5 Data Marshalling

One important Javascript language limitation is serialisation. Only primitive types and aggregations containing them are allowed to be exchanged between client and server. It is impossible to use lists

or vectors that contain objects of different types. This restriction is imposed, because Javascript does not support such functionality

Therefore it was essential to develop an efficient data structure, class *DataTable*, that allows representation of tables of rows with elements of any type and cane be easily sent between client and server.

<Type,Value>	**<Type,Value>**	**<Type,Value>**	
<Type,Value>			
<Type,Value>	**<Type,Value>**	**<Type,Value>**	**<Type,Value>**
<Type,Value>	**<Type,Value>**		

Fig. 3.2. DataTable structure.

Figure 5 describes the general format of a *DataTable* structure. The data table is a dynamic array of variable length vectors. Arguments in the vector are stored as string pairs. The first string denotes type of the argument, while second string stores value of the arguments.

The following types can be used in data tables:

- Integers
- Floats
- Strings
- Booleans
- OID
- *Vectors – Lists*

Vector storage format:

The format is "<*type*>_<*value*>

<*type*> can be:

o – object ID (then its ID is stored)

b – Boolean

s – String

i – Integer

d – Double

v – Vector

Vectors are strings with format {<*type*>_<*value*>|<*type*>_<*value*>|...|<*type*>_<*value*

Vector string example: "{s_ab|b_true|s_test|i_1234|d_12.45|o_[OID #1235]|}"

4 Examples of API usage

The following code snippet shows how to connect to a database server. The callback handler *callbackConnect* uses the method *showError()* to display error messages.

```
protected void showError(Throwable caught)
{
GWTException err = (GWTException) caught;
String err_msg = err.getReason();
/* Error visualisation follows */
}

AsyncCallback callbackConnect = new AsyncCallback()
   {
   public void onFailure(Throwable caught) { showError(caught); }
   public void onSuccess(Object result)
     {/* Some code handling successful outcome of RPC */}
   }
/* Create the client proxy.
 * Note that although you are creating the service interface proper, you cast the result
 *  to the asynchronous version of the interface. The cast is always safe because the
 * generated proxy implements the asynchronous interface automatically.
 */
final AmosRPCAsync testRPC = (AmosRPCAsync) GWT.create(AmosRPC.class);
/ * Specify the URL at which our service implementation is running.
  * Note that the target URL must reside on the same domain and port from
  * which the host page was served.
 */
ServiceDefTarget target = (ServiceDefTarget) testRPC;
/* Next line generates address, where servlet is located
String moduleRelativeURL = GWT.getModuleBaseURL() + "/date";
/* Registering TomCat Servlet Entry Point on client*/
target.setServiceEntryPoint(moduleRelativeURL);
/* Connecting to server */
testRPC.connect("", callbackConnect);
```

The following snippet shows how to call an Amos II database function:

```
protected void showError(Throwable caught)
{
GWTException err = (GWTException) caught;
String err_msg = err.getReason();
/* Error visualisation follows */
}
AsyncCallback callbackCall = new AsyncCallback()
   {
   public void onFailure(Throwable caught) { showError(caught); }
   public void onSuccess(Object result)
     {
     DataTable table = (DataTable) result;
     /* Some code handling successful outcome of RPC */
     }
}
args = new DataTable(); // Creating new table for arguments
args.addRow();
args.addElem(0, 256); // Adding argument to argument lists
testRPC.callFunction("id", args, size, callbackCall); // Sending function call
```

The following snippet returns the 10 Amos II function names:

```
protected void showError(Throwable caught)
{
GWTException err = (GWTException) caught;
String err_msg = err.getReason();
/* Error visualisation follows */
}
AsyncCallback callbackNext = new AsyncCallback()
   {
   public void onFailure(Throwable caught) { showError(caught); }
   public void onSuccess(Object result)
```

```
        {
        DataTable table = (DataTable) result;
        /* Some code handling successful outcome of RPC */
        }
      }
String query = "select name(f) from function f;"; // Generating query to be sent to server
int size = 10;
rpc.execute(query, size, callbackNext);
```

Summary

In this project a GWT-based interface to the Amos II DBMS has been implemented. It allows to create easily new web applications using GWT that use the Amos II system without writing or deploying any server-side code. The interface provides basic database operations to connect to an Amos II server, to execute AmosQL queries, and to call Amos II functions.

