A gentle introduction to the Wt C++ Toolkit for Web Applications

Koen Deforche and Wim Dumon

January, 2006

* Originally published in **Software Developers Journal** April 2006 issue, and brought up-to-date for Wt version 1.1.0.

1. Introduction

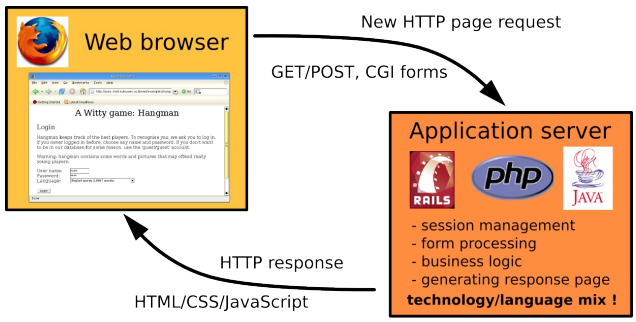
Web application technology for the future

C++ is an established language for developing many kinds of software such as desktop applications, email clients, database engines, and so on. Still, the use of C++ for creating "web applications" has been limited. Instead, languages that dominate web application development are JAVA, PHP, Python, and Perl. With the exception of PHP, which is a language specifically designed for web development, web application frameworks are offered for the other languages to aid in web application development. Examples are J2EE and Struts for JAVA, Perl::CGI for Perl, or Zope for Python. These frameworks provide session-management, support for parsing data transmitted from the web browser from HTML forms and cookies, and help in generating the new page in response.

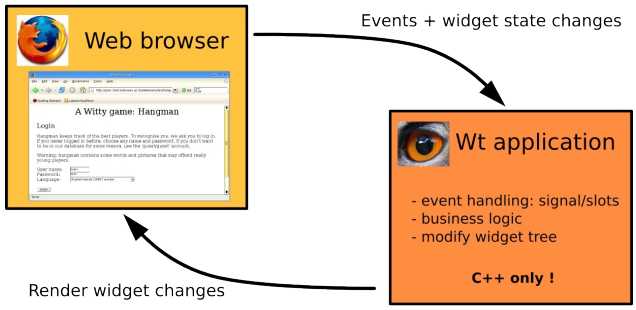
The paradigm followed by these web applications frameworks is illustrated in Figure 1(a). At each step, the web browser requests a new page to the web server, and may submit within the request a number of form values. At the server end, the web application processes the request, identifies the session, and performs business logic. Finally, the application generates the response page. The response page may contain not only HTML, but also JavaScript to enhance the interactivity of the web application. However, JavaScript has many quirks in different web browsers, and therefore requires great effort to write in a portable way.

New and highly successful web applications such as Google's Gmail or Google Maps, however, do no longer follow this page-by-page paradigm. Instead, they use a combination of JavaScript and server-side techniques, often referred to as AJAX, to dynamically update the web page with new content received from the server, *without reloading the web page.* AJAX works by using JavaScript to make a HTTP request in the background to the server (e.g. 'Any new email?'). The server generates the appropriate response (e.g. 'Yes, 2: (1) ..., (2) ...'), in XML format. Finally, client-side JavaScript parses the response and updates the web page by manipulating the Document Object Model (DOM) (e.g. by prepending the two email messages to the inbox list).

This new technology which enjoyed much hype in 2005, causes a fundamental change in how web applications may be built. It is no longer necessary to transmit an entire new web page in response to every event. AJAX has fueled new possibilities for the web, and is therefore sometimes associated with an entire new and more interactive version of the web: Web 2.0. At the same time, the use of these technologies poses many



(a) traditional model for web application frameworks



(b) Wt application model

Figure 1: (a) The dynamic page model traditionally used by web application frameworks, contrasted with (b) the event-driver model traditionally used by GUI toolkits, and Wt.

challenges. The application developer needs to learn and absorb a number of technologies in order to use AJAX. In addition to the server-side framework and HTML/CSS, the developer needs to learn Dynamic HTML (DHTML) which is the manipulation of the Document Object Model (DOM) using JavaScript, some details of the HTTP protocol in order to generate valid GET or POST requests using JavaScript, and finally the usage of the XMLHttpRequest API. And to top it off, as always there is the variety of browser dialects to take into account, plus the desire to keep supporting older browsers lacking these new technologies.

Enter Wt !

