# Prado Quick Start

Qiang Xue

January 22, 2006

Compiled by Wei Zhuo

# Contents

# Preface

Prado quick start doc

# Chapter 1

# Getting Started

## 1.1 Welcome to the PRADO QuickStart Tutorial

This QuickStart tutorial is meant to get you quickly started to build your own Web applications based on PRADO.

## 1.2 What is PRADO?

PRADO stands for **P**HP **R**apid **A**pplication **D**evelopment **O**bject-oriented.

PRADO is a component-based and event-driven programming framework for developing Web applications in PHP 5.

PRADO stipulates a protocol of writing and using components to construct Web applications. A component is a software unit that is self-contained and can be reused with trivial customization. New components can be developed by either inheriting or composing from existing ones. Component-based programming brings great freedom in teamwork anf offers the ultimate extensibility and maintenability to the code. PRADO implements a set of elementary components that represent commonly used Web elements, such as input field, checkbox, dropdown list, etc.

PRADO implements an event-driven programming scheme that allows delegation of extensible behavior to components. End-user activities, such as clicking on a submit button, changing the

content in an input field, are captured as server events. Methods or functions may be attached to these events so that when the events happen, they are invoked automatically to respond to the events. Compared with the traditional Web programming in which developers have to deal with the raw POST or GET variables, event-driven programming helps developers better focus on the necessary logic and reduces significantly the low-level repetitive coding.

Developing a PRADO Web application mainly involves instantiating prebuilt component types, configuring them by setting their properties, responding to their events by writing handler functions, and composing them into pages for the application. It is very similar to RAD toolkits, such as Borland Delphi and Microsoft Visual Basic, that are used to develop desktop GUI applications.

## 1.3 Installing PRADO

If you are viewing this page from your own Web server, you are already done with the installation. The instructions at the end of this page, however, may still be useful for you to troubleshoot issues happened during your development based on PRADO.

Installation of PRADO is very easy. Follow the following steps,

- Go to pradosoft.com to grab a latest version of PRADO.
- Unpack the PRADO release file using *unzip* on Linux or *winzip* on Windows. A directory named *prado* will be created under the working directory.
- Copy or upload everything under the *prado* directory to the DocumentRoot directory (or a subdirectory) of the Web server.
- Your installation of PRADO is done and you can start to play with the demo applications included in the PRADO release via URL *http://web-server-address/demos/*. This QuickStart Tutorial is one of such applications.

If you encounter any problems with the demo applications, please use the PRADO requirement checker script to check first if your server configuration fullfils the conditions required by PRADO.

The minimum requirement by PRADO is that the Web server support PHP 5. PRADO has been tested with Apache Web server on Windows and Linux. Highly possibly it may also run on other platforms with other Web servers, as long as PHP 5 is supported.

# Chapter 2

# Fundamentals

## 2.1 Architecture

PRADO is primarily a presentational framework, although it is not limited to be so. The framework focuses on making Web programming, which deals most of the time with user interactions, to be component-based and event-driven so that developers can be more productive. The following class tree depicts the skeleton classes provided by PRADO,

