# The LiveCode Lab

## Quick Start Guide

# The LiveCode Lab

## Quick Start Guide



July 2014 Edition


Built with LiveCode from RunRev Ltd.

**Address:** Scott McDonald PC Services
PO Box 139, Newtown, 2042
New South Wales, Australia
**Facsimile:** +612-8580-5726
**Website:** http://thelivecodelab.com
**Email:** scott@thelivecodelab.com

# Table of Contents

# Introduction

The LiveCode Lab is an open-source hosting platform for building web apps using LiveCode server.

This type of web app requires what is called *server-side* coding, which works differently from software that runs on your local device, either a desktop, laptop or mobile device.

This means that the code you write executes (runs) on a remote server, and sends HTML and CSS as a web page to the user's browser to show the current state (screen) of the app. So your code is never visible to the user and your web app can run in almost any browser.

In one sense, a web app looks like a website, but because the code is running on a server, a web app is interactive and can do actions that are not possible with a regular website.

With an account at *The LiveCode Lab* (TLCL) you can do server-side coding from any web browser with an active internet connection. You can run and test your app actively during development. And when you think it is ready, you can make the app public for anyone to use.

If you have not done server-side coding before, there are 2 major differences from writing a program that runs on the desktop or another local device.

1. Regular variables are not persistent (they do not keep their values) between each refresh of the web app page.
2. The user interface needs to be re-designed to use HTML forms and other elements.

TLCL uses the LiveCode language for the processing of the web app. This language is designed to be easy to use, and as a high level language, you can do a lot with small amounts of code.

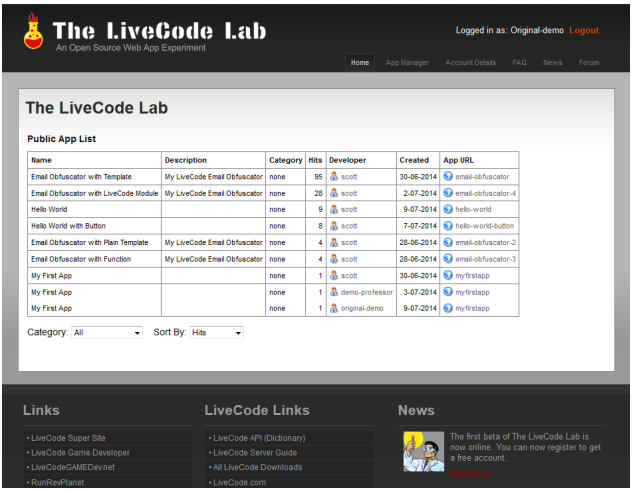
If you have used LiveCode before, the two points above will require you to rethink the way you design your apps. A full answer to working within these constraints is beyond the scope of this Quick Start Guide, but in summary a combination of

- Post and URL data
- MySQL
- Cookies
- Server-side flat files

can be used to store data that must be kept during the running of your app.

For the user interface, your creativity is required to re-organise your app to work within the limitations of what can be done in a web browser.

The *How It Works* chapter has an overview of how a web app works.



# How It Works

TLCL is based around the idea of HTML templates with “placeholders” ready for content that you produce to display as each page of your app. You write code that produces text, or more HTML, that is inserted into different areas (placeholders) on the template.

This page is then “served-up” (sent to) the user’s browser where it appears. Normally the page has button or link elements to allow the user to interact with the app. When one of these it clicked on, a request is sent to the server for an updated page which is produced from your code.

Below is the process of running and interacting with a TLCL app.

1. A user types the URL at the TLCL for your app.
2. The TLCL server executes the code you have created in the index.lc file.

3. Your code:
   (a) Opens a HTML template.
   (b) Does processing to add information and data to the areas on the template.
   (c) Returns a complete HTML page to the TLCL server.

4. The TLCL server sends the page to the user’s browser for display.

5. The user interacts with your app with a click on a link or button.
6. The browser sends this information to the TLCL server.

7. The TLCL server executes the code in either the index.lc file again, or in a different LiveCode file, depending on the structure of your app.

8. Your code:
   (a) Opens the same or a different HTML template.
   (b) Does more processing to add new information and data to the areas on the template.
   (c) Returns a complete HTML page to the TLCL server.

9. The TLCL server sends the page to the user’s browser for display.

Then the process goes back step 6, with the steps from 6 to 9 being repeated for as long as the user is interacting with your app.