In contrast with the page model of traditional web applications frameworks, the model followed by Wt or a traditional GUI library is based on widgets, see Figure 1(b). The widgets are conceptually organized in a tree, and callback functions are attached to particular events. In response to an event, the callback is called, some work gets done, and/or the widget tree is modified.

Wt is a young C++ widget library for developing web applications. The application model of Wt is similar to the application models of existing GUI libraries (such as

Microsoft's MFC or TrollTech's Qt). At the same time Wt hides many underlying technology details (HTML, Forms/CGI, JavaScript, AJAX, etc...) from the developer, not unlike how the Qt library hides the underlying X library or Microsoft Windows GUI details.

Because the API of Wt makes abstraction of the underlying technologies (Forms, JavaScript or AJAX), Wt chooses how to communicate with the web browser depending on technology support in the browser. The responsibility for making the application work in the jungle of web browsers is therefore also transferred from the application developers to the library developers.

In the remainder of this introductory article to Wt, we will first give an overview of the main classes and features, as well as an explanation of what Wt does behind the scenes. Next we will show how to use Wt by implementing the classic “hangman” game.

2. Library overview

Main components

The entire user-interface is organized in a hierarchical widget tree of `WWidget` objects. A `WWidget` corresponds to a visible entity such as for example a piece of text (`WText`), a table (`WTable`), a line edit (`WLineEdit`), or a more complex composite widget (classes that implement `WCompositeWidget`). The user-interface, which corresponds to the web page, is specified by creating and maintaining this widget tree. Every `WWidget` corresponds to a rectangular piece, and manages the contents and events within that rectangle.

The library provides a number of basic widgets that correspond directly to the widgets provided by HTML, and which are all descendants of `WWebWidget` (`WText`, `WTable`, `WImage`, ...). These widgets internally manipulate a server-side HTML DOM, which is then used by the core library to update the web page rendered by the browser. In contrast, `WCompositeWidget` objects are widgets that are implemented by composition of other widgets. These widgets do not manipulate the DOM themselves but merely use the public API of the composing widgets. While Wt provides a number of these composite widgets (such as a tree-list widget and an auto-complete line edit), these widgets do not necessarily belong to the library, since they are implemented on top of Wt.

Every Wt application must start with the instantiation of a `WApplication` object. This object manages the root of the widget tree, information on browser capabilities and manages internationalization support using a locale and message resource bundles (see further).

Session management

Similar to how multiple instances of conventional applications may be run concurrently, so will the Wt core system spawn multiple Wt applications for every independent web session. Each new "session" implies a new path of execution which starts in `wmain()`, which is the application entry point. Thus, the programmer only needs to implement a single-user application, unless users interact with a common component (such as a database) or with each other, for which standard data-sharing mechanisms must be used.

The current version of Wt implements these different paths of execution using different processes. Thus, for every new session, Wt spawns a new process. This has the main benefit of enjoying kernel-level memory protection between user sessions. As a consequence simple programming mistakes will not automatically compromise session-privacy. The downside of this approach is cost: current kernel implementations may require some amount of non-swappable memory associated with every process. In the future, Wt may offer different thread implementation choices, including user-level threads.

Signal/Slot event propagation

Wt uses a signal/slot implementation for event propagation. User-interface events, such as mouse clicks, or text modifications, are exposed by Wt as *signals* associated with particular widgets. To respond to an event, the programmer connects the respective signal to a *slot*. Any object method with a signature compatible with the signal may be used as a slot. Whenever the signal is emitted, all slots that have been connected to the signal are called. The signal/slot paradigm is a well-established type-safe and self-managed alternative to callbacks.

Internationalization

Internationalization and localization is an important property of a website, given the inherent global scope of the World-Wide-Web. Wt assists in internationalization by offering message resource bundles. A `WMessage` object provides a piece of text which is dependent on the current locale. Widgets that display text to the user (such as `WText`) may be given a `WMessage` instead of raw text. The message translations for every locale are stored in XML format in message resource files, one for every locale. When changing the application locale, using `WApplication::setLocale()`, the application automatically updates the corresponding widgets with the localized text.

Non-intrusive upgrades

Web applications enjoy a major advantage over conventional applications since the publisher can easily upgrade all copies of the application, by merely deploying a new version on his website. Usually the publisher may not want to terminate running sessions when deploying a new version, but instead offer the new version to new sessions. This process of non-intrusive upgrades is the default method of upgrading in Wt.

