

FerryWeb - A Web 2.0-based demonstrator for Ferry

Studienarbeit

Lehrstuhl für Datenbanksysteme
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

von
Thomas Ressel

Betreuer: Tom Schreiber, M.Sc.

February 7, 2010

Summary Ferry, an intermediate language between relational databases and various programming or scripting languages, allows programs to out-source data-intensive operations to DBMS such as IBM DB2, taking advantage of their ability to handle big amounts of data. The decision whether to use the programming language or a DBMS for a specific data-intensive operation is taken without bothering the programmer. For demonstrating and teaching purposes we developed a web application - FerryWeb - that allows users to experiment with the compilation of Ferry code. FerryWeb is based on GWT, Google's Web Toolkit, which provides a variety of tools (such as cross compilation of Java code to JavaScript code) for the developer of web applications. The Ferry language, concepts of GWT and implementation details of FerryWeb are the topic of this thesis.

Contents

1	Introduction	3
2	Ferry	3
2.1	An introducing example	3
2.2	The Ferry language	5
3	GWT	6
3.1	Anatomy of a web application	7
3.2	The GWT approach	9
4	FerryWeb	10
4.1	The FerryWeb user interface	11
4.1.1	File Tree	11
4.1.2	Code Widget	11
4.1.3	The result window	12
4.2	Services of FerryWeb and their implementation	12
4.2.1	The GWT RPC mechanism	13
4.2.2	The FerryWeb FileManagementService	13
4.2.3	The FerryWeb ProcessFerryCodeService	15
4.3	Software architectural aspects	15

1 Introduction

With the growth and evolution of the internet and the abilities of today's browsers, the way people use and work with computers has fundamentally changed. The concept of the classic desktop application is vanishing. Hand in hand with that recent trend goes the well-known keyword Web 2.0 which follows the version numbering in software engineering but rather means the step from people getting information in the web passively to people changing web site content pushing the web more to a general information platform. Nevertheless this version 2.0 points to set of web technologies related to this new mode of use of the web.

In this thesis, a web application is presented which is intended to be used in teaching and conferences to demonstrate the abilities of the Ferry programming language. After introducing fundamental concepts of Ferry, we return to the Web 2.0 topic, giving a short introduction of GWT, a tool kit originated in the Web 2.0 context.

Thereafter FerryWeb and its actual implementation will be topic in Chapter 4, further analyzing aspects of GWT in the context of an actual example.

2 Ferry

Ferry is an intermediate programming language that strives to harness the capacity of DBMS to cope with large amounts of data directly in program evaluation, mapping programming language concepts (*e.g.* complex types, iteration, variable assignment and reference) into set-oriented algebraic programs. These programs can then be converted into (bundles of) SQL:1999 statements, that are evaluated close to the data. Besides DBMS, there are other back-ends thinkable to provide such facilitation for programming language runtimes, as long as the rest upon a set-oriented execution model.

2.1 An introducing example

To give substance to the idea of Ferry, a simple example from [1] is shown here: Given a table containing employees and their salaries, we want to know who are to two best-paid employees for each department. In order to do so, one could imagine a code snippet similar to that in Listing 1: Two nested while loops, the first extracting all departments, the second just reading the two best-paid persons. It is obvious that this code excerpt is pretty voluminous given the simple task and that it further isn't even complete: a lot of code lines have been suppressed such as initialization of variables

Employees			
id	name	dept	salary
1	Alex	DE	300
2	Bert	DE	100
3	Cora	DE	200
4	Drew	US	200
5	Erik	US	400
6	Fred	US	300
7	Gina	US	200
8	Herb	NL	600
9	Ivan	NL	400
10	Jill	UK	500

Figure 1: Employees table.

and database connection management. The programming language runtime (in this case Java) even does work from a query processor in this example, namely the querying.

```

1  /* Let dataBase be an object responsible for database
   connection management. More code suppressed here. */
2  ResultSet rsDEPT = dataBase.executeQueryStatement("SELECT
   DISTINCT dept FROM Employees;");
3  while (rsDEPT.next()){
4      String currentDEPT = rsDEPT.getString("dept");
5      ResultSet rsEMPL =
           dataBase.executeQueryStatement("SELECT name, salary
   FROM Employees WHERE dept=\"" + currentDEPT + "\"
   ORDER BY salary DESC FETCH FIRST 2 ROWS ONLY;");
6      while (rsEMPL.next()){
7          /* Build list of shape [(dept, [(name, salary)])]
8             here with the ResultSets getter methods. */
9      }
10     rsEMPL.close();
11 rsDEPT.close();
12 }

```

Listing 1: Query for Employees table in Java (ODBC)

