A Web-based Toolkit for Collaborative Innovation

Donald Cowan
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario Canada
N2L 3G1
dcowan@csg.uwaterloo.ca

Terry Wilkinson
Computer Systems Group
University of Waterloo
Waterloo, Ontario Canada
N2L 3G1
twilkinson@csg.uwaterloo.ca

Douglas Mulholland
Computer Systems Group
University of Waterloo
Waterloo, Ontario Canada
N2L 3G1
dwm@csg.uwaterloo.ca

Paulo Alencar
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario Canada N2L 3G1
palencar@csg.uwaterloo.ca

Fred McGarry
The Centre for Community Mapping
50 Westmount Road North, Suite 206
Waterloo, Ontario Canada
N2L 2R5
mcgarry@comap.ca

Technical Report CS-2012-23

ABSTRACT

In a previous report we outlined the need for a web-based framework for collaborative innovation. In this report we describe the toolkit, meta-components and meta-processes that are needed to support such activity.

1. CATEGORIES AND SUBJECT DESCRIPTORS

H.1[**Information Systems Models and Principles**]:General;
H.4[**Information Systems Applications**]: Miscellaneous;

D.2.0[**Software Engineering**]: General

2. GENERAL TERMS

Web Science Applications

3. KEYWORDS

collaborative innovation; Web Science; asset-mapping; web-based framework; software toolkit; software meta-components, software meta-processes

4. INTRODUCTION

As part of our research in Web Science we have been exploring web-based and mobile applications to support the activities of geographic communities or communities of practice. We have labeled this type of application research collaborative innovation (CI) [2]. CI is a part of Web Science from two points of view: [1] CI embraces the use of the Web as a vast information network of people and communities; and [2] the software engineering approach envisioned for the Web is the same approach used in creating CI systems as depicted in Figure 1 [4].

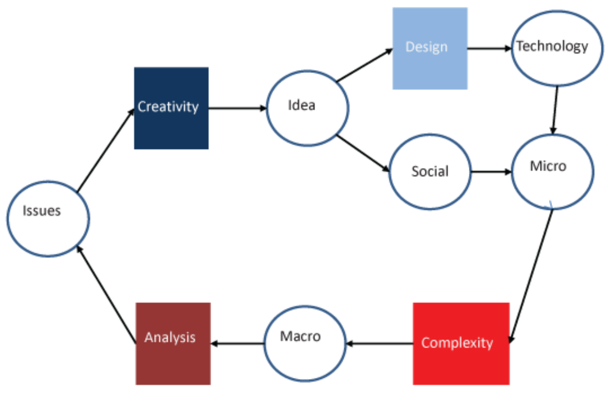


Figure 1: CI and Software Engineering for the Web

CI systems based on the Web and mobility start as a micro system and as the user base evolves, a viral effect comes into play and emergent properties arise. CI changes the way we think about problems or to quote McLuhan [6], “We shape our tools, and then our tools shape us.” The Web provides the connectivity to support CI, but how do we support the emergent needs that arise as a virtual or geographic community works toward a common understanding and solution of a problem? We need to extract the general architectural principles and interactions that occur and then look at the tools and meta-tools that are needed for CI.

Although people collaborate to supply solutions as in the “wisdom of the crowds” [8], there has to be an underlying social and information infrastructure that turns the information supplied by the "crowd" into knowledge and action. This is the purpose of these CI applications.

We have discovered that for CI to be truly effective the users need access to meta-tools and frameworks to implement tailored systems supporting CI directly rather than relying on people with in-depth knowledge of software technologies. Although we have not fully achieved this goal of usability we have made significant advances toward realizing it by making web and mobile applications easier to build and maintain.

This report describes some of our progress. The techniques are evolving constantly as we learn from our experiences and experiments. Using various versions of the software technology outlined in this paper our research team and its partners have produced over 80 web-based and mobile information systems. These span many areas such as economic development, social development, tourism, aboriginal affairs, environment, cultural heritage and population health. The partners in these projects range from non-governmental organizations, local, provincial and federal governments to the United Nations.

5. AN ILLUSTRATIVE EXAMPLE

In this section we describe a simple problem in collaborative innovation that can be used to illustrate the fundamental concepts behind a web-based meta-toolkit. Consider a community of sellers and buyers that wish to set up a web-based and mobile system of mutual benefit.

The sellers wish to notify the buyers about items for sale and a buyer wishes to be notified when anything of specific interest is offered for sale. The sellers can specify items offered for sale by type, description, location, time of sale and price. Similarly buyers provide similar specification for items they want to purchase.

The software system would notify the buyers of items of interest based on their specifications. The location of the item would be specified on a map so that the buyer can see where the item is available. If a new buyer joins the system they will be notified of both new items for sale and of any items that are still on sale. The seller has the right to withdraw or delete any item from a sale, because it is no longer available.

This example could have also introduced buyer agents that would locate the "best" deal among all articles of the same type that are for sale. The agent could have used price, reputation, warranty and location as parameters to determine what should be purchased and either buy the object or provide the buyer with a recommendation. We will not consider agents in this example except in our discussion of the meta-toolkit.

5.1 What components do we need?

Based on the brief specification provided in the previous paragraphs what type of components are needed to build the application?

5.1.1 Data model

The key component is a common data model shared by the buyer and seller. Of course parts of the data model may only be accessible to the buyer and part to the seller.

5.1.2 User interfaces

Sellers need to be able to provide data about themselves (name, address, type of business) and about the item for sale (type, description, price, duration of sale, sale location) through an input form. The seller then needs an output form to check the validity of the input. Buyers need similar facilities to describe themselves and the articles of interest. Both groups need some form of access control to ensure each seller and buyer uses the correct forms and thereby modifies the appropriate part of the database that implements the data model.

5.1.3 Behaviour

Sellers notify buyers of articles for sale and the buyers decide what items are of interest based on what is available. Thus, sellers are publishers of information and buyers are subscribers with the proviso that the buyers are not just able to look at what goes on sale in the future but can also look at anything that is still on sale. In addition a seller can withdraw or delete an item from the sale.

6. WIDE SOFTWARE ASSETS

The Web Informatics Development Environment (WIDE) toolkit and its predecessors have been under development for over 20 years. The primary purpose of this toolkit is to simplify the implementation and maintenance of web-based and mobile applications with the ultimate goal of providing many of the tools to users. Although development is continuing a number of the essential features have been determined and this section is intended to provide a description of them. The example system described in Section 5 is used to motivate many of the meta-components and meta-processes that constitute the toolkit.

The WIDE toolkit is really a meta-toolkit in that it provides meta-components or outlines that can be completed to construct a component, and meta-processes that are needed compose the components into an application. In this section we describe some of the basic meta-components that are needed to construct most web-based and mobile systems that we have encountered. In Section 7 we describe the meta-processes that are used to compose components into an application. Of course the list of such meta-components and meta-processes will never be complete as new applications will be conceived. However we can view the contents of the meta-toolkit as stabilizing over time.

6.1 Data meta-model and model

The data model and corresponding database is often difficult to implement as the data model depends on identifying the entities and relations that support the application system. Simply put, experience is a key factor in constructing data models. However, meta-components can be provided that make the data modeler's task easier.

One meta-component of course is the ability to generate or re-generate a database and database views from entity-relationship diagrams or equivalent XML descriptions. A view creates a virtual table that presents data gathered from one or more tables. Views can be used to isolate an application from changes in the underlying data model. Views to support output are straightforward whereas views where input is provided and then must be posted to a new data model can be more complex.

Since data models and the databases are often restructured as the problem space changes, there must be mechanisms to unload the data from the old database and reload it to the new version.