Session lifetime

Wt uses a keep-alive protocol between client and server to determine session lifetime. As long as the web page is displayed in the user's browser, the session is kept alive, otherwise the session is terminated. In addition the application can choose to end the session (for example in response to the user 'logging out'), by calling

`WApplication::quit()`. Either way, when a session terminates, the main widget is destroyed. This allows the application to release any application-specific resources.

How does it work?

Wt implements two main tasks: rendering and maintaining the HTML DOM tree in the web browser, and responding to user input and user events, such as mouse clicks.

All events that may be caught for processing are mapped to signals, which are available in the respective widgets. When an event is triggered by the user (e.g. a click on an 'OK' button), the web browser communicates the target object and corresponding signal (for example `OkButton->clicked`), together with all form data to the web server (using AJAX or plain HTML form submission). At the server, the corresponding Wt application instance processes first all form data to update the widget tree state. Then, the event is propagated by emitting the signal of the target object, which triggers all connected slots. These may perform business logic and modify the widget tree. Modifications to the widget tree are tracked by the library, and converted to modifications to a server-side model of the HTML DOM tree. Finally, depending on the method for communication, either the DOM tree changes, or the complete modified DOM tree are communicated back to the web browser, completing the event cycle.

Because of the clear separation between user-interface specification using the widget tree and the mechanism of rendering the tree, Wt optimizes rendering for increased responsiveness when AJAX is available. Wt accomplishes this by transmitting only visible widget changes during the first communication with the web browser. As soon as the page is loaded, remaining hidden widgets changes are transmitted in the background. As a consequence, both the initial response is optimized and the appearance of subsequent widgets appears snappy.

3. Tutorial

The tutorial section discusses two small programs to illustrate various Wt library concepts. The first program is a Hello World application, introducing two key concepts of the library: the widget tree and signal/slots. The second larger program is an online version of the classic hangman game, including a user ranking system, backed by a small database. The game is available online[1]. The hangman game illustrates how a widget tree for a more complex web application is constructed and managed, how to write your own widgets, signals and slots, how layout is handled, and offers an example of how data can be extracted from a database to be displayed on the website. The complete source code of the game is around 900 lines including comments. We selected the most interesting parts for this tutorial.

The Wt documentation page[2] contains an exhaustive list of classes, methods, signals and slots exposed in the Wt API. Even the Hangman demo only uses a small portion of the available classes and methods. The complete sources of the tutorial examples, together with Makefiles to build them, are included in the Wt source distribution.

The omnipresent Hello World

The entry point of every Wt program is the `wmain()` function. The simplest Wt program must instantiate the `WApplication` object, and call the application idle loop. For the hello world application, a `WText` object and a `WPushButton` were added, having the root of the widget tree as parent. This will make them appear in the web browser. Clicking the Quit button cleanly terminates the session. This is achieved by connecting the `clicked` signal of the Quit button with the `quit()` slot of the application.

It is important to remark that there is no obligation to call `quit()` explicitly. When the user navigates away from the web application, Wt will detect that keep-alive messages are no longer received, and Wt will terminate the session as if `quit()` was called.

Listing 1. Hello World with Quit button

```
#include <WApplication>
#include <WText>
#include <WPushButton>
int wmain(int argc, char **argv)
{
   WApplication appl(argc, argv);
   // Widgets can be added to a parent
   // by calling addWidget() ...
   appl.root()
     ->addWidget(new Wtext(
                 "<h1>Hello, World!</h1>"));

   // ... or by specifying a parent at
   // construction time
   WPushButton *Button
     = new WPushButton("Quit", appl.root());

   Button->clicked.connect
          (SLOT(&appl, Wapplication::quit));
   return appl.exec();
}
```

My first widget

We start the discussion the hangman game with the login process. This is handled by the LoginWidget. We have kept the business logic to a minimum, and indeed a bit simplistic. The LoginWidget invites the returning user to login using his user name and password, and to choose one of the available dictionaries from which words will used for the game. If the user was not present in the user database, we assume a new user and he is automatically added to the database. On successful login, a confirmation is displayed, otherwise the user is notified of the problem and may try again.

Hangman's LoginWidget demonstrates the possibility to write self-contained widgets in a nice object-oriented fashion, with clean interfaces, ready to be plugged in where they are required. The LoginWidget, with its non-standard behavior, may not immediately be a candidate for reuse, but nevertheless it demonstrates the interface principles. The LoginWidget has only two public member functions. The first is the constructor, which takes the parent widget as an argument. Since all built-in widgets take their parent as an argument in the constructor, it is a consistent approach to do this as well for custom widgets. The second member is a signal that will be emitted when the login process has been successfully completed. The signal also carries the user name, and the chosen game dictionary.