In a somewhat more *database-supported* programming surrounding, the Ruby script shown in Listing 2 can be written to solve the before mentioned task. Variable `d_es` represents a list of shape `[(dept, [(id, dept, name, salary)])]` which is already "close" to the desired result. In line 2, two nested map invocations are used to map on `name`, `salary` and the final result is obtained with sorting and filtering in line 3 of Listing 2.

```

1 # Objects in Employees have methods id, name, dept, salary
2 d_es = Employees.group_by {|e| e.dept}
3 d_ns = d_es.map {|d,es| [d, es.map{|e| [e.name, e.salary]}]}
4 top2 = d_ns.map {|d,ns| [d, ns.sort_by {|n,s| -s}.first(2)]}
5 # Result top2 is of shape [(dept, [(name, salary)])]

```

Listing 2: Query for Employees table in Ruby

The Ruby functions used in Listing 2 may be expressed in comprehensions [3] and are therefore mapped into Ferry, which works with nested, ordered lists just like Ruby. Listing 3 shows a Ferry sample program equivalent to the Ruby script in Listing 2.

```

1 let e = table Employees (id int, name string, dept string,
2   salary int) with keys ((id))
3 in for x in e
4   group by x.dept
5   return (the (x.dept))
6     take (2, for y in zip (x.name, x.salary)
7       order by y.2 descending
8       return y))

```

Listing 3: Query for Employees table in Ferry

Finally SQL code is derived from this Ferry program and passed to a back-end DBMS. This process is explained in the following section.

2.2 The Ferry language

Ferry is designed for operations and especially iteration over arbitrarily nested, ordered lists and tuples. All data that Ferry processes is shaped as the recursive type equation $t = a \mid [t] \mid (t, \dots, t)$. In this equation, a stands for atomic types like `int`, `string` or `boolean`. All programming language types are built with this single equation. For instance, a hash map can be modeled as $[(int, [string])]$, a simple list comes as (a, \dots, a) and so on.

The Ferry syntax follows a fully-orthogonal concept around the `from-where-group-by-order-by-return` construct (see Listing 3). The similarity to XQuery’s FLWR construct is no coincidence: Both languages rest on *list comprehensions* [3]. The syntactic construct `for x in e1 return e2` iterates over all x ’s in $e1$ and evaluates (side-effect free) the expression $e2$.

The library functions of Ferry were inspired by Haskell’s standard prelude [4]. A normal Ferry program consists of chains of these granular functions to realize a more complex functionality.

To obtain database-executable code, in our context SQL:1999, Ferry’s compiler `ferryc` uses a technique called *loop lifting* [2] that was originally

developed for the purely relational *Pathfinder XQuery* compiler (**pf**). The compilation process is depicted in Figure 2. **ferryc** makes use of **pf** to translate algebraic plans into SQL:1999 statements which are fed to the database backend. There are several compilation stages depending on how often the **pf** compiler is run.

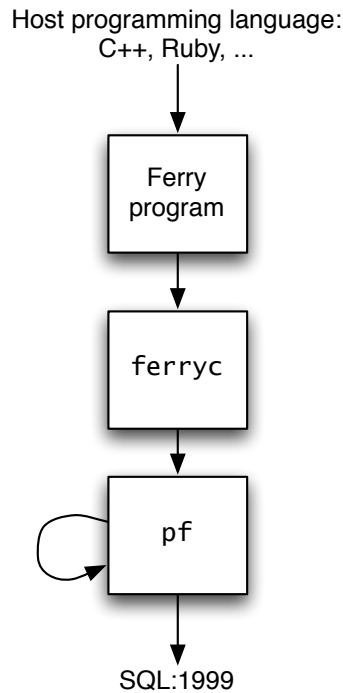


Figure 2: Ferry Compilation Stages. **pf** denotes Pathfinder’s optimizer and code generator.

3 GWT

At a glimpse The Google Web Toolkit equips its users with powerful tools for developing web applications, hiding away some of the complexity of such applications. A major functionality is relieving the developer from the burden of writing code in different programming languages, as it cross-compiles Java code into JavaScript code, which is not only translated but also highly optimized per major browser. GWT also provides efficient solutions for common problems in developing web applications such as undefined (or at least not the from the user intended) behavior of the back/forward button in the browser.

3.1 Anatomy of a web application

A web application (shorter: web app) is an application which is accessed in a web browser over a network such as the Internet. Other than in classic applications, the actual computations are performed on a web server, while the accessing client only performs the presentation.