Good database facilities try to make these processes as transparent as possible but improving the abstraction is important when trying to provide the application user with appropriate tools.

6.1.1 *Indexing and searching*

The data must be able to be searched both by value and time. Therefore the database should support indexing and time-stamping mechanisms.

6.1.2 *Triggers*

A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database. Triggers need to be supported in the model to handle events related to changes in the content of the database. Such triggers can handle events for agents or publish-subscribe patterns.

6.2 The Human-Computer Interface

Once the data model is operational it becomes necessary to construct a human-computer interface to support the input of data or its presentation. The data is either structured, that is, searchable by data type or unstructured in that it is not easily broken into component parts. Items such as books, documents or pictures are viewed as unstructured data. Both types of data can be stored in a modern relational database.

6.2.1 *Output*

Output can consist of the following items:

- structured reports,
- documents,
- multimedia content (audio, video, pictures),
- diagrams or maps and
- statistical reporting (charts and graphs).

Any output presentation can consist of one or more of these items in various combinations. For example a document may have a structured report with embedded multimedia presentation, chart or map.

Structured reports consist of rows and columns. A document is unstructured or semi-structured text like a book where each section is identified; documents often have associated meta-data that provides information such as author, publisher, topic and date of publication. Multimedia is also unstructured and similar to documents in that it is a chunk of information usually with accompanying meta-data.

Diagrams or maps consist of a background and layers with various controls to manipulate the background and layers. The layers are geo-referenced relative to the map and often come from a database containing geo-spatial information. Details of maps are presented separately in Section 6.2.4 as they are a comprehensive form of output and input. Statistical reports and charts are graphical methods for presenting structured data and are an important form of output.

6.2.2 *Input*

Input can consist of the same items as output namely: structured data, documents, maps, diagrams, graphs and charts; or multimedia. Maps are usually delivered from a specialized spatial map-server as described in Section 6.2.4. Graphs and charts are usually generated from structured data and rarely used as input. Multimedia and diagrams are usually produced using separate tools that produce common file formats and can easily be incorporated into structured or document data through standard file extension naming or a hyperlink.

Based on these thoughts we need input forms for structured data and documents. Structured data can be handled through a meta-form that allows the construction of fields that accept data and store the data in the related database. Documents can be constructed using a wiki-like tool that supports text and hyper-linking but can also incorporate the other types of information (maps, diagrams, graphs and charts; or multimedia) as well.

Of course the input mechanism must support bulk upload of data and documents. This topic as related to data was covered in 6.1. With respect to documents the system must be able to upload many of the common document formats (doc, pdf, html, ...) and allow them to be stored and indexed. It should be possible to choose a common format such as text or HTML to allow the documents to be easily indexed for later searching. Documents should also be time-stamped to allow time-dependent searching. The Windows IFilter [10] approach is a valuable extension that provides indexing support for various document file formats.

6.2.3 *Interactive interfaces - combined output/input*

The interface should also be able to support interaction. Fields and documents from a database should support editing. In addition, one should be able to ask what-if questions with respect to data. Views of data can be exported as comma-separated values (csv) to a spreadsheet application that would support changing data to see the effects. The changed data might not be used to update the database, but could be stored in an auxiliary database that could be used to drive the interface.

6.2.4 *Maps and Diagrams*

The map interface or map client should be interactive and contain a map and a bounding box with controls. The map is delivered from a map-server of properly geo-referenced base maps and features such as roads and water courses or lakes.

The base maps are delivered as tiles while any features are overlaid on the map as translucent tiles, vectors or points represented as dots or other symbols. The maps can represent any spatial or map-based concept including thematic maps such as ones showing environmental or demographic data, roadmaps or even floor plans. Combinations of maps or map layers can be displayed such as a road map with a superimposed thematic map. Vector-based maps can also be displayed where required.

Map-client controls in conjunction with the map server support zoom-in or zoom-out functionality and positioning over areas of interest. Positioning can be performed by scrolling the map with a pointing device such as a mouse or finger. When connected to a database or directory of geo-referenced information the controls on the interactive map can be used to:

1. Search for geo-referenced data in multiple databases. The map-area searches are defined by the frame around the map or a shape including a circle, rectangle or general polygon. The search function displays the results as the search frame is defined.
2. Display the results of the search as a point, circle or polygonal shape on the map depending on the result of the search. For example, a building would normally be displayed as a point while a park would be displayed as a polygon.
3. Interact with the location and shape of a geo-referenced object by re-locating it or changing its shape.
4. Interact with a geo-referenced object to cause more information about the object to appear on or next to the map.
5. Interact with a geo-referenced object by completing a form associated with the object that creates or adds to the information about the object in the geo-referenced databases.
6. Display layers of information related to different datasets. By supporting map layers and their associated data it is possible to show how different groups of data are related geographically.

The map client should accept maps from different map servers such as those of Google, bing or OpenStreetMap (OSM), thus providing the same functionality no matter the source of the maps.

The map client and server can support interchange of map data with traditional geographic information systems (GIS) software to support the mapping functionality. The server and client should support appropriate sections of the current version of the Open Geospatial Consortium (OGC) [7] standards. Supporting these standards allows direct communication between applications based on the interactive map client and connected databases and any GIS incorporating the OGC standard. Thus it is possible to communicate with GIS systems used by community governments. The map server and map client should support the import and export of shapefiles [11] as another common interchange format between a GIS and the map client.

6.3 Interaction modes

In developing applications that can operate on the desktop, tablet and smartphone, one needs to consider the screen layout and the interaction modes available to the user. Fortunately the interaction modes are very similar in that one can use a pointing device such as a mouse or finger and can scroll over multiple windows and apps.

The screen area available to the different classes of devices does govern the presentation of information. For example, when indicating a point or object on a map on a desktop machine or reasonable size tablet with a view to querying associated information, one can continue to show the map and the results of the query. However the screen of a smartphone is too small to allow this type of presentation. In that case the results of the query should replace the map with the ability to show the map instead of the query result. Smartphone and tablet native application development tools support this differentiation.

6.4 Notification and Publish-Subscribe

A critical function in almost any computer system is the ability to notify the user or other components when there is a change of state. For example, the clock is ticking over indicating that an appointment on the user's calendar is about to occur, or a document has just been published and all subscribers need to be notified in case they are interested in it. The type of "document" that can be published can vary widely. For example, it can be an event such as an appointment, a request, a document or a message. Based on these comments it is clear that web-based and mobile systems require a form of publish-subscribe pattern [9].

We first describe the ideas behind the basic publish-subscribe pattern and show why it is not adequate to meet all the needs of a general web-based information system. In the basic publish-subscribe pattern an author creates a document and sends it to a publisher who distributes (publishes) the document to unknown subscribers. The subscribers are notified that there are documents.

The subscribers then retrieve the documents based on some criteria or message filters. Criteria could be content (news mentioning the European Union), type (sports or cooking), context (time of day or location of subscriber). This approach is similar to putting a newspaper or magazine on a newsstand where the publisher does not know the identity of the subscriber and the subscriber chooses to acquire the publication based on the subscriber's interests. A typical publish-subscribe pattern is shown in Figure 2 where the message filters capture the criteria under which the subscriber obtains a document.