The object oriented widget tree approach for GUI libraries has led to a significant amount of widget reuse, which is evident from large scale desktop projects such as KDE. Especially where widgets cover pretty simple concepts, reuse of a widget is often no more complex than instantiating it in the right location of the object tree. The remainder of a program may interact with the widget using its methods and by installing callback functions to react to events. Traditional GUI widget examples are an expandable tree list, a file-open dialog, etc... which are almost always included in the GUI libraries. Wt introduces the exact same paradigm to the world of web programming, and invites the programmer to partition a web application in well defined and self

1Available at http://jose.med.kuleuven.ac.be/wt/examples/hangman/hangman.fcg

2Available at http://jose.med.kuleuven.ac.be/wt/doc/

contained widgets.

Listing 2. LoginWidget class definition.

```
class LoginWidget : public WContainerWidget
{
public:
  LoginWidget(WContainerWidget *parent = 0);

public signals:
  Wt::Signal<std::string, Dictionary>
    loginSuccessful;

private slots:
  void checkCredentials();
  void startPlaying();

private:
  WText *IntroText;
  WLineEdit *Username;
  WLineEdit *Password;
  WComboBox *Language;
  std::string User;
  Dictionary  Dict;
  void confirmLogin(const std::string text);
};
```

In Listing 2, we show the class definition of the LoginWidget class. The LoginWidget defines a public signal, `loginSuccessful`, and uses internally a number of slots to react to user events. Therefore, we must inherit (directly or indirectly) from `WObject`. We then declare signals and methods that will be used as in the class declaration.

LoginWidget inherits from `WContainerWidget`. A `WContainerWidget` is a widget which holds and manages child widgets. The parent of a widget is always a `WContainerWidget` or one of its derived classes (such as `WStackedWidget` or `WTableCell`). Children in a `WContainerWidget` are layed out in the order in which they were added to the container. The inline property of a widget determines its default layout behavior within the container. In-line widgets are layed out like words in a text, following lines and wrapping at the right border of the container. Non in-line widgets are formatted as a new paragraph. Widgets may also be lifted out of this default layout flow to be manually positioned in various ways, but we will not discuss this here. Instead, as illustrated in the constructor discussed below, a `WTable` is used to create a more sophisticated layout.

The login widget, as rendered by Firefox, is shown in figure 2. The code that generates this interactive form is entirely located inside the constructor of LoginWidget, which is shown in Listing 3.

Figure 2: The hangman login widget, right after construction. Listing 3 is the source code for this web page.

Listing 3. The LoginWidget constructor implementation.

```
LoginWidget::LoginWidget(WContainerWidget*
                         parent)
  : WContainerWidget(parent)
{
  setPadding(WLength(100), Left | Right);

  WText *title = new WText("Login", this);
  title->decorationStyle().font()
    .setSize(WFont::XLarge);

  IntroText = new Wtext(
    "<p>Hangman keeps track of the best"
    "players. To recognize you, we ask you "
    "to log in. If you never logged in "
    "before, choose any name and "
    "password.</p>",
    this);

  WTable *layout = new WTable(this);
  WLabel *usernameLabel
    = new WLabel("User name: ",
                 layout->elementAt(0, 0));
  layout->elementAt(0, 0)
    ->resize(WLength(14, Wlength::FontEx),
             WLength());
  Username = new WLineEdit(
    layout->elementAt(0, 1));
  usernameLabel->setBuddy(Username);

  WLabel *passwordLabel
```

```
    = new WLabel("Password: ",
                 layout->elementAt(1, 0));
  Password = new WLineEdit(
    layout->elementAt(1, 1));
  Password
    ->setEchoMode(WLineEdit::Password);
  passwordLabel->setBuddy(Password);

  WLabel *languageLabel
    = new WLabel("Language: ",
                 layout->elementAt(2, 0));
  Language = new WcomboBox(
    layout->elementAt(2, 1));
  Language->insertItem(0,
   "English words (18957 words)");
  Language->insertItem(1,
    "Nederlandse woordjes (1688 woorden)");
  languageLabel->setBuddy(Language);

  new WBreak(this);

  WPushButton *LoginButton
    = new WPushButton("Login", this);
  LoginButton
    ->clicked.connect(SLOT(this,
        LoginWidget::checkCredentials));
}
```