Web applications usually consist of several tiers that play roles in the functionality of the application. A common structure is the so called *three-tiered* application, which consists of the tiers *presentation*, *application* and *storage*. Presentation stands for the client, mostly a web browser, whose only task is to represent an (graphical) user interface. The application layer (also: *application logic*) produces dynamic Web content and is where the requests from the client are sent to. Finally, the storage tier takes care of the data management (most commonly by the use of a database). Figure 3 shows these tiers.

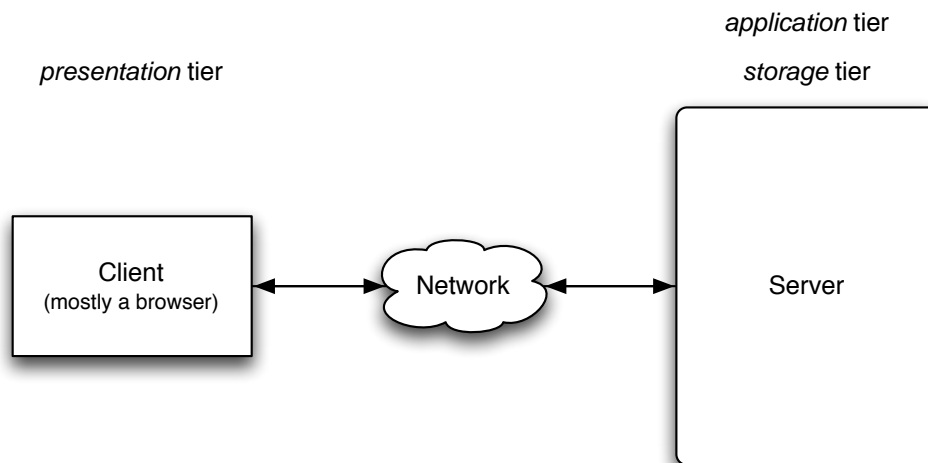


Figure 3: Anatomy of a web application. Along the arrows go requests and responses.

Since GWT uses a `JavaServlet`-based RPC mechanism (see 4.2.1), we are concentrating on web applications according to the `JavaServlet` specification. `Java servlets` (from *server* and *applet*) are server-side `Java` classes that receive requests from clients and answer them dynamically creating content, `HTML` pages for instance.

Web applications according to the `JavaServlet` specification are shipped in so called `WAR` files that are basically `JAR` files containing everything that forms a web application: `Static HTML` and `CSS` files, `Java` classes, `XML`

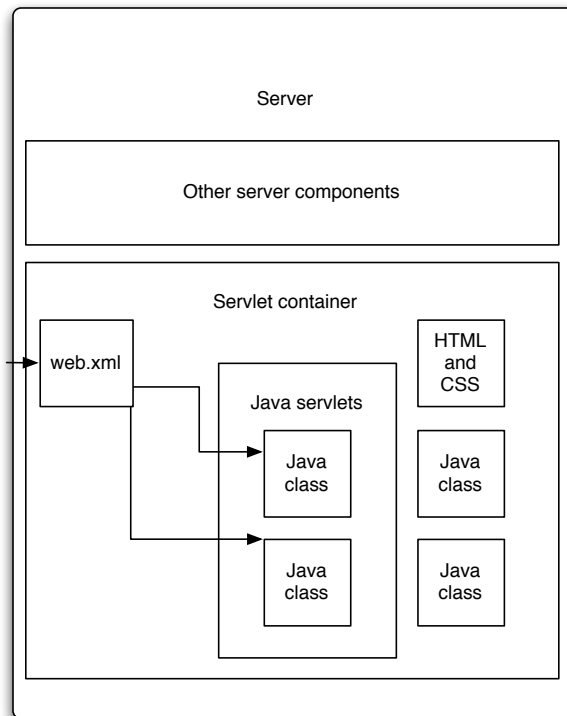


Figure 4: Structure of a web server running Java servlets

files and JavaServlets. One important file is the *web.xml* file, which contains mapping information for the server the application is run on.

On a server, the application is then run in a so called *Servlet container* which maps requests from clients to the corresponding Servlets. See Figure 4 for the structure of a server.

Difficulties developing a web application The environment a web application is run in brings up some issues in the developing process. Since web applications change the HTML documents in the browser dynamically, the forward/back buttons can show other functionality than the expected. GWT solves this inconvenience with a history stack on which actions that should be possible to undo via the back button are simply put on. Developing web applications also brings up performance issues since it is not clear where the application is finally run on. Specific quirks of web applications such as having to send a HTTP request for every single picture on a web site is another example. GWT also offers a solution for that problem, a so called Image Bundle, which bundles up all images to one package, that is then sent

via a single HTTP request to the client.

Difference to Web service While in a web application, the interaction with a user is mandatory, a *web service* does not necessarily interact with a user. Its results are normally represented as XML documents and used within other applications without the users notice.