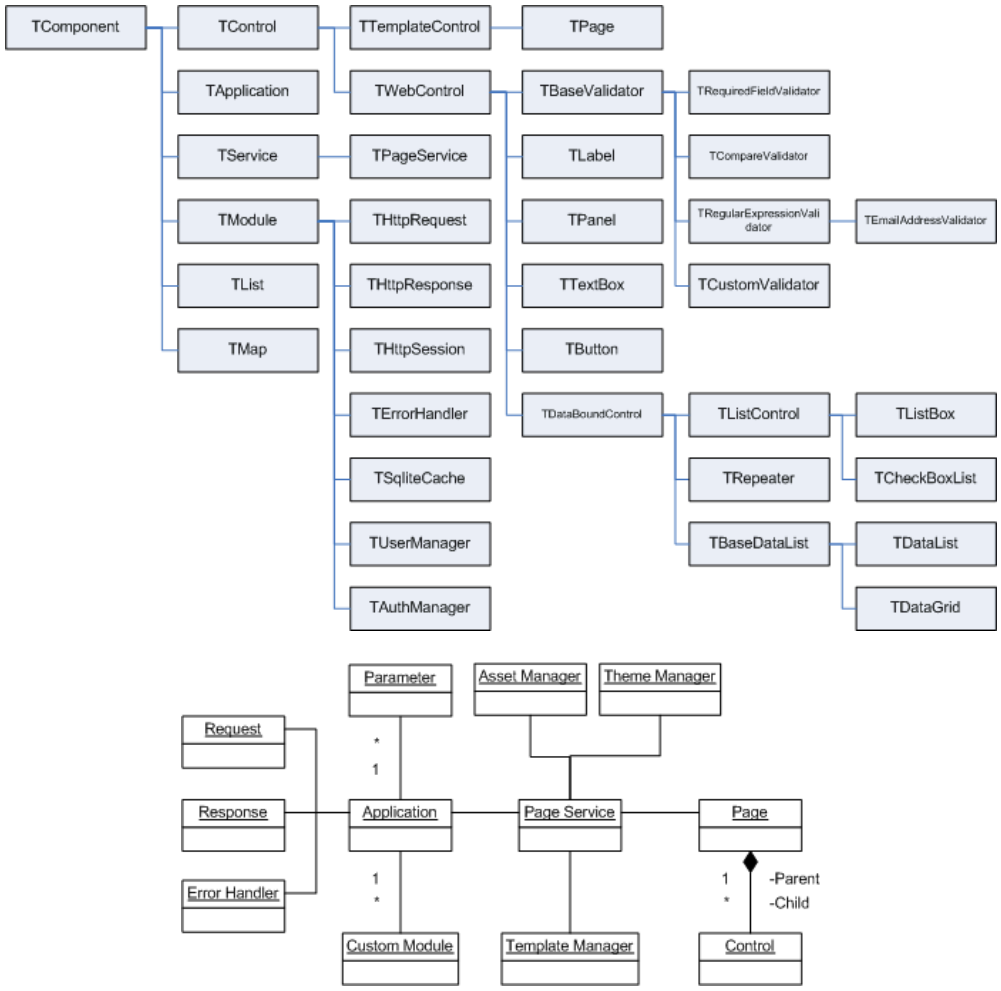
When a PRADO application is processing a page request, its static object diagram can be shown as follows,

## 2.2 Components

A component is an instance of `TComponent` or its child class. The base class `TComponent` implements the mechanism of component properties and events.

### 2.2.1 Component Properties

A component property can be viewed as a public variable describing a specific aspect of the component, such as the background color, the font size, etc. A property is defined by the existence




of a getter and/or a setter method in the component class. For example, in `TControl`, we define its `ID` property using the following getter and setter methods,

```
class TControl extends TComponent {
    public function getID() {
        ...
    }
    public function setID($value) {
        ...
    }
}
```

To get or set the `ID` property, do as follows, just like working with a variable,

```
$id = $component->ID;
$component->ID = $id;
```

This is equivalent to the following,

```
$id = $component->getID();
$component->setID( $id );
```

A property is read-only if it has a getter method but no setter method. Since PHP method names are case-insensitive, property names are also case-insensitive. A component class inherits all its ancestor classes' properties.

#### Subproperties

A subproperty is a property of some object-typed property. For example, `TWebControl` has a `Font` property which is of `TFont` type. Then the `Name` property of `Font` is referred to as a subproperty (with respect to `TWebControl`).

To get or set the `Name` subproperty, use the following method,

```
$name = $component->getSubProperty('Font.Name');
$component->setSubProperty('Font.Name', $name);
```

This is equivalent to the following,

```
$name = $component->getFont()->getName();
$component->getFont()->setName( $name );
```

### 2.2.2 Component Events

Component events are special properties that take method names as their values. Attaching (setting) a method to an event will hook up the method to the places at which the event is raised. Therefore, the behavior of a component can be modified in a way that may not be foreseen during the development of the component.

A component event is defined by the existence of a method whose name starts with the word `on`. The event name is the method name and is thus case-insensitve. For example, in `TButton`, we have

```
class TButton extends TWebControl {
    public function onClick( $param ) {
        ...
    }
}
```

This defines an event named `OnClick`, and a handler can be attached to the event using one of the following ways,

```
$button->OnClick = $callback;
$button->OnClick->add( $callback );
$button->OnClick[] = $callback;
$button->attachEventHandler( 'OnClick' , $callback );
```

where `$callback` refers to a valid PHP callback (e.g. a function name, a class method `array($object,'method')`, etc.)

### 2.2.3 Namespaces

A namespace refers to a logical grouping of some class names so that they can be differentiated from other class names even if their names are the same. Since PHP does not support namespace intrinsically, you cannot create instances of two classes who have the same name but with different definitions. To differentiate from user defined classes, all PRADO classes are prefixed with a letter 'T' (meaning 'Type'). Users are advised not to name their classes like this. Instead, they may prefix their class names with any other letter(s).

