
A hands-on introduction to Wt

Table of Contents

1. Introduction	1
2. Main components	2
2.1. Widgets	2
2.2. Session management and lifetime	3
2.3. Deployment	3
3. Hello, Wt	3
4. Hangman	6
4.1. A first custom widget	6
4.2. Unleashing (some of) Wt's power	10
4.3. Internal paths	11
5. Summary	13

Pieter Libin <pieter@emweb.be>
Koen Deforche <koen@emweb.be>
Wim Dumon <wim@emweb.be>

For Wt 3.2.0 (December 5 2011)

1. Introduction

Wt is a C++ library for developing web applications. Admitted, C++ doesn't come to mind as the first choice for a programming language when one talks about web development. Web development is usually associated with scripting languages, and is usually implemented at the level of generating responses for incoming requests. Since both requests and responses are text encodings, web programming is ultimately a text processing task, and thus conveniently expressed in a scripting language.

Yet, from a functional point of view, and as a programming task, a modern web application resembles more a desktop application: it is highly interactive and no longer organized in pages (perhaps still conceptually, but not physically). Interaction with the server happens more and more in the background and not with complete page refreshes. Indeed, the browser is being transformed into a platform for applications where users interact with data and more often than not with each other.

While some developers choose to implement the application in JavaScript and use a client-server protocol to access server-side resources, this has a number of inherent downsides. You need to let go of static typing (or add another layer, such as GWT). Type safety, enforced by a compiler, is invaluable as a project grows in complexity and number of developers. Moreover, you need to design a client-server protocol and minimize communication and associated round-trip latencies. Finally, this approach cannot work for applications that need to meet accessibility guidelines or need to be search engine optimized and thus require a HTML-only version of the application. Wt, as a server-side solution, overcomes these problems, and with little to no interactivity loss compared to a JavaScript application.

Wt's main advantage over other server-side approaches is its **widget abstraction**. An application is defined as a hierarchy of widgets. Some of these widgets are generic and provided by the library (like push buttons, line edits, table views, ...), and others are specialized for a particular application (e.g. an entire blog widget). A widget encapsulates the view and behavior aspects, consumes and produces events and also participates in URL handling and may interact with other HTTP resources. More often than not, a widget delegates the actual "logic" to a model, a layering approach typically known as MVC. A widget, unlike a page or "partlet", is a truly reusable, customizable and extensible building block (as a poster-child of Object-Oriented Programming) for a modern web application. The actual request handling and rendering is abstracted, with as benefit that a full page rendering model (plain HTML) or incremental updates (Ajax/WebSockets) can be used depending on configuration and browser properties.

Although implemented in C++, Wt's main focus or novelty is not its performance, but its focus on developing maintainable applications and its extensive library of built-in widgets. But because it is popular and widely used in embedded systems, you will find that performance and foot-print has been optimized too, by virtue of a no-nonsense API, thoughtful architecture, and C++ ...

In this tutorial, we will use two small programs to illustrate how to use Wt to create web applications. After this tutorial you should have a good grasp of what the possibilities are, how Wt applications are built, and how it offers you a tool to manage complexity.

The first application is the obligatory Hello World [<http://www.webtoolkit.eu/wt/examples/hello/hello.wt>] application and introduces two key concepts of the library: creating and updating a widget tree and reacting to user input. The second, slightly larger application is the classic hangman game [<http://www.webtoolkit.eu/wt/examples/hangman-game/hangman.wt>]. Both applications are included in the Wt source distribution.

But before we dive in, let's quickly go over the main concepts of the library.

2. Main components

2.1. Widgets

The user-interface, rendered in a browser window, is specified by creating and maintaining a widget tree. Every widget corresponds to a rectangular part of the user-interface, and manages its contents and behavior. At the heart, the library takes charge of two tasks within a single session: (1) **rendering** this widget tree as HTML/JavaScript in the web browser, and tracking changes as incremental updates, and (2) **synchronizing user input** and **relaying events** from the browser to these widgets.