The constructor introduces a number of new concepts. We had encountered the `WText` and `WPushButton` already in the hello world example. The new widgets will hold few surprises. `WLineEdit` provides a single line edit input and `WComboBox` provides a drop-down selection box. The latter is populated with selection options using `insertItem()`. We use the `WLabel` class to provide labels for the three input fields, and tie them to the corresponding input-field using `setBuddy()`. By using `WLabel` instead of `WText`, the user may click on the label to give focus to the corresponding input field. `WTable` is a table, in this case used for layout purposes. The table cells, accessed using elementAt(row, column), are used as parent widget for some of the text widgets, the line inputs, and the combo box, so that they are layed out in an array. Finally, `WBreak` is the equivalent of the HTML line break tag (`<br />`), and lets subsequent inline widgets start a new line.

The `setPadding()` call adds empty space within LoginWidget between its border and its children. The `WLength` class offers an interface to the CSS method of specifying sizes. An 'automatic' length is created by calling the default constructor. When constructed with parameters, a value and a unit (defaulting to `WLength::Pixels`) are specified. All CSS units (pixels, font height, font width, cm, percentage, ...) are supported by the `WLength` class. A few lines below the `setPadding()` call, the `WLength` class appears again, in the line where the table cell is resized. The table width is set to 14 font width units, while the height remains the default.

The last important new aspect of Wt in this constructor is the use of `decorationStyle()` to access style properties of a widget, which we use to set the font size of the title to extra large. In the hello world example, we used the old-fashioned `<h2>..</h2>`, but here we demonstrate that you can apply CSS-based styles. This method will reappear in other functions.

We use `setEchoMode()` to mask the entered password with stars. Finally, the `connect()` call is similar as in the hello world application, but here we connect the `clicked` signal to the `LoginWidget::checkCredentials` slot.

Listing 4. Methods `checkCredentials()` and `confirmLogin()` of the LoginWidget class.

```
void LoginWidget::checkCredentials()
{
  User = Username->text();
  std::string pass = Password->text();
  Dict
    = (Dictionary) Language->currentIndex();

  if (HangmanDb::validLogin(User, pass)) {
    confirmLogin("<p>Welcome back, "
                 + User + ".</p>");
  } else
    if (HangmanDb::addUser(User, pass)) {
      confirmLogin("<p>Welcome, "
        + User
        + ". Good luck with your first"
          " game!</p>");
    } else {
      IntroText
        ->setText("<p>You entered the wrong"
          " username/password, please try"
          " again.</p>");
      IntroText->decorationStyle()
        .setForegroundColor(Wt::red);
      Username->setText("");
      Password->setText("");
    }
}

void LoginWidget::confirmLogin(
                      const std::string text)
{
  clear();

  WText *title
    = new WText("Loging successful", this);
  title->decorationStyle().font()
    .setSize(WFont::XLarge);

  new WText(text, this);
  (new WPushButton("Start playing", this))
    ->clicked.connect(SLOT(this,
                LoginWidget::startPlaying));
}
```

In `checkCredentials()` we validate the user and

password that were entered. Therefore, it is a fine example of how user input is retrieved and how the webpage can be modified as a reaction to user input. In the first three lines, the user name, password, and language selection provided by the user are retrieved from the `WLineEdit` and `WComboBox` widgets, using respectively the `text()` and `currentIndex()` method calls. These methods always return the up-to-date values for these widgets, without any intervention from the programmer. This may not be surprising for a GUI library, but is a huge simplification compared to the traditional and tedious form-based content retrieval commonly found in web application frameworks.

In the subsequent code, the credentials are verified, and we call `confirmLogin()` when the login was successful. Otherwise we change the displayed message stored in the IntroText widget using `setText()` to notify the user of the failure. In addition, we change the message text color to red to alert the user. Text color is another property that can be accessed using `decorationStyle()`, which we previously used to set the font size of the title. Finally, we complete the slot implementation by clearing the user name and password text.

The most interesting and maybe surprising aspect of this slot implementation may be in the code that is not there! Wt has two possible ways for updating the web browser page: either by letting the browser move to a new page, or by using JavaScript, AJAX and DHMTL to update the current web page. Because the code does not specify the mechanism but only the desired result, Wt may use either of these methods depending on support for JavaScript and AJAX at the client.