There are concerns related to this form of publish-subscribe pattern. The publisher may need to have a direct or indirect means of knowing who has the documents. Some publishers may require a recall or update in the event that a defect is discovered in a published document or data set, or if a new version of publication is made available. A document may only be published if certain contextual conditions are met or the publisher may want to put restrictions on the document such as only publish to subscribers on a restricted list. A subscriber may want to edit and re-publish a document, in

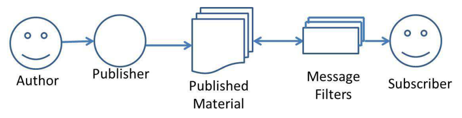


Figure 2: Publish-Subscribe

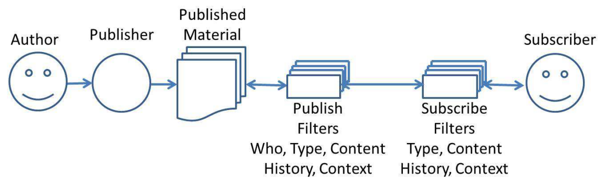


Figure 3: Mediated Publish-Subscribe

effect a subscriber may want to switch roles and become a publisher.

The primary change to the publish-subscribe pattern is shown in Figure 3 where publish filters have been added to the pattern. After the document is published it passes through a number of publish filters. Who are the recipients?; What type is the document (sports, local news)?; What is its content (indexed)?; What is its history (time of publication, recipients etc.)?; What is its context (time/place to be published)? Note that the publisher may or may not know know the identity of the subscribers directly, but they are at least known to the system because we may want to replace or recall a document (bad data). This situation is similar to a paid subscription to a magazine or newspaper where the delivery agent knows the identity of the subscriber.

The subscriber also has extra filters based on history in case the subscriber wants to recover documents that have been lost. Context could be time or place or more complex situations as in “I only want documents from my boss between 9am and 5pm.” Whenever a subscriber joins a publish-subscribe situation the subscriber must be able to specify whether documents published in the past should also be made available.

6.4.1 *Collaboration and Mediated Social Networks*

Although social networks connect individuals, there is a need for a social network for groups. A mediated social network is such an object and can be viewed as a group of individuals seated around a real or virtual conference room table collectively working on collaborative tasks. The group and its set of purposes form around a geographic community or community of interest and the group may break into sub-groups to divide the tasks and make them more manageable. The assets and the value created from them are held and controlled by the group. The group may appoint mediators to manage the composition of the group and sub-groups, to assign tasks, and manage and control the group’s assets and derived value. In contrast a social network is focused on the individual rather than the group. The individual controls what can be seen and shared, and ownership of assets and derived value belong to the individual. The group may collaborate synchronously or asynchronously.

Such a social network can use tools already described for the human interface including maps, input forms, reports, text, video, pictures, audio, wikis, blogs, and databases. Tags, social bookmarks, and other social networking tools can help bring order to the avalanche of information that is involved in forming a creative network and managing the output from the collaboration.

One key component of a mediated social network is the publish-subscribe pattern where documents and events are published to members of the network who then subscribe to them. Role reversal also occurs in that publishers become subscribers and vice-versa. Asynchronous collaboration occurs when an artifact is published and then consumed at some unspecified time in the future, although time limits may apply as in a sale. Synchronous collaboration occurs when an artifact is published and is immediately seen by the subscriber. There may be other conditions as well, including the subscriber becoming a publisher and responding within a specific time frame.

6.4.2 *Agents*

Software agents may be characterized as reactive, persistent, autonomous and social. Reactive means that a software agent acts upon the occurrence of some event. In other words an agent can be characterized as a subscriber in a publish-subscribe context.

The agents then have many other properties primarily related to their social aspects. Thus, patterns for agents can evolve.

6.5 Access Control Rules

Collaboration requires that a group forms around an idea or situation with the objective of working together. The group is self-limiting by expertise or interest although it may grow or change in composition as the collaboration forms and changes. By its nature collaboration is not completely open. Therefore there must be moderators who manage the group composition and delegate authority to members of the group related to responsibilities. These moderators must be given a set of tools not only to enable the collaboration but to admit participants with responsibilities, so-called transactional access control rules. Such access controls can be role-based [1], attribute-based [3] or use other confidentially models [5]. Participants could be allowed a subset of operations on the asset base such as read, write, update, or write/update with history log. These same participants could also be limited in the collaborative tools that they can use or the portion of the asset base they can see through an interface such as a map. For example, experts on some scientific topic could use a wiki or blog to discuss issues around protocols for identification and remediation, whereas laymen could read the content of the wiki or blog, but could not offer an expert opinion.

7. THE WIDE TOOLKIT

The Web Informatics Development Environment (WIDE) toolkit has been under development for over 20 years and has gone through many versions as our ideas have been refined. The primary purpose of this toolkit is the holy grail of software development namely to make web-based and mobile applications easier to build and maintain and to make application development tools more accessible to domain experts or users so that they can build and modify their own applications. Although development is continuing, many essential features have been identified and this section is intended to describe some of them and one approach to assembling components. The example system described in Section 5 has been used in Section 6is used to motivate many of the components that constitute the toolkit.

In the current version of WIDE under development we divide an application into 2 parts. One part specifies what an application is to do and the other part specifies how it is to do it.

The first part is expressed in WIDE by a nested set of building blocks. Each block specifies a component of the application, and the nesting specifies a relationship between those components. The second part is the underlying implementation of those blocks and is expressed in a conventional programming language. However, a developer does not have to access the second part unless a new building block is needed.

WIDE has a built-in set of core building blocks, some of which specify user-interface components while others, specify components of the data model. Among these are blocks that variously support SQL databases, mapping via map clients and chart plotting. A programmer can add new or custom blocks to WIDE by writing them in a supported language and putting them in the appropriate extension libraries. They are then available for use in other applications.

The current implementation of WIDE targets Javascript, HTML5 and CSS3 and as a result, only operates in modern browsers and mobile hybrid apps that support that technology. However, the concepts can be applied to web-based applications in general where the target languages differ from those just mentioned.

7.1 Introduction to WIDE

In the current version of WIDE, an application is viewed as a nested collection of blocks, where each block has a parent and possibly siblings and children. Blocks often have one or more attributes. WIDE is designed to be extensible, meaning that new functionality can be added to the language either by:

- defining new blocks which would then be available for use in other applications, or
- defining new blocks available only to a specific application.

You might visualize such a set of blocks as shown in Figure 4. This Figure might represent the layout of some visible blocks, but it is important to realize that the nesting represents the relationships between the blocks, rather than their physical layout in a user-interface (UI). Not all blocks in WIDE are UI blocks. For example, a block can represent a line in a report or a database reference. Blocks can access information described in blocks in which they are nested. An application is one block that contains the individual page descriptions and accompanying data descriptions to generate those pages.

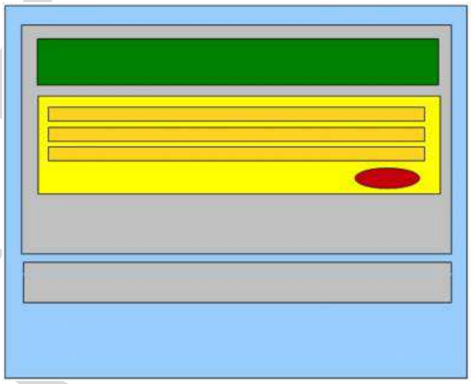
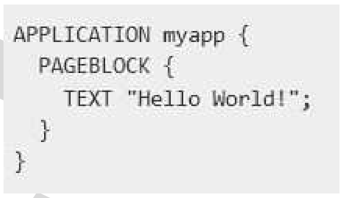


Figure 4: Overview of WIDE



```
APPLICATION myapp {
  PAGEBLOCK {
    TEXT "Hello World!";
  }
}
```

Figure 5: Hello World Application

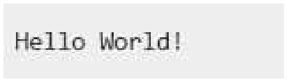
7.1.1 *Two Examples*

Here is the ubiquitous “Hello World!” application. The code is shown in Figure 5. This simple application block consists of a single PAGEBLOCK block within an APPLICATION block. The APPLICATION block represents the entire application while the PAGEBLOCK represents one viewable, scrollable page. The PAGEBLOCK block has a TEXT attribute with the value “Hello World!”.