Because of the clear separation between user-interface specification using the widget tree and the rendering of the tree, the library implements several optimizations for rendering when Ajax is available. Not only can the library update the interface incrementally, there are other tricks such as only rendering visible widget changes during in response to an event (or initial loading), and in the background render changes to hidden widgets. As a consequence, both the initial response is optimized and the appearance of subsequent widgets appears snappy.

2.2. Session management and lifetime

Another aspect that is entirely handled by the library is session management. For every new session, which corresponds to a new user accessing your web application, the library creates a new `WApplication` object. As a developer, you can implement an application pretty much as a single-user application, unless you let users interact with a common component (such as a database) or with each other, for which standard data-sharing mechanisms must be used.

Depending on the deployment model, the library will map sessions onto dedicated or shared processes. When using dedicated processes a new process is created for every distinct session, this provides the kernel-level isolation of different sessions, which may be useful for highly security sensitive applications. When using shared processes new sessions are allocated randomly to one of the processes available to the library. This reduces the danger for DoS attacks, but requires more careful programming as memory corruption affects all sessions in a single process, and sessions are not isolated by any other mechanism but correct programming.

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. When the user closes his window, navigates away, or after a timeout when connectivity is lost, the session gets terminated. When a session is terminated, the application object together with the entire widget tree is deleted, therefore you should release resources held by your widgets or application in the destructors of these objects.

2.3. Deployment

Several deployment options are available:

- The `wthttp` connector library implements a webserver that implements HTTP(S) and WebSockets. It is not only convenient during development, but also an efficient solution for deployments ranging from small embedded systems or mobile devices to multi-server deployments.
- The `wtfcgi` connector library implements the FastCGI protocol. It allows a Wt application to be integrated into an existing web server (such as Apache, Lighttpd or Nginx).
- The `wtisapi` connector library implements the ISAPI protocol. On Windows platforms, it allows a Wt application to integrate into Microsoft Internet Information Server (IIS).

3. Hello, Wt

In this example [<http://www.webtoolkit.eu/wt/examples/hello/hello.wt>], we show an application that prompts the user for his name. When he pushes the button, the greeting text is updated based on the name that was entered.



A new session starts with creating an instance of class `WApplication`. This object manages the root of the widget tree and contains other session information, such as the connected browser's capabilities.

A complete "Hello world" application.

```
#include <Wt/WApplication>
#include <Wt/WBreak>
#include <Wt/WContainerWidget>
#include <Wt/WLineEdit>
#include <Wt/WPushButton>
#include <Wt/WText>

class HelloApplication : public Wt::WApplication
{
public:
    HelloApplication(const Wt::WEnvironment& env);

private:
    Wt::WLineEdit *nameEdit_;
    Wt::WText *greeting_;

    void greet();
};

HelloApplication::HelloApplication(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
{
    setTitle("Hello world");

    root()->addWidget(new Wt::WText("Your name, please ? "));
    nameEdit_ = new Wt::WLineEdit(root());
    Wt::WPushButton *button = new Wt::WPushButton("Greet me.", root());
    root()->addWidget(new Wt::WBreak());
    greeting_ = new Wt::WText(root());
    button->clicked().connect(this, &HelloApplication::greet);
}
```

```
void HelloApplication::greet()
{
    greeting_>setText("Hello there, " + nameEdit_>text());
}

Wt::WApplication *createApplication(const Wt::WEnvironment& env)
{
    return new HelloApplication(env);
}

int main(int argc, char **argv)
{
    return Wt::WRun(argc, argv, &createApplication);
}
```

You can build and run this application locally, if you want. All you need to do is compile the code above and link against the Wt library (`libwt`) and built-in HTTP server (`libwthttp`).

On UNIX-like systems, you could do the following:

```
$ g++ -o hello hello.cc -lwthttp -lwt
$ ./hello --docroot . --http-address 0.0.0.0 --http-port 9090
```

Let's start with the last part of the application, the `main()` function.