We have kept the database interface simple on purpose. `HangmanDb::validLogin()` verifies if the user/password combination is stored in the database. If this fails, we try to add the user by means of the `HangmanDb::addUser()` call. This call will fail if the user name was already in the database. Even though the implementation of these functions will not be discussed in this article, it is worth to mention that they use the MySQL++ library to interact with a MySQL database.

When the login was successful, we display a welcome message and a confirmation of the login, which is implemented in `confirmLogin()`. If you have digested the LoginWidget implementation well so far, you will find that this method contains no new magic. First, we use `clear()` to clear the container widget contents. Finally, in one statement we create a new button and at once connect its `clicked` signal to the `startPlaying()` slot.

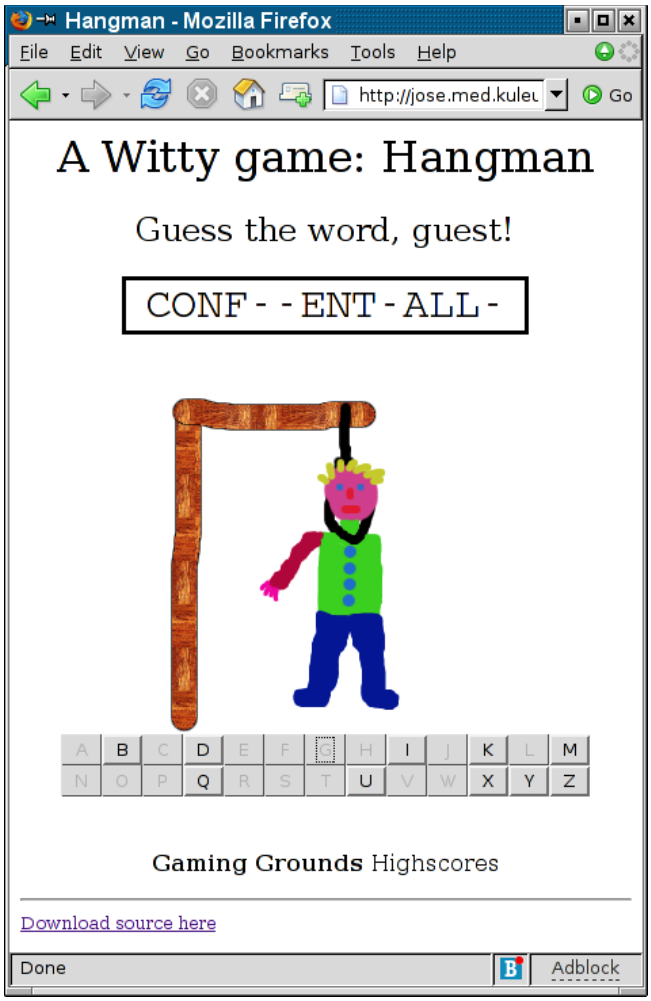


Figure 3: Hangman screen shot: an AJAX and DHTML web-application, entirely programmed in C++. All images are preloaded in the browser for improved user experience.

Listing 5. `LoginWidget::startPlaying()` slot implementation.

```
void LoginWidget::startPlaying()
{
   emit(loginSuccessful(User, Dict));
}
```

The `startPlaying()` slot demonstrates how to emit a signal, in this case with arguments. What happens in response to the signal depends on slots that have been connected to this signal.

This concludes the entire implementation of the LoginWidget. In the remaining of the tutorial, we will reuse the same concepts that were used for the LoginWidget, to create highly responsive AJAX-enabled web applications. Because of the widget abstraction, you need no knowledge of JavaScript and even your knowledge of HTML can be minimal. On the other hand, Wt makes no big effort to abstract the decorative CSS

concepts, instead exposing them almost directly using `decorationStyle()`. Therefore, familiarity with CSS will help you to style a Wt application.

The second widget: unleashing Wt's power

Until now, we introduced a rather unique paradigm to program web applications. We demonstrated Wt's added value to the programmer. The next widget also illustrates some new Wt widgets and features, but we also demonstrate an important aspect of Wt that highly enhances the user experience. One of the most appealing features of popular web applications like Google's Gmail and Google Maps is the excellent response time. Google has spent quite some effort in developing client-side JavaScript and AJAX code to achieve this. With few effort – indeed almost automatically – you can get similar response times using Wt, and indeed the library will be using similar techniques to achieve this. A nice bonus of using Wt is that the application will still function correctly even when AJAX or JavaScript support is not available! The HangmanWidget class, which we discuss next, contains some of these techniques.