To turn this simple definition into an executing application, we compile it using the WIDE toolkit and then deploy the compiled code either for execution in a modern browser such as Google Chrome, or on a mobile device such as Apple, Android or BlackBerry tablet or smartphone.

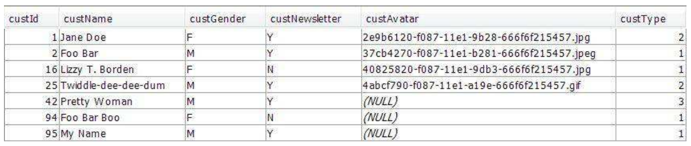
With default styling information, the application output will appear as shown in Figure 6.

The second example is more extensive and illustrates some of the core blocks and attributes in WIDE. It simply lists the contents of a server database table. Assume a server-side or remote database with a table named DBA.Customer



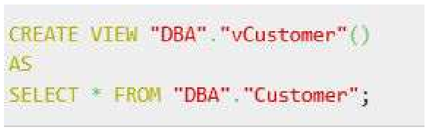
```
Hello World!
```

Figure 6: Hello World Application



custId	custName	custGender	custNewsletter	custAvatar	custType
1	Jane Doe	F	Y	2e9b6120-f087-11e1-9b28-666f6f215457.jpg	2
2	Foo Bar	M	Y	37cb4270-f087-11e1-b281-666f6f215457.jpeg	1
16	Lizzy T. Borden	F	N	40825820-f087-11e1-9db3-666f6f215457.jpg	1
25	Twiddle-dee-dee-dum	M	Y	4abcf790-f087-11e1-a19e-666f6f215457.gif	2
42	Pretty Woman	M	Y	*(NULL)*	3
94	Foo Bar Boo	F	N	*(NULL)*	1
95	My Name	M	Y	*(NULL)*	1

Figure 7: A Simple Example



```
CREATE VIEW "DBA"."vCustomer"()
AS
SELECT * FROM "DBA"."Customer";
```

Figure 8: The SQL View of the database

as shown in Figure 7.

The SQL View of this simple database is defined by the SQL statements shown in Figure 8 that is stored in a file. An appropriate configuration file entry links the file name to the database name "db00" in Figure 9.

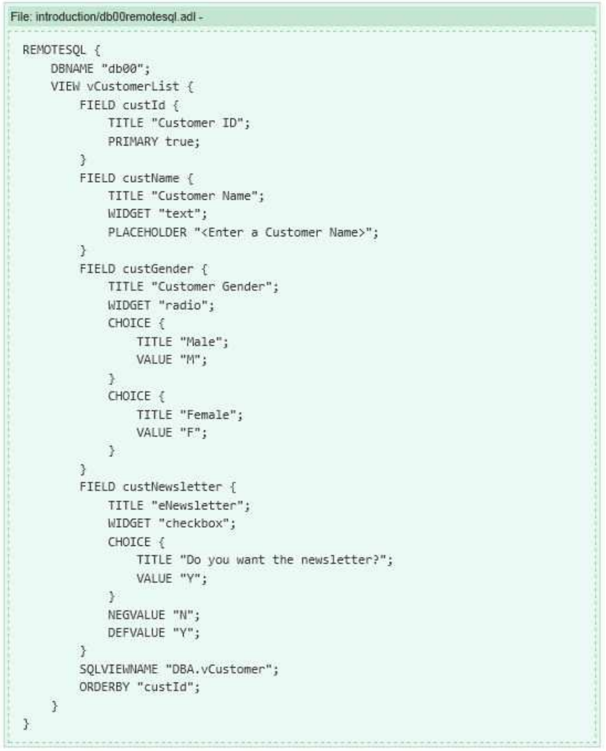
The WIDE source code is shown here in two parts. Figure 9 shows the part that describes the interface to DBA.vCustomer in the database. The main block is REMOTESQL with an attribute for the database name (DBNAME) and a block (VIEW) describing a table/view in that database. The VIEW block has an attribute referencing the database table (SQLVIEWNAME) at the bottom of the block in Figure 9, and FIELD blocks for each of the fields we want to access. Each FIELD block has appropriate meta-data attributes such as TITLE, PRIMARY, WIDGET, PLACEHOLDER, and CHOICE.

Figure 10 shows the code for the application that displays the database table. It consists of an APPLICATION block, a couple of #INCLUDE directives, and a single PAGEBLOCK (of course, most applications would have more than one PAGEBLOCK). The first #INCLUDE directive needs to be there in most WIDE applications and references a file containing locale-specific internal messages. The second #INCLUDE directive references the file defined in Figure 9.

Since this application simply displays the content of the table, our PAGEBLOCK contains a DATABLOCK with a DATAVIEW attribute referencing the VIEW in the REMOTESQL block Figure 9.

This technique, separating the definition of the database table from the actual display of its contents, might seem unnecessary here, but it does make things easier in applications more complex than this one.

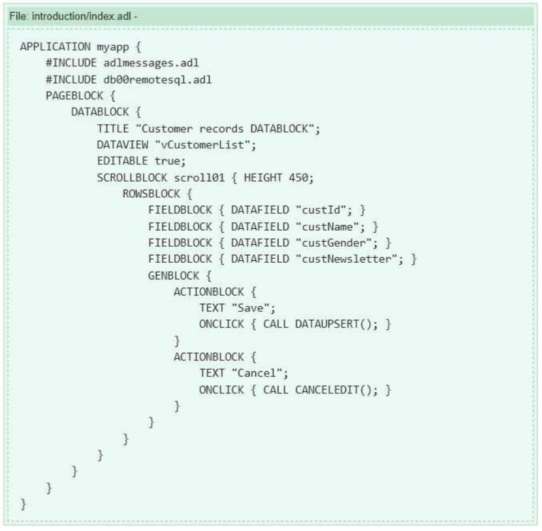
Inside the DATABLOCK is a SCROLLBLOCK and inside that, a ROWSBLOCK. The ROWSBLOCK describes how each row of data is to be composed. Each row will have 5 fields (specified by a FIELDBLOCK block and associated DATAFIELD attribute), followed by action blocks for "Save" and "Cancel" operations. Clicking on an ACTIONBLOCK will invoke the associated DATAUPSERT or CANCELEDIT built-in action.