In the `main()` function, we call `WRun()` to start the application server. This method will return when the application server shuts down (by catching the KILL signal or Windows equivalent).

Inside `WRun()`

`WRun()` is actually a convenience function which creates and configures a `WServer` instance. If you want more control, for example if you have multiple “entry points”, or want to control the server starting and stopping, you can use the `WServer` API directly instead.

The `WRun()` function passes `argc` and `argv` (which for some connectors such as the built-in webserver configures the server), and accepts a function object as last argument. This function will be called when a new session is started and returns a new `WApplication` instance for that session. This function, in turn, has as input a `WEnvironment` object, and this object can be used to customize the application or authenticate the user.

The example instantiates four widgets into the application's root container: a text ("Your name, please?"), a line edit (`nameEdit_`), an empty text (`greeting_`) and a button (`button`). These three types of widgets are generic widgets provided by the library, and map directly on native HTML elements. In the hangman example below we will see how other custom and more specialized widgets are used in exactly the same way.

After we instantiated the widgets, we specify that we want to react to a click on the button. We connect the `greet()` method to the button's `clicked()` signal. Events propagate from one widget (the button) to other widgets or, as in this case, the application object, using signals. A glimpse at the reference documentation of a widget lists the signals that are exposed by a particular widget. For basic widgets, such as a push button, these are the typical mouse and keyboard

events. Higher-level widgets may advertise other events (for example, a calendar widget has a `selectionChanged()` signal), and you can add events to your own custom widgets.

How event propagation works

When an event is triggered by the user, the web browser communicates the target object and corresponding signal together with all form data to the web server (using a full page refresh or Ajax/WebSockets). At the server, after the request has been validated as genuine, form data such as line edit text is first processed to update the widget tree state. Then, the event is propagated by emitting the signal on the target widget, which propagates the event to all connected methods, such as in our example the `greet()` method. Modifications to the widget tree are tracked, and after the event has been handled these changes are reflected on the rendered HTML DOM tree, again using a full page refresh or incrementally using Ajax or WebSockets.

The most interesting thing about the implementation of the `greet()` method may be the code that is not there: no JavaScript to update the text using DOM manipulations or to re-render the page, no JavaScript code to post the event end line edit value using Ajax or WebSockets, no HTTP request decoding to interpret the line edit value or button event, and no security-related code. All this is handled instead by the library. While this could still be manageable for such a small example, imagine a situation where the page contains various form elements related to different tasks and thus managed by different widgets, and where during event propagation many unrelated widgets get updated.

4. Hangman

For those of you who forgot the game-play of hangman: the challenge is to guess a word. You can pick a letter, one at a time. If the word contains the chosen letter, it is indicated at the correct location(s). If the word does not contain the letter, you loose a turn and get one step closer to hanging. To win, you need to find the word before you die hanging. In our version, we will let the user choose a dictionary (English or Dutch), and we keep track of users and his high-scores.

4.1. A first custom widget

We first discuss the `HangmanWidget`, which is a custom widget that encapsulates the game itself: it allows a user to play one or more games. It does not deal with updating the user's score, instead it indicates score update events to other widget(s) using a signal.

The following screenshot shows how the widget is composed of different widgets:



The `HangmanWidget` combines widgets provided by the library (`WText`: `title_`, `statusText_`, `WComboBox`: `language_`), and three custom widgets (`WordWidget`: `word_`, `ImagesWidget`: `images_` and `LettersWidget`: `letters`). As you will see, custom widgets are instantiated and used in pretty much the same way as library widgets, including reacting to events generated by these widgets.

With this information, we can implement the class definition.