This interaction is what makes the app work, and it is the combination of the HTML and the your coding that is processed in steps 3(b) and 8(b) that defines what it does.

- You are not limited to the small number of HTML templates included with TLCL. You can add and create your own templates as modules of your app.

To help with the putting of information and data onto a HTML template, TLCL has a Runtime Library (RTL) that includes these commands:

```
wasPut
wasPutP
wasPutBR
wasRaw
wasSetTemplate
wasSetActiveArea
wasSetAppendArea
wasSetAreaText
```

These 8 commands (plus the regular LiveCode Server API) is all you need to make a web app in TLCL. The RTL is documented in the *TLCL API* chapter.

# Hello World

Here is the complete code for a simple Hello World app.

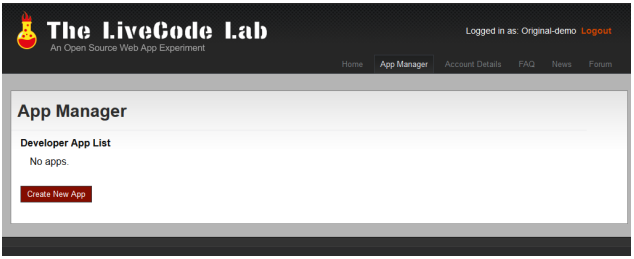
```
wasPut "Hello World!"
```

Even if you can deduce how it works, following the steps in this chapter is a good introduction to understanding how TLCL is organised.

The wasPut command puts text into the active area of the template. In this one line app, a template has not been set, so the text will appear on a plain white page in the browser.

Here are the steps to make this app.

1. Select the App Manager.
2. Click on Create New App.

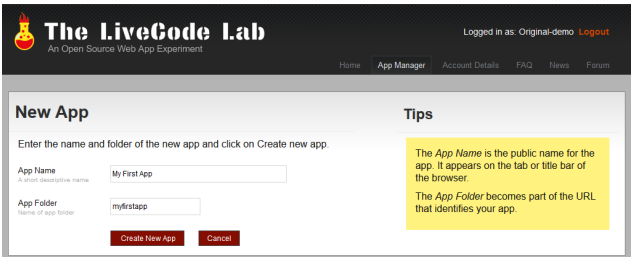



3. Enter the App Name and App Folder. For example, you could enter *My First App* and *myfirstapp*.

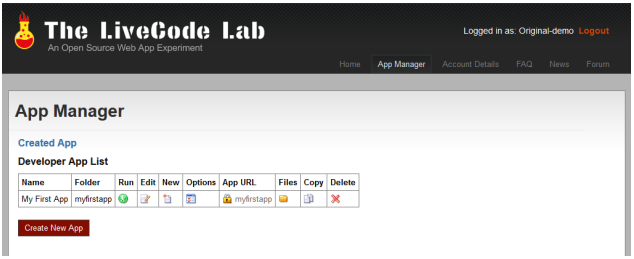
   The App Name is the public name for the app. It appears on the Tab or Title bar of the browser. The App Folder becomes part of the URL that identifies your web app.

   Note, your User Name becomes part of the URL that identifies your web app, so the App folder does not need to be unique across all the web apps in TLCL.

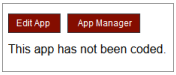
4. Click on Create New App.




My First App is now in your Developer App List.

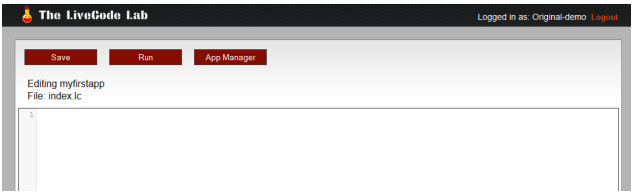



5. Click on the Run icon and see what happens. The *This app has not been coded* message appears.



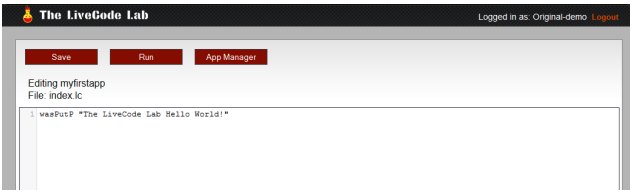

Note the Edit App and App Manager buttons. When running your app from the App Manager, these buttons always appear at the top of the page to allow you to return to TLCL.

6. To edit the code of the new My First App, you can click on Edit App, or you could click on App Manager, and then the Edit icon. Do one of these now to show TLCL editor.