File: introduction/db00remotesql.adl -

```
REMOTESQL {
    DBNAME "db00";
    VIEW vCustomerList {
        FIELD custId {
            TITLE "Customer ID";
            PRIMARY true;
        }
        FIELD custName {
            TITLE "Customer Name";
            WIDGET "text";
            PLACEHOLDER "<Enter a Customer Name>";
        }
        FIELD custGender {
            TITLE "Customer Gender";
            WIDGET "radio";
            CHOICE {
                TITLE "Male";
                VALUE "M";
            }
            CHOICE {
                TITLE "Female";
                VALUE "F";
            }
        }
        FIELD custNewsletter {
            TITLE "eNewsletter";
            WIDGET "checkbox";
            CHOICE {
                TITLE "Do you want the newsletter?";
                VALUE "Y";
            }
            NEGVALUE "N";
            DEFVALUE "Y";
        }
        SQLVIEWNAME "DBA.vCustomer";
        ORDERBY "custId";
    }
}
```

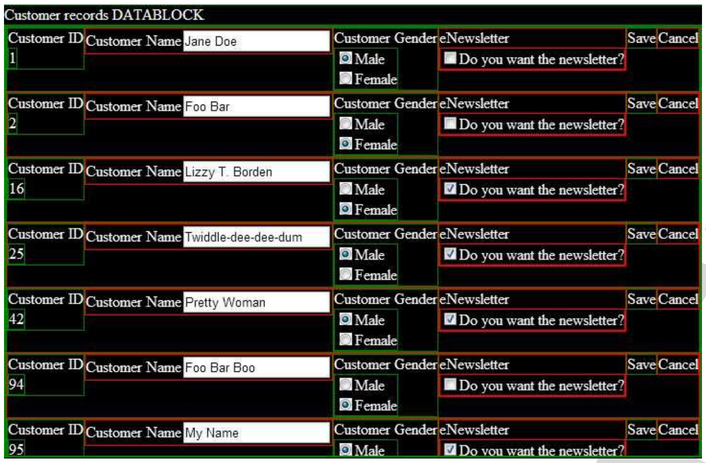
Figure 9: Interface to the Database



File: introduction/index.adl -

```
APPLICATION myapp {
    #INCLUDE adlmessages.adl
    #INCLUDE db00remotesql.adl
    PAGEBLOCK {
        DATABLOCK {
            TITLE "Customer records DATABLOCK";
            DATAVIEW "vCustomerList";
            EDITABLE true;
            SCROLLBLOCK scroll01 { HEIGHT 450;
                ROWSBLOCK {
                    FIELDBLOCK { DATAFIELD "custId"; }
                    FIELDBLOCK { DATAFIELD "custName"; }
                    FIELDBLOCK { DATAFIELD "custGender"; }
                    FIELDBLOCK { DATAFIELD "custNewsletter"; }
                    GENBLOCK {
                        ACTIONBLOCK {
                            TEXT "Save";
                            ONCLICK { CALL DATAUPSERT(); }
                        }
                        ACTIONBLOCK {
                            TEXT "Cancel";
                            ONCLICK { CALL CANCELEDIT(); }
                        }
                    }
                }
            }
        }
    }
}
```

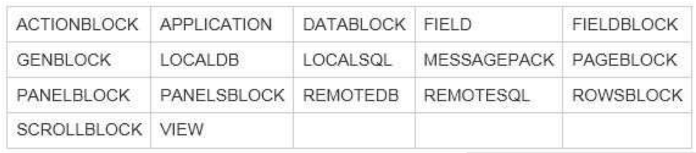
Figure 10: Application Code



Customer records DATABLOCK

Customer ID	Customer Name	Customer Gender	eNewsletter		
1	Jane Doe	● Male ☐ Female	☐ Do you want the newsletter?	Save	Cancel
2	Foo Bar	☐ Male ● Female	☐ Do you want the newsletter?	Save	Cancel
16	Lizzy T. Borden	☐ Male ● Female	✓ Do you want the newsletter?	Save	Cancel
25	Twiddle-dee-dee-dum	● Male ☐ Female	✓ Do you want the newsletter?	Save	Cancel
42	Pretty Woman	● Male ☐ Female	✓ Do you want the newsletter?	Save	Cancel
94	Foo Bar Boo	☐ Male ● Female	☐ Do you want the newsletter?	Save	Cancel
95	My Name	● Male	✓ Do you want the newsletter?	Save	Cancel

Figure 11: Output from Example



ACTIONBLOCK	APPLICATION	DATABLOCK	FIELD	FIELDBLOCK
GENBLOCK	LOCALDB	LOCALSQL	MESSAGEPACK	PAGEBLOCK
PANELBLOCK	PANELSBLOCK	REMOTEDB	REMOTESQL	ROWSBLOCK
SCROLLBLOCK	VIEW			

Figure 12: WIDE Blocks

With some default styling, the resulting window might look like Figure 11.

7.2 WIDE System Overview

WIDE is an extensible system. This means that we expect the blocks, attributes, events, actions and values available to the developer to evolve over time. It also means that specialized blocks can be defined for use in one application and not be present in other applications.

7.2.1 *Core Language Components*

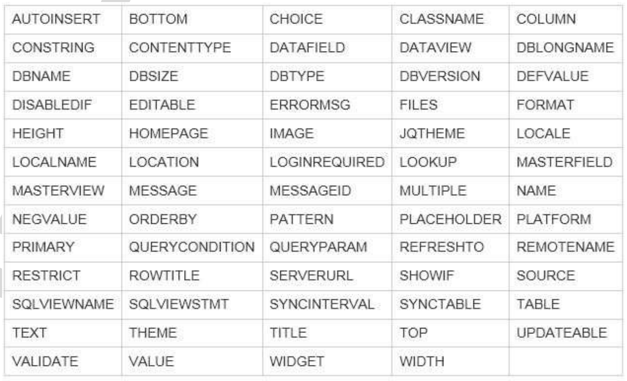
The current implementation of WIDE includes the following types of core blocks, attributes, events, actions and values as shown in Figures Figures 12 through 15. By their name most of them should be self-explanatory and the use of some of them has been described in Section 7.1.1

7.2.2 *WIDE Data model*

The WIDE data model describes various views of the underlying database, separate from the user interface of the application. The REMOTESQL, VIEW and FIELD blocks describe the underlying database, while the DATABLOCK, ROWSBLOCK and FIELDBLOCK blocks describe the data used in the user-interface. We now consider each in turn.

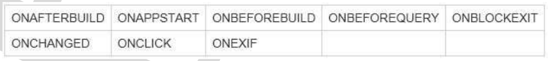
7.2.3 *Remote Data model*

The REMOTESQL block describes a database that resides on the server. Views onto the tables of the database are described by the VIEW and FIELD blocks. In practice, the VIEW block usually refers to an SQL VIEW. AN example of the REMOTESQL block is shown in Figure 9



AUTOINSERT	BOTTOM	CHOICE	CLASSNAME	COLUMN
CONSTRING	CONTENTTYPE	DATAFIELD	DATAVIEW	DBLONGNAME
DBNAME	DBSIZE	DBTYPE	DBVERSION	DEFVALUE
DISABLEDIF	EDITABLE	ERRORMSG	FILES	FORMAT
HEIGHT	HOMEPAGE	IMAGE	JQTHEME	LOCALE
LOCALNAME	LOCATION	LOGINREQUIRED	LOOKUP	MASTERFIELD
MASTERVIEW	MESSAGE	MESSAGEID	MULTIPLE	NAME
NEGVALUE	ORDERBY	PATTERN	PLACEHOLDER	PLATFORM
PRIMARY	QUERYCONDITION	QUERYPARAM	REFRESHTO	REMOTENAME
RESTRICT	ROWTITLE	SERVERURL	SHOWIF	SOURCE
SQLVIEWNAME	SQLVIEWSTMT	SYNCINTERVAL	SYNCTABLE	TABLE
TEXT	THEME	TITLE	TOP	UPDATEABLE
VALIDATE	VALUE	WIDGET	WIDTH	

Figure 13: WIDE Attributes



ONAFTERBUILD	ONAPPSTART	ONBEFOREBUILD	ONBEFOREQUERY	ONBLOCKEXIT
ONCHANGED	ONCLICK	ONEXIF		

Figure 14: WIDE Events

7.2.4 *DATABLOCK and result sets*

The DATABLOCK block describes a result-set, that is, the rows resulting from a query executed on a VIEW block shown in Section 7.2.3. Enclosed ROWSBLOCK and FIELDBLOCK blocks describe how that result-set is displayed. An example is in Figure 10.