HangmanWidget declaration.

```
class HangmanWidget : public Wt::WContainerWidget
{
public:
    HangmanWidget(const std::string &name, Wt::WContainerWidget *parent = 0);

    Wt::Signal<int>& scoreUpdated() { return scoreUpdated_; }

private:
    Wt::WText          *title_;
    WordWidget         *word_;
    ImagesWidget       *images_;
    LettersWidget      *letters_;
    Wt::WText          *statusText_;
    Wt::WComboBox      *language_;
    Wt::WPushButton   *newGameButton_;

    Wt::Signal<int>    scoreUpdated_;
};
```

```
std::string      name_;
Dictionary       dictionary_;
int              badGuesses_;

void registerGuess(char c);
void newGame();
};
```

This widget is implemented as a specialized `WContainerWidget`. This is a typical choice for widgets that combine other widgets in a simple layout. Following a customary practice for widgets, we take an optional `parent` container as the last argument of the constructor. We declare one public method `scoreUpdated()`, which provides access to the signal that will be used to indicate changes to the user's score as he plays through games. A `Signal<int>` used here, indicates that an integer value will be passed as event information, and will reflect the score update itself (positive when the user wins, or negative when the user loses). Any function or object method with a signature compatible with the signal may be connected to it and will receive the score update.

The private section of the class declaration holds references to the contained widgets, and state related to the game.

The constructor implementation shows some resemblance with the hello world application we discussed earlier: widgets are instantiated and event signals are bound. There are some novelties however.

HangmanWidget constructor.

```
using namespace Wt;

HangmanWidget::HangmanWidget(const std::string &name, WContainerWidget *parent)
    : WContainerWidget(parent),
      name_(name),
      badGuesses_(0)
{
    setContentAlignment(AlignCenter);

    title_ = new WText(tr("hangman.readyToPlay"), this);

    word_ = new WordWidget(this);
    statusText_ = new WText(this);
    images_ = new ImagesWidget(MaxGuesses, this);

    letters_ = new LettersWidget(this);
    letters_->letterPushed().connect(this, &HangmanWidget::registerGuess);

    language_ = new WComboBox(this);
    language_->addItem(tr("hangman.englishWords").arg(18957));
    language_->addItem(tr("hangman.dutchWords").arg(1688));

    new WBreak(this);

    newGameButton_ = new WPushButton(tr("hangman.newGame"), this);
    newGameButton_->clicked().connect(this, &HangmanWidget::newGame);

    letters_->hide();
}
```

Wt supports different techniques to layout widgets that may be combined (see also the sidebar): namely widgets with CSS layout, HTML templates with CSS layout, or layout managers. Here, we chose to use the first approach, since we simply want to put everything vertically centered.

Other layout options

Although layout managers are a familiar concept in GUI development, CSS is king of layout in web development. Some things are hard to do with CSS though, in particular vertical centering or vertical size adjustments. It is for this purpose that layout managers have been added to Wt. These layout managers use JavaScript to compute the width and/or height of widgets based on dimensions of other widgets.

The `LetterWidget` exposes a signal that indicates that the user chose a letter. We connect a private method `registerGuess()` to it, which implements the game logic of dealing with a letter pick. Notice how this event handling for a custom widget is no different than reacting to an event from a push button, making that widget as much reusable as the widgets provided by the library (assuming you are in the business of hangman games).

To support internationalization, we use the `tr("key")` function (which is actually a method of `WWidget` which calls `WString::tr()`, to lookup a (localized) string given a key. This happens in a message resource bundle (see `WMessageResourceBundle`), which contains locale-specific values for these strings. Values may be `arg()` method of `WString`, as used for example for the "hangman.englishWords" string which has as actual English value "English words ({1} words)".

For completeness, we show below the rest of the `HangmanWidget` implementation.