A namespace in PRADO is considered as a directory containing one or several class files. A class may be specified without ambiguity using such a namespace followed by the class name. Each namespace in PRADO is specified in the following format,

```
PathAlias.Dir1.Dir2
```

where `PathAlias` is an alias of some directory, while `Dir1` and `Dir2` are subdirectories under that directory. A class named `MyClass` defined under `Dir2` may now be fully qualified as `PathAlias.Dir1.Dir2.MyClass`.

To use a namespace in code, do as follows,

```
Prado::using('PathAlias.Dir1.Dir2.*');
```

which appends the directory referred to by `PathAlias.Dir1.Dir2` into PHP include path so that classes defined under that directory may be instantiated without the namespace prefix. You may also include an individual class definition by

```
Prado::using('PathAlias.Dir1.Dir2.MyClass');
```

which will include the class file if `MyClass` is not defined.

For more details about defining path aliases, see application configuration section.

### 2.2.4 Component Instantiation

Component instantiation means creating instances of component classes. There are two types of component instantation: static instantiation and dynamic instantiation. The created components are called static components and dynamic components, respectively.

#### Dynamic Component Instantiation

Dynamic component instantiation means creating component instances in PHP code. It is the same as the commonly referred object creation in PHP. A component can be dynamically created using one of the following two methods in PHP,

```
$component = new ComponentClassName;
$component = Prado::createComponent('ComponentType');
```

where `ComponentType` refers to a class name or a type name in namespace format (e.g. `System.Web.UI.TControl`). The second approach is introduced to compensate for the lack of namespace support in PHP.

#### Static Component Instantiation

Static component instantiation is about creating components via configurations. The actual creation work is done by the PRADO framework. For example, in an application configuration, one

can configure a module to be loaded when the application runs. The module is thus a static component created by the framework. Static component instantiation is more commonly used in templates. Every component tag in a template specifies a component that will be automatically created by the framework when the template is loaded. For example, in a page template, the following tag will lead to the creation of a `TButton` component on the page,

```
<com:TButton Text="Register" />
```

## 2.3 Controls

A control is an instance of class `TControl` or its subclass. A control is a component defined in addition with user interface. The base class `TControl` defines the parent-child relationship among controls which reflects the containment relationship among user interface elements.

### 2.3.1 Control Tree

Controls are related to each other via parent-child relationship. Each parent control can have one or several child controls. A parent control is in charge of the state transition of its child controls. The rendering result of the child controls are usually used to compose the parent control's presentation. The parent-child relationship brings together controls into a control tree. A page is at the root of the tree, whose presentation is returned to the end-users.

The parent-child relationship is usually established by the framework via templates. In code, you may explicitly specify a control as a child of another using one of the following methods,

```
$parent->Controls->add($child);
$parent->Controls[]=$child;
```

where the property `Controls` refers to the child control collection of the parent.

### 2.3.2 Control Identification

Each control has an `ID` property that can be uniquely identify itself among its sibling controls. In addition, each control has a `UniqueID` and a `ClientID` which can be used to globally identify

the control in the tree that the control resides in. `UniqueID` and `ClientID` are very similar. The former is used by the framework to determine the location of the corresponding control in the tree, while the latter is mainly used on the client side as HTML tag IDs. In general, you should not rely on the explicit format of `UniqueID` or `ClientID`.

### 2.3.3 Naming Containers

Each control has a naming container which is a control creating a unique namespace for differentiating between controls with the same `ID`. For example, a `TRepeater` control creates multiple items each having child controls with the same `ID`s. To differentiate these child controls, each item serves as a naming container. Therefore, a child control may be uniquely identified using its naming container's `ID` together with its own `ID`. As you may already have understood, `UniqueID` and `ClientID` rely on the naming containers.

A control can serve as a naming container if it implements the `INamingContainer` interface.

### 2.3.4 ViewState and ControlState

HTTP is a stateless protocol, meaning it does not provide functionality to support continuing interaction between a user and a server. Each request is considered as discrete and independent of each other. A Web application, however, often needs to know what a user has done in previous requests. People thus introduce sessions to help remember such state information.

PRADO borrows the viewstate and controlstate concept from Microsoft ASP.NET to provides additional stateful programming mechanism. A value storing in viewstate or controlstate may be available to the next requests if the new requests are form submissions (called postback) to the same page by the same user. The difference between viewstate and controlstate is that the former can be disabled while the latter cannot.

Viewstate and controlstate are implemented in `TControl`. They are commonly used to define various properties of controls. To save and retrieve values from viewstate or controlstate, use following methods,

```
$this->getViewState('Name',$defaultValue);
$this->setViewState('Name',$value,$defaultValue);
$this->getControlState('Name',$defaultValue);
```

```
$this->setControlState('Name',$value,$defaultValue);
```

where `$this` refers to the control instance, `Name` refers to a key identifying the persistent value, `$defaultValue` is optional. When retrieving values from viewstate or controlstate, if the corresponding key does not exist, the default value will be returned.