7.2.5 *Displaying data values*

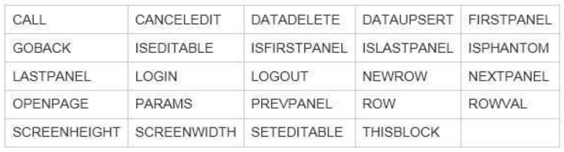
Various attributes related to displaying database values in the UI can be specified in either the FIELDBLOCK or FIELD blocks.

In Figure 9 we have defined certain attributes in the FIELD block. This allows a degree of isolation where things such as whether to use radio-buttons or select drop-downs can be defined once and used consistently throughout the application. Most of those attributes could instead have been placed in the FIELDBLOCK blocks in Figure 10.

7.2.6 *Foreign-key relationships*

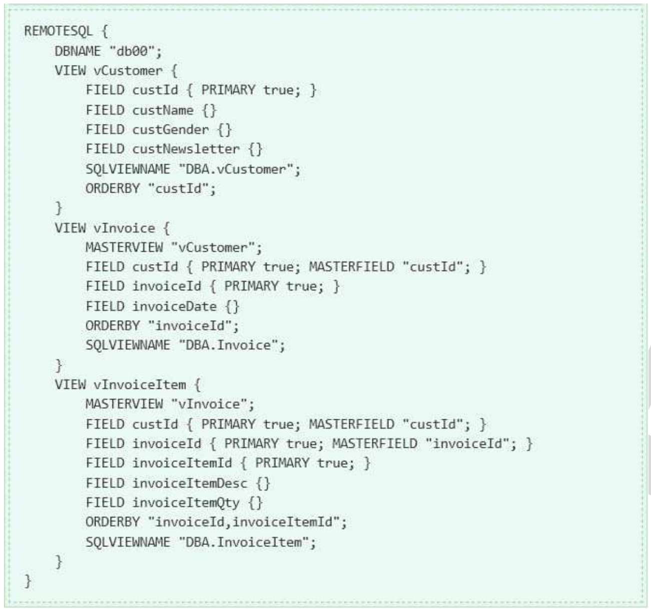
WIDE has a data model component that supports the foreign-key relationships between tables of a relational database. It does this using the VIEW block and MASTERVIEW attribute, along with the FIELD block and PRIMARY and MASTERFIELD attributes.

We consider one table to be the starting point or top view, and then we use foreign-key relationships to move to lower



CALL	CANCELEDIT	DATADELETE	DATAUPSERT	FIRSTPANEL
GOBACK	ISEDITABLE	ISFIRSTPANEL	ISLASTPANEL	ISPHANTOM
LASTPANEL	LOGIN	LOGOUT	NEWROW	NEXTPANEL
OPENPAGE	PARAMS	PREVPANEL	ROW	ROWVAL
SCREENHEIGHT	SCREENWIDTH	SETEDITABLE	THISBLOCK	

Figure 15: WIDE Actions and Values



```
REMOTESQL {
    DBNAME "db00";
    VIEW vCustomer {
        FIELD custId { PRIMARY true; }
        FIELD custName {}
        FIELD custGender {}
        FIELD custNewsletter {}
        SQLVIEWNAME "DBA.vCustomer";
        ORDERBY "custId";
    }
    VIEW vInvoice {
        MASTERVIEW "vCustomer";
        FIELD custId { PRIMARY true; MASTERFIELD "custId"; }
        FIELD invoiceId { PRIMARY true; }
        FIELD invoiceDate {}
        ORDERBY "invoiceId";
        SQLVIEWNAME "DBA.Invoice";
    }
    VIEW vInvoiceItem {
        MASTERVIEW "vInvoice";
        FIELD custId { PRIMARY true; MASTERFIELD "custId"; }
        FIELD invoiceId { PRIMARY true; MASTERFIELD "invoiceId"; }
        FIELD invoiceItemId { PRIMARY true; }
        FIELD invoiceItemDesc {}
        FIELD invoiceItemQty {}
        ORDERBY "invoiceId,invoiceItemId";
        SQLVIEWNAME "DBA.InvoiceItem";
    }
}
```

Figure 16: WIDE Customer List Example

levels of detail.

Consider the following example of a list of customers, their invoices and invoice items:

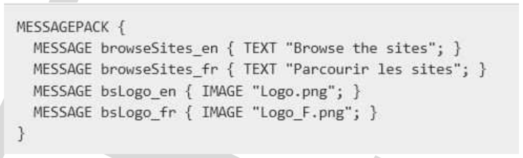
You can see that the following relationships exist:

- the VIEW vCustomer in this case is the top view.
  - the FIELD custId is its primary key
- the VIEW vInvoice refers to vCustomer as its MASTERVIEW
  - the FIELD custId refers to the MASTERFIELD custId in the MASTERVIEW vCustomer
  - the FIELDs custId and invoiceId make up its the primary key
- the VIEW vInvoiceItem refers to vInvoice as its MASTERVIEW
  - the FIELDs custId and invoiceId refer to MASTERFIELDs custId and invoiceId in the MASTERVIEW vInvoice
  - the FIELDs custId, invoiceId and invoiceItemId make up its primary key

7.2.7 Styling and Layout

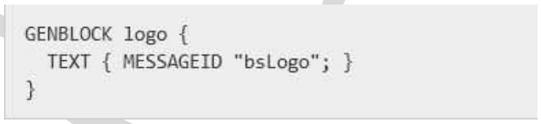
The present implementation of the WIDE language includes only minor features to support the styling and layout of the user-interface (UI).

Currently, one can select a theme from a set of defined themes (essentially pre-written CSS files), and then extend it with additional CSS code provided by the developer.



```
MESSAGEPACK {
  MESSAGE browseSites_en { TEXT "Browse the sites"; }
  MESSAGE browseSites_fr { TEXT "Parcourir les sites"; }
  MESSAGE bsLogo_en { IMAGE "Logo.png"; }
  MESSAGE bsLogo_fr { IMAGE "Logo_F.png"; }
}
```

Figure 17: WIDE Message Example



```
GENBLOCK logo {
  TEXT { MESSAGEID "bsLogo"; }
}
```

Figure 18: WIDE Message Example continued

7.2.8 Mobile devices

Mobile devices come in a variety of screen sizes and densities and WIDE supports this situation with the PLATFORM attribute. Using this value, WIDE will dynamically select appropriate CSS files and screen height and width values at run-time allowing the application to adapt to the platform on which it is running.

7.2.9 Localization

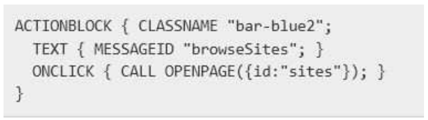
WIDE supports localization or internationalization by allowing the developer to define the text messages used in the application in a separate set of blocks and then reference those messages from the TEXT attribute.

You define messages as a MESSAGE in the MESSAGEPACK block and then reference it using MESSAGEID in TEXT and IMAGE blocks. The application can then set an internal value that represents the desired locale (usually one of the ISO country codes) and the appropriate messages and images will be used by WIDE.

Every WIDE application should include the directive #INCLUDE adlmessages.adl. This file defines the messages used internally by the WIDE core modules. It can be overridden with an application-specific set of messages if desired.