HangmanWidget: game logic implementation.

```
void HangmanWidget::newGame()
{
    WString title(tr("hangman.guessTheWord"));
    title_>setText(title.arg(name_));

    language_>hide();
    newGameButton_>hide();

    Dictionary dictionary = (Dictionary) language_>currentIndex();
    word_>init(RandomWord(dictionary));
    letters_>reset();
    badGuesses_ = 0;
    images_>showImage(badGuesses_);
    statusText_>setText("");
}

void HangmanWidget::registerGuess(char c)
{
    bool correct = word_>guess(c);

    if (!correct) {
        ++badGuesses_;
        images_>showImage(badGuesses_);
    }

    if (badGuesses_ == MaxGuesses) {
```

```

WString status(tr("hangman.youHang"));
statusText_>setText(status.arg(word_>word()));

letters_>hide();
language_>show();
newGameButton_>show();

scoreUpdated_.emit(-10);
} else if (word_>won()) {
    statusText_>setText(tr("hangman.youWin"));
    images_>showImage(ImagesWidget::HURRAY);

    letters_>hide();
    language_>show();
    newGameButton_>show();

    scoreUpdated_.emit(20 - badGuesses_);
}
}

```

4.2. Unleashing (some of) Wt's power

Until now, we introduced a rather unique way to develop web applications, and a powerful building block for reuse – the widget. The next widget in the Hangman game that we will tackle, is one we've already used just earlier: the `ImagesWidget`. It illustrates an important aspect of the library that highly enhances the user experience for users with an Ajax session (which should be the majority of your users). One of the most appealing features of popular web applications like Google's Gmail and Google Maps is an excellent response time. Google may have spent quite some effort in developing client-side JavaScript and Ajax code to achieve this. With little effort – indeed almost automatically – you can get similar responsiveness 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 when Ajax or JavaScript support is not available. The `ImagesWidget` class, which we'll discuss next, contains some of these techniques. Hidden widgets are prefetched by the browser, ready to be displayed when `show()` is called.

ImagesWidget: implementation.

```

ImagesWidget::ImagesWidget(int maxGuesses, WContainerWidget *parent)
    : WContainerWidget(parent)
{
    for (int i = 0; i <= maxGuesses; ++i) {
        std::string fname = "icons/hangman";
        fname += boost::lexical_cast<std::string>(i) + ".png";
        WImage *theImage = new WImage(fname, this);
        hangmanImages_.push_back(theImage);

        theImage->hide();
    }

    WImage *hurray = new WImage("icons/hangmanhurray.jpg", this);
    hurray->hide();
    images_.push_back(hurray);

    showImage(HURRAY);
}

```

```
void ImagesWidget::showImage(int image)
{
    image(image_)->hide();
    image_ = index;
    image(image_)->show();
}

WImage *ImagesWidget::image(int index) const
{
    return index == HURRAY ? images_.back() : images_[index];
}
```

In the constructor, we meet one more basic widget from the library: `WImage`, which unsurprisingly corresponds to an image in HTML. The code shows how widgets corresponding to each state of the hangman example are created and added to our `ImagesWidget`, which specializes a `WContainerWidget`. Each image is also hidden – we’ll want to show only one at a time, and this is implemented in the `showImage()` function.

But why do we create these images only to hide them? A valid alternative could be to simply create the `WImage` that we want to show and delete the previous, or even better, to simply manipulate the image to point to another URL? The difference has to do with the response time, at least when Ajax is available. The library first renders and transfers information of visible widgets to the web browser. When the visible part of web page is rendered, in the background, the remaining hidden widgets are rendered and inserted in the DOM tree. Web browsers will also preload the images referenced by 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, and this no longer requires a new image to be loaded. An alternative implementation would have caused the browser to fetch the new image, making the application appear sluggish. Using hidden widgets is thus a simple and effective way to preload contents in the browser and improve the responsiveness of your application. Important to remember is that these hidden widgets do not compromise the application load time, since visible widgets are transferred first. A clear win-win situation thus.