References

[1] Google Web Toolkit. http://code.google.com/webtoolkit/

[2] Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, and Martin Sköld. Amos II 8 User's Manual. UDBL Technical Report, Dept. of Information Science, Uppsala University, Sweden, March 29, 2005 (Latest revision September 8, 2006) http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html

[3]. T. Risch, V. Josifovski, and T. Katchaounov: Functional Data Integration in a Distributed Mediator System, in P. Gray, L. Kerschberg, P. King, and A. Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analysing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4, 2003. http://user.it.uu.se/~torer/publ/FuncMedPaper.pdf

[4] Elin, D, Risch, T: Amos II Java Interfaces, UDBL, Uppsala University, Sweden, August 2000,

http://user.it.uu.se/~torer/publ/javaapi.pdf

[5] J. J. Garrett. February 18, 2005. Ajax: A New Approach to Web Applications http://www.adaptivepath.com/publications/essays/archives/000385.php

[6] GWT overview. http://code.google.com/webtoolkit/overview.html

[7] GWT JRE Emulation Library http://code.google.com/webtoolkit/documentation/jre.html [8] Wikipedia: AJAX. http://en.wikipedia.org/wiki/AJAX

[9] GWT language support. http://code.google.com/webtoolkit/documentation/com.google.gwt.doc.DeveloperGuide.Fundamentals.JavaToJavaScriptCompiler.LanguageSupport.html

[10] W3C: Cascading Style Sheets http://www.w3.org/Style/CSS/

[11] W3C: XHTML http://www.w3.org/TR/xhtml1/

[12] W3C: XMLHttpRequest http://www.w3.org/TR/XMLHttpRequest/

[13] JSON (JavaScript Object Notation) http://www.json.org/

[14] W3C: Document Object Model (DOM) http://www.w3.org/DOM/

[15] GWT Guide. Using GWT with MySQL and PHP.

http://angel.hurtado.googlepages.com/tutorialgwt2

[16] GWT Remote Procedure Calls
http://code.google.com/webtoolkit/documentation/com.google.gwt.doc.DeveloperGuide.RemoteProcedureCalls.html

[17] GWT Runtime Library Support.
http://code.google.com/webtoolkit/documentation/com.google.gwt.doc.DeveloperGuide.Fundamentals.JavaToJavaScriptCompiler.JavaRuntimeSupport.html

[18] G Wiederhold: Mediators in the Architecture of Future Information Systems. IEEE Computer, 25(3), 38-49, 1992.

Appendix: A simple application

The code below is the complete code of an application where the user enters an Amos II query as a string that is sent to the database for evaluation. The first 10 rows of the result scan are displayed on the screen. The user can ask for the next 10 rows etc. until the scan is exhausted. It illustrates the functionality of the system.

```
package se.uu.client;

import com.google.gwt.core.client.GWT;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.rpc.ServiceDefTarget;
import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.Widget;
import com.google.gwt.user.client.ui.TextArea;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.FlexTable;

public class SimpleApp implements EntryPoint {

	protected void showError(Throwable caught)
	{
	GWTException err = (GWTException) caught;
	err_msg.setText(err.getReason());
		RootPanel.get().add(err_msg);
	}

	final Button qrymore =   new Button("Next page");
	final Button callQuery =    new Button("Execute query");
	final int size = 10;
	Label  err_msg = new Label("");
	FlexTable query_result = new FlexTable();
	TextArea TA = new TextArea();

	AsyncCallback callquery = new AsyncCallback()
	{
	public void onFailure(Throwable caught)     { showError(caught); }
	public void onSuccess(Object result)
	{
	DataTable info = (DataTable) result;
	query_result.setBorderWidth(1);
		if (info.isLast) { RootPanel.get().remove(qrymore); }
	for (int k = query_result.getRowCount()-1; k >= 0;k--)
		query_result.removeRow(k);
		for (int i = 0; i < info.getSize(); i++)
			for (int j = 0; j < info.getLength(i); j++)
```

```
                        query_result.setText(i, j, info.getString(i, j));
    }
    };

    AsyncCallback answer = new AsyncCallback()
            {
            public void onFailure(Throwable caught) {  showError(caught);    }
      public void onSuccess(Object result)
            {
                    err_msg.setText("Connected Successfully!");
                    RootPanel.get().add(err_msg);
            }
            };

     public void onModuleLoad() {
            final AmosRPCAsync testRPC = (AmosRPCAsync) GWT.create(AmosRPC.class);
             ServiceDefTarget target = (ServiceDefTarget) testRPC;
             String moduleRelativeURL = GWT.getModuleBaseURL() + "/date";
             target.setServiceEntryPoint(moduleRelativeURL);

             try {testRPC.connect("", answer);}
             catch (GWTException e) { e.printStackTrace(); }

             qrymore.addClickListener(new ClickListener() {
                      public void onClick(Widget sender)
                     {
                     try { testRPC.next(size, callquery);}
                     catch (GWTException e) { e.printStackTrace();}
                     }
                     });

             callQuery.addClickListener(new ClickListener() {
                      public void onClick(Widget sender)
                     {
                     try { testRPC.execute(TA.getText() , size, callquery);}
                     catch (GWTException e) { e.printStackTrace(); }
                     RootPanel.get().add(query_result);
                     RootPanel.get().add(qrymore);
                     }
                     });
      RootPanel.get().add(TA);
            RootPanel.get().add(callQuery);
     }
    }
```