3.2 The GWT approach

Its obvious that the situation in Figure 3 is heavily simplified. First, there are a variety of browsers that act as clients. These browsers can be run on systems that vary strongly when it comes to hardware architectural aspects. Every browser has also some kind of peculiarities that the developer has to face in order to write an application that works properly *everywhere* where it is run. Hardware architectural differences behind browsers and JavaScript in general lead to performance issues which directly influence the user experience in a negative way. But there can arise more performance limiting problems such as DOM layout and CSS style matching. All these difficulties make developing a web application an error-prone and time consuming task.

Google confronts these challenges with a set of tools that in total facilitates the whole development process. This process is separated in different phases that Google calls *write*, *debug*, *optimize* and *run* and that are described more precisely in the following. A more detailed description of the mentioned features and the structure of a GWT project is later illustrated at the example of FerryWeb in Chapter 4.

Write In GWT, everything is written in Java. For the client side, this Java code is later cross-compiled into JavaScript which is then optimized per browser. This allows the developer to use all the productivity augmenting tools available for Java, such as error highlighting. This may seem like a trivial example, but in JavaScript, typos are only identified at run time. GWT comes with a set of Widgets and APIs for building user interfaces and logic for the client, with the benefit of being able to reuse UI components even in different projects. It also provides an optimized RPC mechanism to facilitate the communication between server and client. But this "all is Java" paradigm is only half of the story. It is of course possible to run the server with anything other than Java, as well as its possible to mix in handwritten JavaScript code into the client-side Java code. Not using Java on the server side brings the inconvenience of loosing the possibility to use the GWT RPC mechanism, which bases on JavaServlets. In this case, *JavaScript Object*

Notation (JSON) can be used.

Debug In developing mode the server and client both run Java byte code. This has the great advantage of being able to use the Java debugging methods such as setting breakpoints, inspecting variables and stepping through the code. Besides, the application can be debugged in any browser at this moment, since running the application from an IDE provides an URL that simply can be pasted in any browser running on the system. This also inherits the ability to make use of further debugging tools that exist for those browsers.

Optimize JavaScript can be a bottle neck, but not the only one. Every browser runs JavaScript in a different way, which results in different run time. That could seriously impact the user experience. GWT solves this with cross-compiling optimized code per browser, which is aware of the specific browser quirks with the positive side effect that users download only code for the browser they are using, which reduces download time. As applications grow, this "per browser" code can again be splitted into smaller pieces to reduce for example the startup time of a more complex application. Another bottle neck can be CSS and generally the way browsers layout pages. For the latter, Google added a tool named *SpeedTracer* (since version 2.0) which shows performance measurements for the Google Chrome browser as it runs a certain web app. This simplifies to a great extend the task of finding out where time is wasted.

Run The last step is packaging the web app into a WAR file (see 3.1) which can be run in common JavaServlet containers, such as Apache Tomcat. Another possibility is to deploy it to Googles AppEngine.

4 FerryWeb

There is one major task FerryWeb shall offer to users and that is to compile Ferry code and represent the results of the compilation in the web. Ferry code shall be entered by the user directly or it is loaded from a file the user can choose.

The corresponding desktop application, which provides many more features (*e.g.* compile-as-you-type) is called *FerryDeck*.

4.1 The FerryWeb user interface

With this specification, the design of the user interface is pretty clear. It is shown in Figure 5.

Building user interfaces is a pretty comfortable task in GWT as it provides a set of widgets, that can directly be built into web applications. FerryWeb takes advantage of that and uses the GWT widgets to implement its UI. As shown in Figure 5 it consists of three parts - a *file tree*, *code widget* and *result window* - which are now described shortly and further explained in 4.1.1, 4.1.2 and 4.1.3, respectively.

File Tree The file tree is a standard file navigation tree and lets the user browse through files with Ferry code, that reside on the server. It is also possible to create new files and folders as well as change the file contents. This feature can also be switched off, so that the file tree is read only.

Code Widget In the code widget, the user can enter Ferry code, which will then be compiled via a button. It is also possible to set the degree of compilation (see Chapter 2).

The result window The result window bases entirely on Cover Flow, originally invented by Andrew Coulter Enright, later purchased by Apple Inc. and used in products like iTunes and Mac OS X. FerryWeb uses a JavaScript-based implementation called ImageFlow by Finn Rudolph.

4.1.1 File Tree

The file tree consists basically of a composite widget that conjoins GWT widgets such as a standard tree, labels and buttons. There are two arguments to pass over at instancing a file tree: A code widget and a boolean flag, that specifies whether the files in the tree should be read only or if the user should be able to change files and its contents.