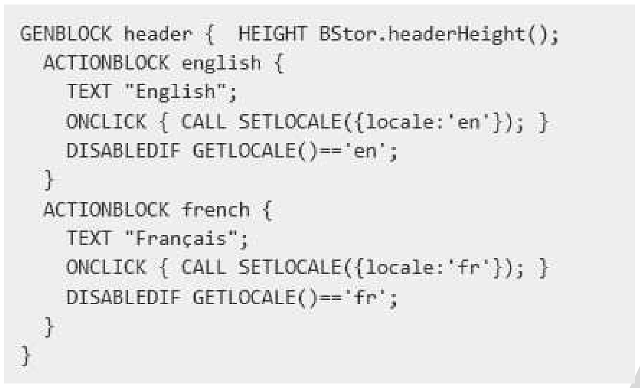
For example, you could define some messages as shown in Figure 17 and then use them with code as in Figures 18 or 19.

Depending on the internal value set for locale, either the French or the English version of each message would be rendered. That value could be managed with code as in Figure 20



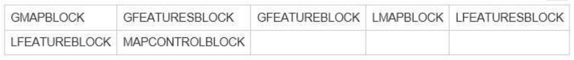
```
ACTIONBLOCK { CLASSNAME "bar-blue2";
  TEXT { MESSAGEID "browseSites"; }
  ONCLICK { CALL OPENPAGE({id:"sites"}); }
}
```

Figure 19: WIDE Message Example continued



```
GENBLOCK header {  HEIGHT BStor.headerHeight();
  ACTIONBLOCK english {
    TEXT "English";
    ONCLICK { CALL SETLOCALE({locale:'en'}); }
    DISABLEDIF GETLOCALE()=='en';
  }
  ACTIONBLOCK french {
    TEXT "Français";
    ONCLICK { CALL SETLOCALE({locale:'fr'}); }
    DISABLEDIF GETLOCALE()=='fr';
  }
}
```

Figure 20: WIDE Multi-lingual Example



GMAPBLOCK	GFEATURESBLOCK	GFEATUREBLOCK	LMAPBLOCK	LFEATURESBLOCK
LFEATUREBLOCK	MAPCONTROLBLOCK			

Figure 21: WIDE Map Blocks

7.2.10 Mapping Module

There are WIDE modules which support mapping. They have the blocks, attributes, events and actions as shown in Figures 21 through 23.

Figure 24 is a block fragment that displays a list of points on a Google map. Currently there are mapblocks for each type of map. Each map is retrieved and presented based on some default values. There is a GMAPBLOCK that defines the map, and a nested DATABLOCK that selects the point data from a DATAVIEW. The GFEATURESBLOCK and GFEATUREBLOCK specify how the data is to be presented.

7.2.11 Charting module

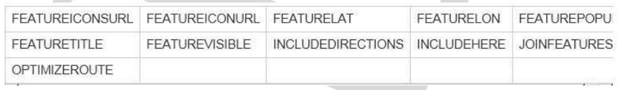
WIDE has a charting module which can plot data on a chart in various ways. They have the Block and Attributes as shown in Figure 25 and 26.

Figure 27 is a block fragment that shows a simple list of points on a chart. There is CHARTBLOCK that defines the chart and a nested DATABLOCK that selects data from a DATAVIEW. The CHARTSERIES block specifies the data to be charted.

The output might then look like Figure 28.

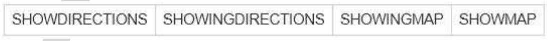
8. CURRENT IMPLEMENTATION

WIDE is currently implemented to generate HTML5 content which is displayed in a web browser such as Google



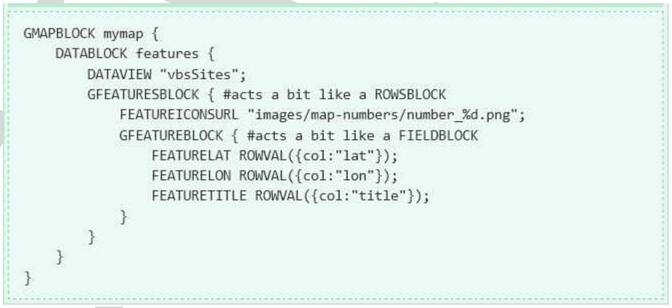
FEATUREICONSURL	FEATUREICONURL	FEATURELAT	FEATURELON	FEATUREPOPU
FEATURETITLE	FEATUREVISIBLE	INCLUDEDIRECTIONS	INCLUDEHERE	JOINFEATURES
OPTIMIZEROUTE				

Figure 22: WIDE Map Block Attributes



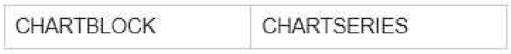
SHOWDIRECTIONS	SHOWINGDIRECTIONS	SHOWINGMAP	SHOWMAP

Figure 23: WIDE Map Block Events



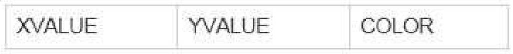
```
GMAPBLOCK mymap {
    DATABLOCK features {
        DATAVIEW "vbsSites";
        GFEATURESBLOCK { #acts a bit like a ROWSBLOCK
            FEATUREICONSURL "images/map-numbers/number_%d.png";
            GFEATUREBLOCK { #acts a bit like a FIELDBLOCK
                FEATURELAT ROWVAL({col:"lat"});
                FEATURELON ROWVAL({col:"lon"});
                FEATURETITLE ROWVAL({col:"title"});
            }
        }
    }
}
```

Figure 24: Map Block showing a list of points on a Google Map



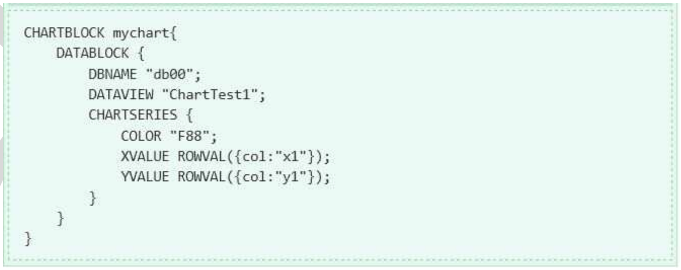
CHARTBLOCK	CHARTSERIES

Figure 25: A WIDE Chart Block



XVALUE	YVALUE	COLOR

Figure 26: WIDE Chart Block Attributes



```
CHARTBLOCK mychart{
    DATABLOCK {
        DBNAME "db00";
        DATAVIEW "ChartTest1";
        CHARTSERIES {
            COLOR "F88";
            XVALUE ROWVAL({col:"x1"});
            YVALUE ROWVAL({col:"y1"});
        }
    }
}
```

Figure 27: WIDE Chart Example

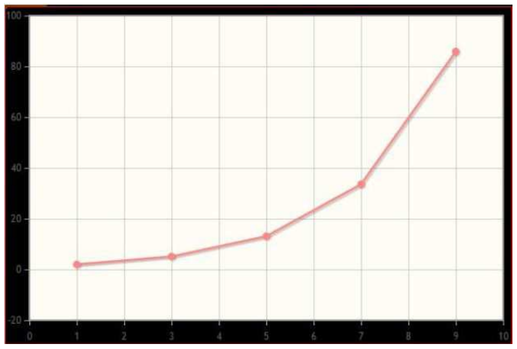


Figure 28: Sample Chart produced from Chart Example

Chrome, or some other web-view container such as is found on Android, Apple and Blackberry smartphones and tablets.

There are 3 fundamental components to the system, described below and illustrated in Figure 29.

8.1 Development-component

This is where the application definition code is translated into HTML5, CSS, Javascript and PHP.

The main pieces are:

1. WIDE Compiler (written in PHP)
2. the application code written in WIDE
3. any custom application blocks written in PHP and/or Javascript

8.2 Server-component

This server-side code supports the remote database requirements of a WIDE application.