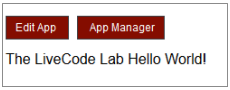


7. Type the following into the editor field.

```
wasPutP "The LiveCode Lab Hello World!"
```




8. Click on the Run button.




Your first web app.

9. Before leaving the page in your browser, right click on the page and choose the View Source (or equivalent) command in the browser window.

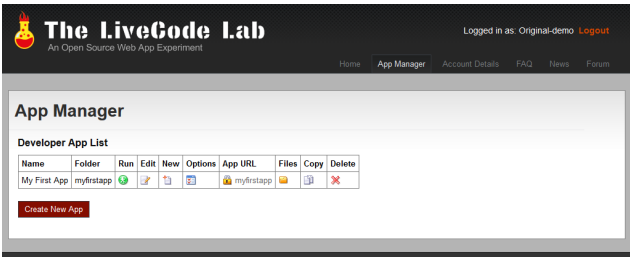
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html lang="EN" dir="ltr">
<head>
<title>My First App</title>
<meta http-equiv="content-type" content="text/xml; charset=utf-8" />
<meta name="generator" content="The LiveCode Lab 1.0.0.0">
<meta name="robots" content="noindex,nofollow">
<link rel="stylesheet" type="text/css" href="../../wasTemplates/main.css" />
</head>
<body>
<!-- generated html, not in public apps -->
<p><span class='redbuttonlink'><a href='../../appedit.tlcl?wasdebug=original-
demo,12345678,myfirstapp'> Edit App </a></span>  <span
class='redbuttonlink'><a href='../../appmanager.tlcl?wasdebug=original-
demo,12345678,myfirstapp'> App Manager </a></span></p>
<!-- end generated html -->

<p>The LiveCode Lab Hello World!
</p>
</body>
</html>
```

This shows the HTML from your code visible at the end.

```
<p>The LiveCode Lab Hello World!
</p>
```

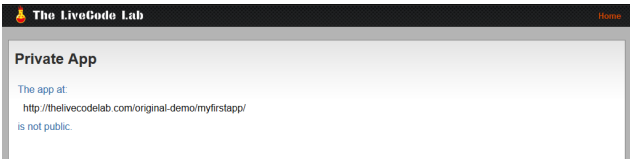
The rest is generated by TLCL.

10. Close the pop-up source window and click on the App Manager button.




The Lock icon in the App URL column means your app is not public. A new app is always private. This allows you to do development without worrying about visitors trying to use it before you are ready.

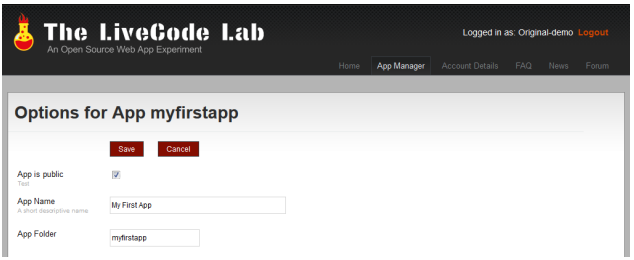
11. Click on the *myfirstapp* link in the App URL column.




This is what is shown when a web app is not public. As the developer of the app you can always run a private app from the Run icon in the App Manager, but no-one else can.

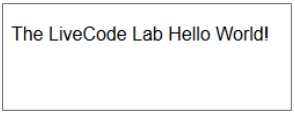
You can make the app public by changing the App Options.

12. Click on the icon in the Options column.




13. Turn on the *App is public* check box and click on Save. The App URL column now shows the globe icon to indicate the app is public.

14. Click on the *myfirstapp* link again. This is your app as the world sees it. The Edit App and App Manager buttons are missing which is why the link opens in a new tab (or window) in your browser.




    If you want to share your app, you can copy the myfirstpp link and post or send it to anyone.

This completes your first “Hello World” app. It is not a proper web app, but the process of making it introduces the basic features of TLCL. The *Hello World with Button* chapter explains more about how you make an app with interaction.

# Hello World with Button

The one line program from the *Hello World* chapter doesn't allow for any interaction. When run, a message is displayed and that is all. To be a web app there needs to be a way for the user to interact with the program. Here is a second Hello World app that includes a button. Once you know how a button works you can make something interactive.

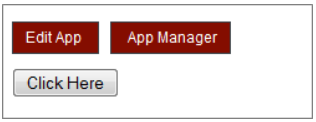
This chapter does not detail the procedure for making the app one step at a time. It assumes you know how to do this from the *Hello World* chapter. If you are not sure about the steps, please refer to that chapter.

Make a new app named *My Second App* with a folder of *mysecondapp*.

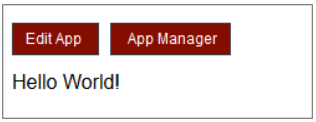
Type, or paste, the code below into the editor for the main index.lc file.