The file tree's obvious task is thus the representation of the file structure on the server. therefore, it has to know the format of the file representation that the server sends to it. A quick glimpse of that format is given in Figure 8, for a further explanation see Chapter 4.2.2.

4.1.2 Code Widget

The Code Widget is mainly a text area widget that can be easily replaced via wrapper class. That way, besides the standard GWT text area there can

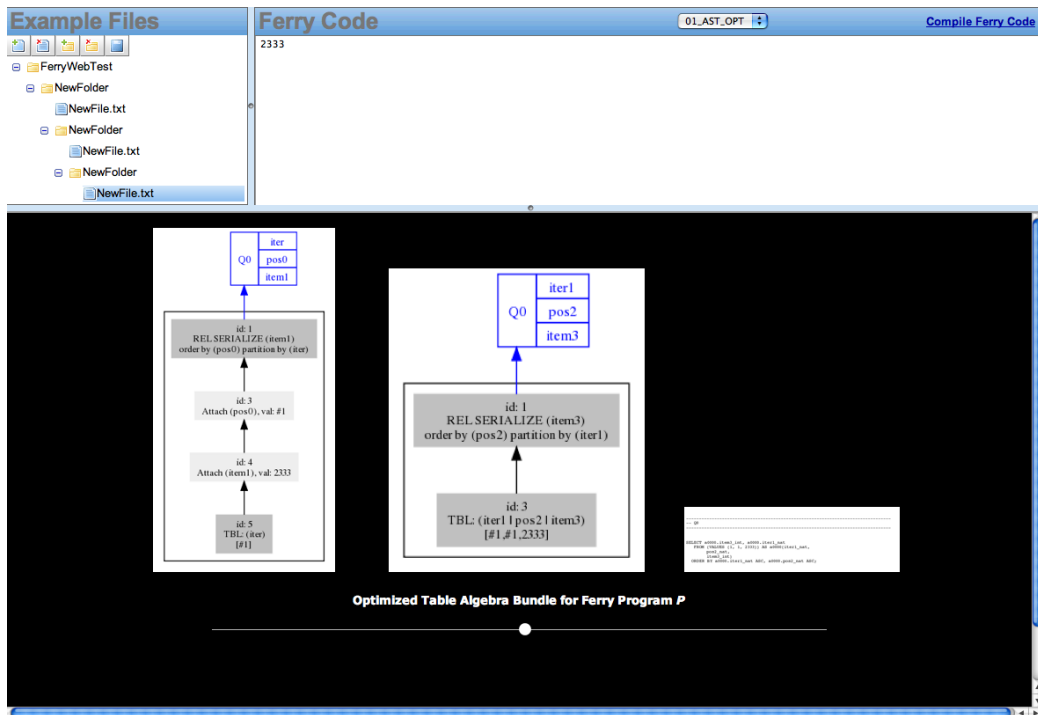


Figure 5: FerryWeb user interface layout. On the upper left the file tree, on the upper right the code widget and on the bottom the result window.

be put other widgets in the future, *e.g.* one that can syntax highlight the code.

4.1.3 The result window

ImageFlow is a very easy-flowing implementation of CoverFlow. Its based on JavaScript and therefore highly platform independent. It consists of one folder containing an HTML file, some CSS style sheets and of course JavaScript files. The images to show are specified in the HTML file, which fits perfectly the needs of FerryWeb, since the Ferry compiler always produces output files of the same names. With the openness of GWT, CoverFlow can easily be built in into FerryWeb.

4.2 Services of FerryWeb and their implementation

Following the specification at the begin of this chapter, FerryWeb communicates via two services with the user in order to fulfill its task. These services are called *FileManagementService* and *ProcessFerryCodeService*. Both are

implemented with the GWT RPC mechanism. Thus, FerryWeb is a pure GWT application. The representation of the compilation result does not rely on RPC services, since it uses the ImageFlow technology described in 4.1.3. Consequently, it does not take advantage of the Image Bundle concept of GWT, which was mentioned in Chapter 3.

4.2.1 The GWT RPC mechanism

A *remote procedure call* or RPC is a technology for inter-process communication with the two involved processes normally running on different machines connected via a network, having different address spaces. This setting is pretty much the same as in Figure 3. In order to communicate, messages are sent from the client to the known server and an answer is sent from the server to the client. Originally, the client had to wait for the invoked remote procedure to return, which would take away much of the dynamics of an application. Many technologies were developed over the years to solve this inconvenience. GWT uses *Asynchronous JavaScript and XML* (AJAX) on top of RPC to overcome it and to provide a more dynamic user experience. See Figure 6 for an example.