The main pieces are:

1. WIDE Server/Database modules (written in PHP)
2. Application database (Sybase SQL Anywhere)
3. Application-specific modules (written in PHP)

8.3 Client-component

This component is the compiled version of the application. Most of this code is generated by the compiler (see development-component).

The main pieces are:

1. Application-specific code (generated from the WIDE code)
2. WIDE Run-time modules (core Javascript code provided by WIDE)
3. Any necessary application-specific code written by the developer (Javascript).
4. CSS3 styling code written by the developer (see Styling and Layout)

The files that form the client-component will initially reside on a server. From there they can be deployed to be executed in a browser or a web-view container.

In the typical mobile environment, these client-component files would be packaged and installed on a smartphone or tablet as an 'app'. If the 'app' uses REMOTESQL, it would, at run-time, access the remote database via HTTP and the server-component.

Figure 29 shows how they interact in a typical configuration.

Theoretically, these components can be arranged in a number of ways. We have chosen a structure that allows us to:

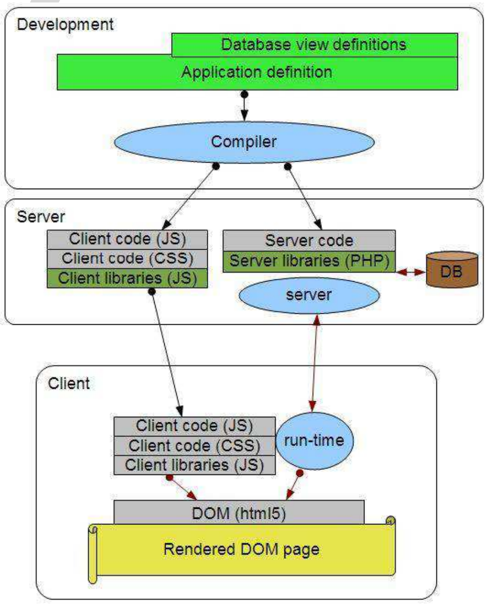


Figure 29: Code Production and Operation

- compile and execute the application in a single step, or
- simply execute a pre-compiled version of the application.

8.4 WIDE Application designer - the IDE

We are developing an application designer tool for assembling the block components into an application and specifying the necessary attributes. Currently it is a text-based tool but will shortly use a graphical format with drop-down menus to specify types of blocks, attributes, events and values. Its operation is illustrated in Figure 30.

9. CURRENT AND FUTURE WORK

We have provided an overview of the WIDE toolkit as it is currently used for building mobile apps. However, there is significant development occurring to make it more broadly applicable.

Currently there is new graphical user interface being constructed by Toacy Oliveira and one of his students at the Federal University of Rio de Janeiro. The objective is to allow the developer to "draw" the interface.

The system now incorporates a local database that supports synchronization with remote databases thereby supporting disconnected operations.

Interaction with maps through map editing is also a feature being added.

Future work includes improvements to the charting capability, possible new block structures and new target languages such as Java.

We are also examining how to expose the event manager

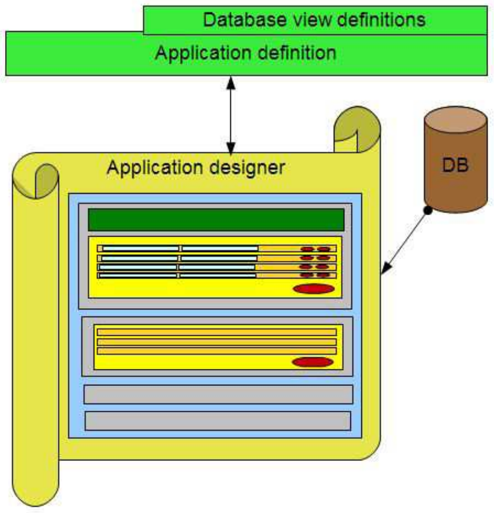


Figure 30: Application Design

structure so as to support various versions of publish-subscribe and the systems that can be derived from it. Finally we need to keep looking at the language and ensuring that it meets the needs of the user community. Creating data models and views of those models are exercises in abstraction which many developers and users find difficult. One open question focuses on how we can provide the right tools to guide developers and users to the "right" abstractions particulary with data models and data views. Tools such as spreadsheets and file systems such as WATFILE [12] might provide some guidance.

10. CONCLUSIONS

This paper has outlined our thinking on simplifying development and maintenance for web-based and mobile applications and to putting software development tools in the hands of the domain experts or users. Although the work is not complete we believe we have identified many of the basic approaches to constructing software assets or meta-components and meta-processes to fit our model, particularly in the area of data models, human interfaces and event handling.

The work outlined in this paper is our latest attempt at constructing a framework for our thinking. The method has been partially validated in that it was used by one person to construct two successful mobile apps over three months while the model was under development by that same individual. The mobile apps are "Crush the Crave" and "Building Stories." The first one was developed for the Android platform, while the second one runs on Android, BlackBerry Torch and iPhone. Both apps also required development of an extensive back-end portal and database, which was performed by a second member of our team.

We have many years experience in understanding and working with complex programming structures and have also developed methods that will deal with complex publish-subscribe, mediated social networks, and synchronous collaboration that will be supported by the model. Agents are very broad in application and we dealt with them in a limited way, but believe they can fit in our framework although agent development will likely require programmers for the foreseeable future.

11. ACKNOWLEDGMENT

The authors would like to thank the Ontario Research Fund (ORF), the Natural Science and Engineering Research Council of Canada (NSERC) and the Centre for Community Mapping for their financial support and cooperation in the technology described in this paper. They would also like to thanks Carlos Lucena, Arndt von Staa and Toacy Oliveira for their valuable and valued input. A special thank you is owed to Doug Mulholland who has been involved in the various versions of WIDE since the beginning of the project. Thanks are also due to Kyle Young, Anthony Robins, Jessica Battrick and Marcus Doran who have worked on many of the applications developed using various versions of the WIDE toolkit and have made many valuable suggestions on research directions.

12. REFERENCES

[1] E. Bertino, L. Martino, F. Paci, and A. Squicciarini. *Security for Web Services and Service-Oriented Architectures*. Springer, 2010.

[2] D. Cowan, P. Alencar, F. McGarry, and C. Lucena. A web-based framework for collaborative innovation. Technical Report Technical Report CS-2012-02, David R. Cheriton School of Computer Science, University of Waterloo, 2012.

[3] D. Gollman. *Computer Security*. Wiley Publishing, 3 edition, 2011.

[4] J. Hendler, N. Shadbolt, W. Hall, T. Berners-Lee, and D. Weitzner. Web science: An interdisciplinary approach to understanding the web. *Communications of the ACM*, 51(7):60–69, July 2008.

[5] J. Longstaff, M. Lockyer, and J. Nicholas. The tees confidentiality model: an authorisation model for identities and roles. In *Proceedings of the eighth ACM symposium on Access control models and technologies*, 2003.

[6] M. McLuhan. *Understanding Media: The Extensions of Man*. MIT Press, 1994.

[7] Open GeoSpatial Consortium (OGC). OGC® Standards and Supporting Documents. Web, 2012. Available at: http://www.opengeospatial.org/standards.

[8] J. Surowiecki. *The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies and Nations*. Random House, 2004.

[9] Wikipedia. Publish-subscribe pattern. Available at http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern.

[10] Wikipedia. Web, 2012. Available at: http://en.wikipedia.org/wiki/IFilter.

[11] Wikipedia. Shapefiles. Web, 2012. Available at: http://en.wikipedia.org/wiki/Shapefile.

[12] T. Wilkinson. The watfile tutorial and user's guide, February 1985. Available at: http://www.uic.edu/depts/adn/infwww/txt/v4407001.txt.