HangmanWidget contains the real hangman game logic. Figure 3 is a screen shot of the game in action. For each new game, the program chooses a random secret word for the player to guess. From the alphabet, the player guesses a letter, and if the letter is part of the secret word, its occurrences in the word are revealed. In case of a wrong guess, you get one step closer to a hanging man. At the end of the game, we update the users score in the database, and offer the user the possibility to start a new game. The implementation of HangmanWidget contains few novelties, except for how we handle the hangman images.

In the constructor we construct the user interface. The part that constructs the images is isolated in the method `createImages()`.

Listing 6. Hidden widgets are prefetched by the browser, ready to be displayed when `show()` is called.

```
void HangmanWidget::createHangmanImages
                   (WContainerWidget *parent)
{
  for (unsigned int i = 0;
       i <= MaxGuesses;
       ++i) {
    std::string fname = "icons/hangman"
      + boost::lexical_cast<std::string>(i)
      + ".png";

    WImage *theImage
      = new WImage(fname, parent);
    HangmanImages.push_back(theImage);
  }

  HurrayImage
    = new Wimage("icons/hangmanhurray.png",
                 parent);

  resetImages(); // Hide all images
}

void HangmanWidget::resetImages()
{
  HurrayImage->hide();
  for (unsigned int i = 0;
       i < HangmanImages.size();
       ++i)
    HangmanImages[i]->hide();
}
```

This function introduces a new widget: the `WImage`, which not surprisingly corresponds to an image in HTML. The code shows how all widgets are created and inserted into the HangmanWidget. With what we discussed until now, we would expect that all images are displayed at the same time, which is clearly not what we want. Therefore, we call `resetImages()` after the images are created, and this method calls `hide()` on every image, after which none of them are visible. The game logic will show and hide the images, so that only the correct one is visible at any point in the game. Every `WWidget` can be hidden, and hidden widgets can be redisplayed by calling `show()`. But why do we create and hide them, where instead we could simply create and delete the `WImage` that we want to show? Alternatively we could work with only one image and modify the source of the image to change image! The answer lies in the response time, at least when AJAX is available. Wt first transfers information about visible widgets to the web browser. When the visible part of web page is rendered, the remaining hidden widgets are transmitted and inserted by the web browser into the DOM tree. Most web browsers will also preload the images referred to in these hidden widgets. As a consequence, when the user clicks on a letter button and we need to update the hangman image, we simply hide and show the correct image widget. Then, only a single HTTP request with a small response are communicated between the web browser and the server. An alternative implementation would invariably have caused the browser to fetch the new image, requiring a second round-trip to the server, plus the time to download the image. The hangman game uses this principle of hidden widgets frequently, for example also when you switch between high scores and the game. At any point in time, only one of these widgets is shown, and the user switches between these two widgets using the menu bar at the bottom. Wt is able to further reduce the reaction time in some cases by transferring the slot implementation completely to the browser, with the use of the so-called static signal/slot connections, but that discussion falls outside the scope of this introductory tutorial.

In summary, the use of hidden widgets is a simple and effective way to implement performant Wt applications. Hidden widgets do not compromise the application load time, since visible widgets are transferred first.

We will skip the implementation of the HighScoresWidget and the HangmanGame because they demonstrate no fundamental additional features. HighScoresWidget displays the highest ranking users, and HangmanGame connects together the LoginWidget, the HangmanWidget, and the HighScoresWidget. Information-hungry readers are however invited to take a quick look at the HangmanGame source, since it uses clickable text (no, not hyperlinks) to implement a menu bar and demonstrates the use `WStackedWidget`, a specialization of the `WContainerWidget`.

4. Summary

The Wt library provides an effective way to implement web applications and frees the application developer of many technical aspects and quirks associated with new web technologies such as JavaScript, AJAX and DHTML. Because of the many similarities between Wt and other GUI toolkits, application developers can treat the web browser in many aspects as just another GUI platform.

The tutorial demonstrated many important Wt features, but far from all Wt features. Static slots, which further improve event response times, file uploads and dynamic resources, internationalization, full CSS support, and many undiscussed widgets are only a selection of what we had to skip. For more information, we refer the reader to the online Wt documentation resources.