To create a new service in a GWT project, there are two steps to do: First an interface, which extends *RemoteService* has to be created on the client side. This interface defines the method stubs of the service. Then, to implement the functionality of the service, a class which extends *RemoteServiceServlet* has to be written on the server side. There will be automatically created another interface on the client side, called the asynchronous interface. It is directly related to the first hand-written interface and is the interface, that is later actually called from the client. See Figure 7 for an overview of this situation.

4.2.2 The FerryWeb FileManagementService

In the FileManagementService just happens what everybody would expect. There are methods needed to open/save/create/delete files and folders and get a representation of the file system on the server to the client. This is accomplished by an database-entry-similar string per file/folder, that contains the following information: ID, FILE/DIR, NAME, SORTORDER, PARENTID. In the following, an algorithm that composes that strings is shown. Figure 8 shows an example of the output.

The getNavigationTree procedure:

```
begin getFileDirStrings(currentWorkingDirectory)
```

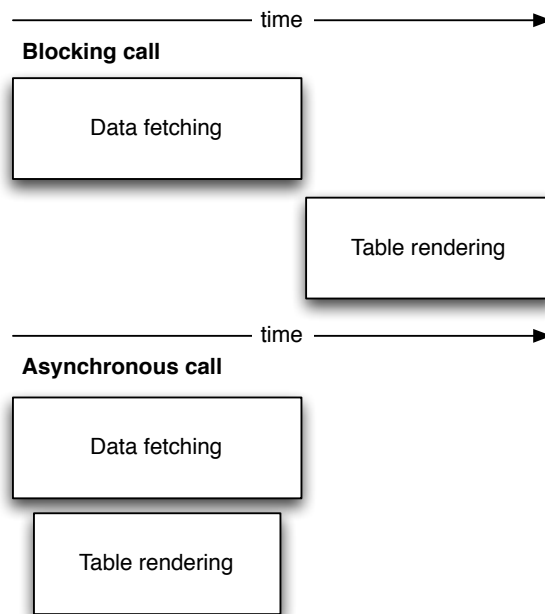


Figure 6: AJAX example call. In the top part of the figure, data fetching has to be completed before rendering the table is possible. Below, data is fetched from the server and the browser renders the table parallelly. (This involves of course more technologies such as DHTML.)

```

where
proc getFileDirStrings(file) ≡
  String[] result
  if file ≡ directory then dirHash.put(file.path, currentID) fi
  result.add(getStringRepresentationOfFile(file))
  for all subfiles child in file do result.addArray(getFileDirStrings(child)) od
where
proc getStringRepresentationOfFile (file) ≡
  (Computes a String representation of file
   of shape ID, FILE/DIR, NAME, SORTORDER, PARENTID.
   HashMap dirHash is needed here for parent determination.)
.
.
end

```

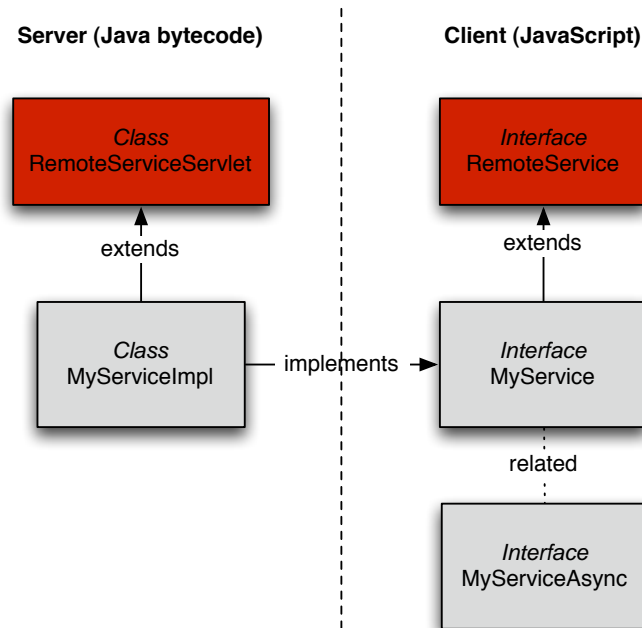


Figure 7: Classes to constitute a GWT service.

4.2.3 The FerryWeb ProcessFerryCodeService

The ProcessFerryCodeService contains the main functionality of FerryWeb, *i.e.* - as the name implies - process the Ferry code entered by the user. As an output, it creates a set of files which are then shown in the result window. In order to do so, it invokes the Ferry compiler tool chain with the user code. Since with the implementation of ImageFlow (see 4.1.3) it is not necessary to return any values to the client, ProcessFerryCodeService has only one method and that is the one to invoke the compilation.

Because different users will enter different code there is a need to deal with the management of sessions. For this, the Java API for HTTP sessions is used. FerryWeb creates for every user an individual output folder, whose location is then passed back to the client to make sure everyone sees just his compilations results.