```
if $_POST["butnOK"] is empty then
  wasPut "<form action='^wasindex.lc' method='POST'>"
  wasPut "<input type='submit' name='butnOK' value='Click Here' />"
  wasPut "</form>"
else
  wasPut "Hello World!"
end if
```

Run it and see how it works.




After clicking on the button.




This program includes a HTML form and a conditional *if* statement. The $_POST variable is a special LiveCode array that is only for server-side programming. It contains values when a form as been posted.

When you first run mysecondapp, $_POST is empty because a form has not been submitted.

So this expression evaluates to true.

```
$_POST["butnOK"] is empty
```

Then these statements are executed.

```
wasPut "<form action='^wasindex.lc' method='POST'>"
wasPut "<input type='submit' name='butnOK' value='Click Here' />"
wasPut "</form>"
```

These construct a HTML form that uses the POST method and includes a submit button.

```
wasPut "<form action='^wasindex.lc' method='POST'>"
```

The form action attribute uses a special TLCL “placeholder”. In TLCL, a placeholder always begins with the ^ and is followed by a name or number. The placeholder is then substituted with other text by TLCL when your program runs.

Here the ^wasindex is used because when running an app from the App Manager during development, the executed main file is not named index.lc. (This is necessary to allow you to run your app even when it is not public.)

Next is this line.

```
wasPut "<input type='submit' name='butnOK' value='Click Here' />"
```

It puts a submit button on the form with the name of butnOK. Which was the same name checked earlier in the $_POST array. Then the closing tag for the form is output.

```
wasPut "</form>"
```

The remaining line in the program is in the *else* part of the if statement and it not executed. This results in the Click Here button appearing in the browser.

As in the *Hello World* chapter, if you examine the source in your browser, there is no sign of LiveCode anywhere. All the LiveCode code executes on the server, and your app user only sees HTML.

Once the page is served up, there is no code running on the server. The execution of the app ends. (The executed code may not even be in the server memory anymore. It could have been replaced by other programs are running on the server.)

But because your app contains a form, when the Click Here button is clicked on, your program runs again.

This time $_POST is not empty because a form has been submitted with the POST action. The $_POST["butnOK"] element of the array contains *Click Here,* the value of the submit button.

So the *else* part of the *if* statement executes.

```
wasPut "Hello World!"
```

Showing Hello Word!

This example illustrates a key point about TLCL.

- TLCL does not hide the mechanics of HTML forms and the fundamentals of creating apps that run in a browser.

Experimenting and developing in TLCL is the opposite of using a framework or library that “does magic” and hides the workings of a web app. By using TLCL, you will gain a solid foundation and understanding of how server-side coding and web apps work.

# HTML Templates

TLCL is based around the idea of HTML templates that have area placeholders for the content that your app produces. You write code that produces the text (or more HTML) that is inserted into different areas on the template. The commands in the *TLCL API* chapter are all about putting text onto a template, for output as the page of your app.

- A simple app does not need to use a template. In that case, the text from the wasPut commands is inserted between the body tags of a blank white page.

A template has the following parts:

1. optional <body> </body> tags
2. optional css section
3. a main section that includes the html and area placeholders.

- A template *must not* include the <html> </html> tags and the <head> </head> tags, or the contents between the head tags. These are all automatically generated by TLCL.

Below is a simplified version of the *line-input-output* included in TLCL. This template includes the <body> tag because it has a background image, but if the <body> has no styles then the <body> </body> tags can be omitted.

This template includes style sheet data inline as part of the template.