4.3. Internal paths

Ignoring the login screen for a moment, then our application has two main windows: the game itself and the high scores. These are implemented by the `HangmanWidget` which we discussed earlier, and a `HighscoreWidget` (which we will not be discussing in this tutorial). Both are contained by a `WStackedWidget`, which is a container widget which shows only one of its contained children at a time (and which, in all honesty we should have used to implement the `ImagesWidget`, were it not that we wanted to explain a bit more about preloading of contents). Unless we do something about it, a Wt application presents itself as a single URL, and is thus a single-page web application. This is not necessarily bad, but, it may be better to support multiple URLs which allows a user to navigate within your application, bookmark particular “pages”, or put links to them. It also is instrumental to unlock the contents within your application to search engine robots. Wt provides you with a way to manage URLs which are subpaths of the application URL, which are called “internal paths”.



A Witty game: Hangman

Logged in as **guest** | [Logout](#)

Hall of fame

Congratulations! You are currently leading the pack.

RANK	USER	GAMES	SCORE	LAST GAME
1	guest	470	1498	16 seconds ago
2	x	30	304	2 days ago
3	ThePill	18	238	one week ago

Internal paths are best used in combination with anchors (provided by another basic widget, `WAnchor`). An anchor can point either to external URLs, to private application resources (which we'll not discuss but are useful for dynamic non-HTML contents), or to internal paths. When such an anchor is activated, this changes the application's URL (as one could expect), and the `internalPathChanged()` signal is emitted. Thus, to respond to an internal path change, we connect an event handler to this signal.

Internal paths: a perfect illusion

Normally, when a user navigates a link, the browser fetches the document linked to and replaces the current HTML page with the new page. This system of "full page refreshes" causes the browser to re-render the whole page each time, and is exactly what Ajax came to avoid. Using new features in HTML 5 (JavaScript History support), and falling back to tricks involving URL fragments in older browsers, Wt creates the illusion of navigating to a new page, but instead uses Ajax to update the page to reflect the URL change and navigation events. At the same time, search engines and plain HTML sessions will view your application using full page refreshes.

This is the implementation of the method that we connected:

HangmanGame: internal path handling.

```
void HangmanGame::handleInternalPath(const std::string &internalPath)
{
    if (session_.login().loggedIn()) {
        if (internalPath == "/play")
            showGame();
        else if (internalPath == "/highscores")
            showHighScores();
        else
            WApplication::instance()->setInternalPath("/play", true);
    }
}
```

Thus, if a user is logged in, we show the game when the path is `/play` and the high scores when the path is `/highscores`. For good form, we redirect all other paths to `/play` (which will end up triggering the same function again). In our game we make authentication (whether a user is currently logged in orthogonal to the internal paths): in this way a user may arrive at the game using any internal path, log in, and automatically proceed with the function for that internal path.

You may imagine that this is what you want in a complex application: the login function should not prevent the user from directly going to a certain “page” within your application.

We did not discuss other parts of the hangman game example application: namely how user scores are stored, and the authentication system. Database access is implemented using a `Wt::Dbo`, which is a C++ ORM that comes with Wt. This tutorial [<http://www.webtoolkit.eu/wt/doc/tutorial/dbo.html>] introduces the database layer. The authentication module, `Wt::Auth`, as used in this example, is introduced here [http://www.webtoolkit.eu/wt/blog/2011/11/14/an_introduction_to_wt_auth].

5. Summary

In this tutorial we provided you with the basic techniques for creating web applications using Wt, from small to more complex. While a tutorial is no place to discuss a real-life complex application, with a small leap of faith, it should be clear that the same techniques of creating an application using widgets as building blocks, provides an effective way to manage complexity (and evolving features), while freeing the application developer of many technical aspects and quirks associated with the web platform. Because of the many similarities between Wt and other GUI toolkits, developers can treat the web browser in many aspects as just another GUI platform.

In this tutorial we touched on many important Wt features. But Wt adds much more to your toolbox that weren't mentioned: file uploads, dynamic resources, painting, tree and table views and their models, a charting library, animation effects, WebSockets, built-in security measures, authentication, etc... For more information, please see the online documentation [<http://www.webtoolkit.eu/wt/documentation>].