4.3 Software architectural aspects

The basic underlying software architecture pattern used in FerryWeb is the well-known Model-View-Controller pattern, or MVC. Its main idea is to separate the data (the model) from the presentation (the view). This brings the

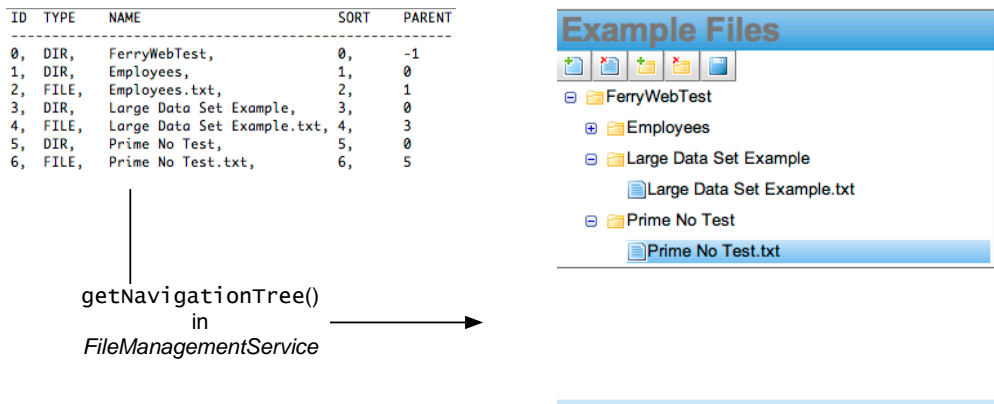


Figure 8: The representation of the server file system.

advantage of reusable components and the easy exchangeability of the view while maintaining the core functionality of an application in the model.

The components of the Model-View-Controller pattern are now explained in more detail and the basic scheme is shown in Figure 9.

Model The model represents the heart and core functionality of an application. This includes the data, its representation as well as algorithms. If data has changed, it advises its observers of the changes so that they can update the view. These observers build the exclusive connection point between the model and the view. Thus, views can be exchanged easily.

View The view takes charge of the (graphical) representation of an application's data and provides an UI. As such, it receives user input and forwards this events to the model. Generally (*i.e.* in the desktop context), when the data changes, the view is advised by the model of the specific changes.

Controller While the view only forwards events from the user, the controller has to process them and forward them to the model. Therefore, it uses the model's interface for changing data. A common feature which is implemented in the controller is the undo/redo functionality: The controller caches the incoming events (*e.g.* with a representation for commands) and all the information needed to undo or redo them. In order to do so it could be possible, that besides the view, the controller also is an observer of the model to get notice of eventual changes.

MVC in web applications As mentioned before, things in the web are a little different to classic desktop applications. As web applications grow, software architectural aspects play an important role. But not all the concepts of the desktop world are applicable to the web world without changes.

In a web application, the view runs inside the browser of the client. The method of connection is normally HTTP or HTTPS. The controller can be understood as twofold, one part at the client side sending requests, the other server-sided receiving requests and vice versa. The role of the view is also somewhat apart: Of course, the representation takes place in the browser, but it is "only" send there by the server. So the view on the server side refers to the HTML files build by the web application and send to the browser.

Another aspect is that the above introduced observer role cannot be used in web applications. Since communication via HTTP is used, the server can only send data to the client when a request was made in the first place.

MVC in FerryWeb To give an overview of the FerryWeb software architecture, we will have a first look at the Model-View-Controller pattern implemented in FerryWeb. Since FerryWeb uses the GWT RPC mechanism and thus HTTP, the scheme is slightly different to that in Figure 9. (The communication between model and view is not accomplished directly but via a callback object that is passed with the request.)

Figure 10 shows an UML representation of FerryWeb: At the top, the client-side FerryWeb classes are shown. As mentioned before, they play the role of the view. Below the controller is drawn, to which the UI passes information that will go from there to the server, which resides at the very bottom. For example, a request to compile Ferry code would start in the view (in the FerryWeb class) then go in the ProcessFerryCodeServiceAsync class and from there to the server. The way the response takes isn't depicted, but is achieved by a callback object that is passed to the server with the request.