```
<body>
<style type="text/css">
body {
  background:#999999 url(../../wasTemplates/tlcl-main-bg.png) repeat-x top;
  font: Arial, Helvetica, sans-serif;
  padding:0;
  margin:0;
  color:#000000;
}
#bodycontainer {
  margin:0;
  padding:0;
  background:url(../../wasTemplates/tlcl-container-bg.png) top repeat-x;
}
.contentcontainer {
  width:1024px;
  margin:0 auto;
  padding:0;
}
.header {
  width:1024px;
  margin:0 auto;
  padding:0;
  border-top: 1px solid #000;
}
```

```
</style>

<div id="bodycontainer">
  <div class="contentcontainer">
    <div class="contentbackground">
      <h1>^name</h1>
      ^description
      <form action="^wasindex.lc" method="POST">
        <input type="text" name="input"  style="width:^inwidth;" value="^input" />
        <p><input type="submit" name="process" value=" Process " />
        </p>
        <input type="text" name="output" style="width:^outwidth;" value="^output" />
      </form>
      ^instruction
      <p> 
      </p>
    </div>
  </div>
</div>
</body>
```

This template has 8 placeholders.

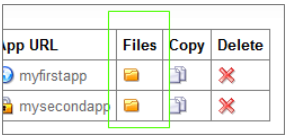
```
^description
^name
^wasindex
^inwidth
^input
^outwidth
^output
^instruction
```

Except for the ^wasindex placeholder, which is a special type described in the *Hello World With Button* chapter, you should put text into each one of these areas. Otherwise, the placeholder name will be visible on the page of your app.

If you do not require an area on a template, you should set it to empty to stop it appearing on the page. For example, in the template above when no instructions are needed, you can use this line.

```
wasSetAreaText "instruction",empty
```

You can make your own templates by making a new HTML module from the App Manager. In that case, the template is stored in your web app folder (not in wasTemplates) and you can use the *File Manager* in the App Manager to upload image and other resources needed to support the HTML of the template and app.



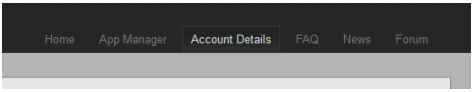

- The *stock-trader* demo app is a good example of an app that uses custom templates.

# Using a MySQL Database

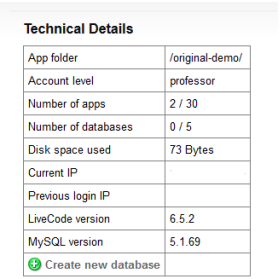
The *Doctorate* and *Professor* accounts at TLCL can create and use MySQL databases in your app.

To make a database.

1. Select the Account Details.




2. Click on Create New Database in the Technical Details table.



**Technical Details**

| | |
|---|---|
| App folder | /original-demo/ |
| Account level | professor |
| Number of apps | 2 / 30 |
| Number of databases | 0 / 5 |
| Disk space used | 73 Bytes |
| Current IP | |
| Previous login IP | |
| LiveCode version | 6.5.2 |
| MySQL version | 5.1.69 |
| Create new database | |

It may take a few seconds for the database is created.

A database name and password is then shown in the Technical Details table. These can be used in your app with the LiveCode *revOpenDatabase* command to open a database.

In TLCL it is important to use the correct combination of parameters to make it work. You must use these value for the parameters.

| **Parameter** | **Value** |
|---|---|
| databaseType | "mysql" |
| host | "localhost" |
| databaseName | the database name from the Technical Details enclosed in quotes |
| userName | the same database name from the Technical Details in quotes |
| password | the database password from the Technical Details in quotes |
| socket | "/var/lib/mysql/mysql.sock" |

The other parameters should be empty. Note how the userName is the database name repeated.

Here is an example, shown over two lines here, but would be a single line.

```
put revOpenDatabase("mysql","localhost","dbname","dbname",
         "dbpassword",,"/var/lib/mysql/mysql.sock") into dbID
```

This puts the database into a variable named *dbID* for use in other LiveCode commands that access the database.

# TLCL API

This section documents TLCL API. The API is 8 commands that assist with the creation of the HTML that is output for each page of your app.

Each entry has a short description of the action of the command and a list of parameters. Lastly, general comments about the command are included.

- These commands are not an application framework. Except for basic buffering of the HTML of your app, TLCL does not simplify or hide the coding necessary to create a web app.

---

**wasPut** **command**

```
command wasPut pText[,pSub1,pSub2,etc]
```

Appends text to the active area on the HTML template, with substitution of optional additional parameters into the text.

**Parameters**
pText: the text to append to the active area on the template
pSub1: optional text to substitute into pText

**Comment**
Apostrophes in pText is replaced with ". To insert an apostrophe precede it with a backslash \. The text in pText can optionally contain numbered placeholders that begin with ^, which are substituted with the optional parameters.

For example, if wasPut is called with this line:

```
wasPut "Your name is \'^1\' and you are ^2.",username,userage
```

And username and userage are variables containing *Asuka* and *15* respectively, the actual text put into the HTML template, is:

```
Your name is "Asuka" and you are 15.
```

In this example, you can see the apostrophe and placeholder substitution in action. You can have as many placeholder as you need, but there should be a parameter to wasPut for each one.

Note, it is the number after the ^ that refers to the number of the parameter. In other words the placeholders do not need to appear in the same order as the parameters,

although often it aids readability if they are.

For example, if wasPut is called with this line:

```
wasPut "My two best friends are ^2 and ^1.","Shinji","Rei"
```

Here variables haven't been used, so you wouldn't need to use placeholders in this case, but that is just an example about the ordering of placeholders.

```
My two best friends are Rei and Shinji.
```

Note how Shinji is still replaces ^1. This is because it is the placeholder number that is significant, not the order from left to right of the placeholders in the text.

---

## wasPutBR command

**command** wasPutBR pText[,pSub1,pSub2,etc]

Appends text to the active area followed by a HTML line break on the HTML template, with substitution of optional additional parameters into the text.

#### Parameters

pText: the text to append to the active area on the template
pSub1: optional text to substitute into pText

#### Comment

The method of substitution used in this command is identical to the wasPut command. Refer to the *wasPut* entry in this section for more details. The only additional feature is the appending of the HTML <br /> tag.

For example, if wasPutBR is called with this line:

```
wasPutBR "Watch out for impact."
```

The text put into the HTML template, is:

```
Watch out for impact.<br />
```

## wasPutP command

```
command wasPutBR pText[,pSub1,pSub2,etc]
```

Appends text to the active area enclosed by HTML paragraph tags on the HTML template, with substitution of optional additional parameters into the text.

#### Parameters
pText: the text to append to the active area on the template
pSub1: optional text to substitute into pText

#### Comment
The method of substitution used in this command is identical to the wasPut command. Refer to the *wasPut* entry in this section for more details. The only additional feature is the enclosing with the HTML   tags.

For example, if wasPutP is called with this line:

```
wasPutP "Watch out for impact."
```

The text put into the HTML template, is:

```
<p>Watch out for impact.</p>
```

## wasPutRaw command

```
command wasPutRaw pText
```

Appends text to the active area on the HTML template.

#### Parameters
pText: the text to append to the active area on the template

#### Comment
This command puts text into the HTML template without any substitution occurring. What you pass in the pText parameter is what appears on the app page.

## wasSetActiveArea command

```
command wasSetActiveArea pArea
```

Sets the area of the HTML template that the wasPut commands put text into.

#### Parameters
pArea: the name of the area placeholder on the template. Do not include the ^

#### Comment
A template can have area placeholders that begin with ^. This is similar to the placeholders of the wasPut command, but an area placeholder does not need to be numeric and can have a proper name. After calling wasSetActiveArea, the wasPut commands continue to put text into that area of the template, until wasSetActiveArea is called with a different area name.

- If you switch to a different area with wasSetActiveArea, and then later call wasSetActiveArea with an area already used, the wasPut commands *replace* what is already in the area. If you need to add to existing text in an area, call wasSetAppendArea instead.

If the area is not found the diagnostic error message:

```
wasSetActiveArea: Cannot find area
```

is appended to the HTML output of the app.

## wasSetAppendArea command

```
command wasSetAppendArea pArea
```

Sets the area of the HTML template that the wasPut commands append text to.

#### Parameters
pArea: the name of the area placeholder on the template

#### Comment
The areas used in this command work in a similar way to the wasSetActiveArea command. Refer to the *wasSetActiveArea* entry in this section for more details about area placeholders. This command is different because the wasPut command append text to the area, keeping any existing text from previous calls to wasSetActiveArea and wasSetAppendArea.

If the area is not found the diagnostic error message:

```
wasSetAppendArea: Cannot find area
```

is appended to the HTML output of the app.

---

## wasSetTemplate command

**command** wasSetTemplate pName

Sets the HTML template used for the page of the web app.

#### Parameters
pName: the file name of the template

#### Comment
A template is a file with a .html extension in the folder of your app, or in the TLCL templates folder. This command looks in your app folder first, and if a file matching the name exists it is set as the template. If the file does not exist the TLCL templates folder is checked for a matching file. If the template is not found the diagnostic error message:

```
wasSetTemplate: Cannot find template
```

is appended to the HTML output of the app.

■■■■