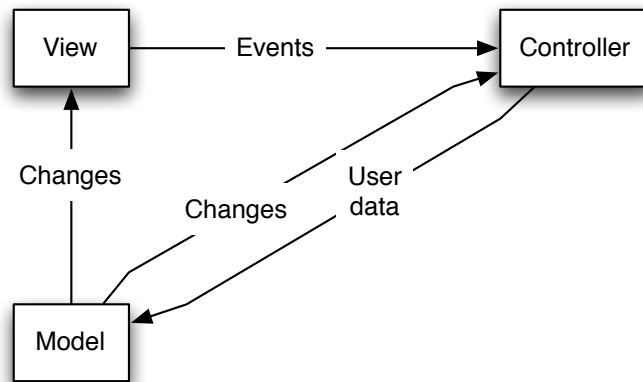


Figure 9: Model-View-Controller pattern. Basic scheme of the well-known pattern for desktop applications.

List of Figures

1	Employees table	4
2	Ferry Compilation Stages	6
3	Anatomy of a web application	7
4	Structure of a web server running Java servlets	8
5	FerryWeb user interface layout	12
6	AJAX example call	14
7	GWT services	15
8	FileTree	16
9	Model-View-Controller pattern	18
10	UML diagram of FerryWeb	20

Listings

1	Query for Employees table in Java (ODBC)	4
2	Query for Employees table in Ruby	5
3	Query for Employees table in Ferry	5

References

[1] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: database-supported program execution. In Ugur Çetintemel, Stanley B.

- Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 1063–1066. ACM, 2009.
- [2] Torsten Grust, Sherif Sakr, and Jens Teubner. Xquery on sql hosts. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 252–263. VLDB Endowment, 2004.
- [3] Simon Peyton Jones and Philip Wadler. Comprehensive comprehensions. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 61–72, New York, NY, USA, 2007. ACM.
- [4] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

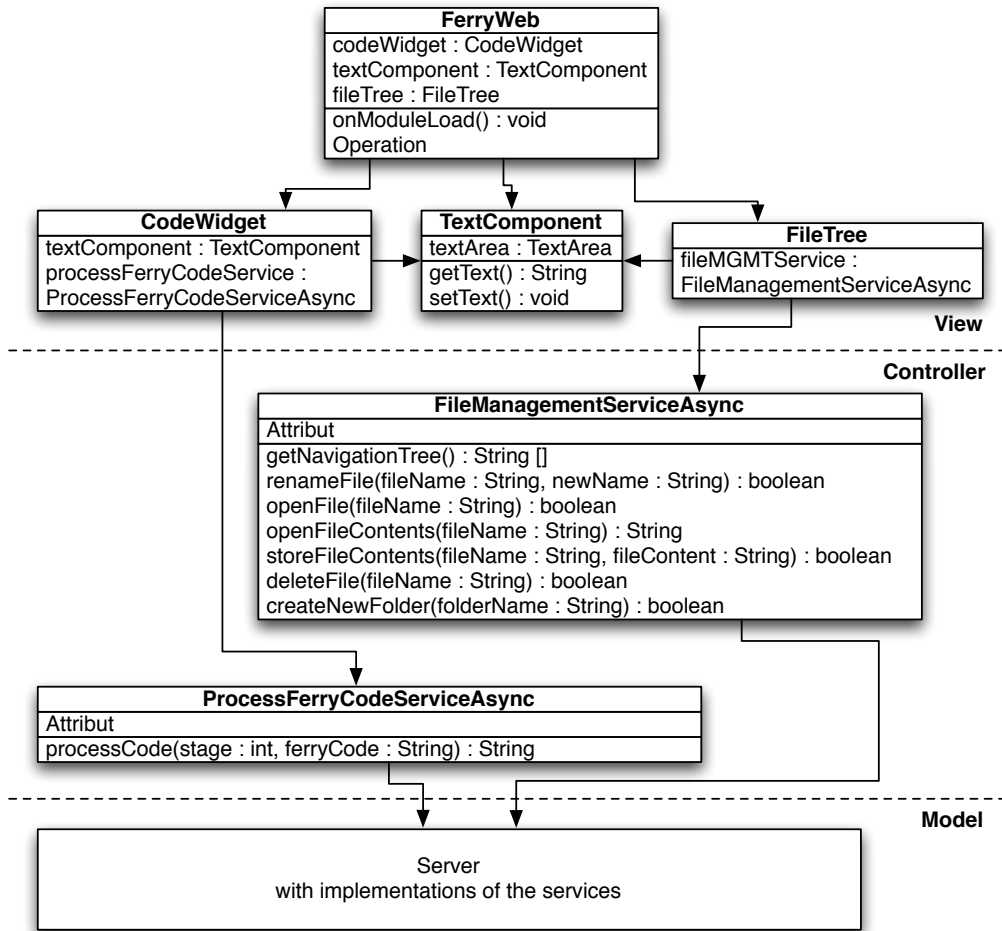


Figure 10: UML diagram of FerryWeb. Note that since this diagram is intended to show the MVC pattern in FerryWeb it does not show all the classes in FerryWeb.