## 2.4 Pages

Pages are top-most controls that have no parent. The presentation of pages are directly displayed to end-users. Users access pages by sending page service requests.

Each page must have a template file. The file name suffix must be `.page`. The file name (without suffix) is the page name. PRADO will try to locate a page class file under the directory containing the page template file. Such a page class file must have the same file name (suffixed with `.php`) as the template file. If the class file is not found, the page will take class `TPage`.

### 2.4.1 PostBack

A form submission is called *postback* if the submission is made to the page containing the form. Postback can be considered an event happened on the client side, raised by the user. PRADO will try to identify which control on the server side is responsible for a postback event. If one is determined, for example, a `TButton`, we call it the postback event sender which will translate the postback event into some specific server-side event (e.g. `Click` and `Command` events for `TButton`).

### 2.4.2 Page Lifecycles

Understanding the page lifecycles is crucial to grasp PRADO programming. Page lifecycles refer to the state transitions of a page when serving this page to end-users. They can be depicted in the following statechart,

## 2.5 Modules

A module is an instance of a class implementing the `IModule` interface. A module is commonly designed to provide specific functionality that may be plugged into a PRADO application and shared by all components in the application.

PRADO uses configurations to specify whether to load a module, load what kind of modules, and how to initialize the loaded modules. Developers may replace the core modules with their own implementations via application configuration, or they may write new modules to provide additional functionalities. For example, a module may be developed to provide common database logic for one or several pages. For more details, please see the configurations.

There are three core modules that are loaded by default whenever an application runs. They are request module, response module, and error handler module. In addition, session module is loaded when it is used in the application. PRADO provides default implementation for all these modules. Custom modules may be configured or developed to override or supplement these core modules.

### 2.5.1 Request Module

Request module represents provides storage and access scheme for user request sent via HTTP. User request data comes from several sources, including URL, post data, session data, cookie data, etc. These data can all be accessed via the request module. By default, PRADO uses `THttpRequest` as request module. The request module can be accessed via the `Request` property of application and controls.

### 2.5.2 Response Module

Response module implements the mechanism for sending output to client users. Response module may be configured to control how output are cached on the client side. It may also be used to send cookies back to the client side. By default, PRADO uses `THttpResponse` as response module. The response module can be accessed via the `Response` property of application and controls.

### 2.5.3 Session Module

Session module encapsulates the functionalities related with user session handling. Session module is automatically loaded when an application uses session. By default, PRADO uses `THttpSession` as session module, which is a simple wrapper of the session functions provided by PHP. The session module can be accessed via the `Session` property of application and controls.

### 2.5.4 Error Handler Module

Error handler module is used to capture and process all error conditions in an application. PRADO uses `TErrorHandler` as error handler module. It captures all PHP warnings, notices and exceptions, and displays in an appropriate form to end-users. The error handler module can be accessed via the `ErrorHandler` property of the application instance.

### 2.5.5 Custom Modules

PRADO is released with a few more modules besides the core ones. They include caching modules (`TSqliteCache` and `TMemCache`), user management module (`TUserManager`), authentication and authorization module (`TAuthManager`), etc.

When `TPageService` is requested, it also loads modules specific for page service, including asset manager (`TAssetManager`), template manager (`TTemplateManager`), theme/skin manager (`TThemeManager`), and page state persister (`TPageStatePersister`).

Custom modules and core modules are all configurable via configurations.

## 2.6 Services

A service is an instance of a class implementing the `IService` interface. Each kind of service processes a specific type of user requests. For example, the page service responds to users' requests for PRADO pages.

A service is uniquely identified by its `ID` property. By default when `THttpRequest` is used as the request module, GET variable names are used to identify which service a user is requesting. If a GET variable name is equal to some service `ID`, the request is considered for that service, and the

value of the GET variable is passed as the service parameter. For page service, the name of the GET variable must be `page`. For example, the following URL requests for the `Fundamentals.Services` page,

```
http://hostname/index.php?page=Fundamentals.Services
```

Developers may implement additional services for their applications. To make a service available, configure it in application configurations.

### 2.6.1 Page Service

PRADO implements `TPageService` to process users' page requests. Pages are stored under a directory specified by the `BasePath` property of the page service. The property defaults to `pages` directory under the application base path. You may change this default by configuring the service in the application configuration.

Pages may be organized into subdirectories under the `BasePath`. In each directory, there may be a page configuration file named `config.xml`, which contains configurations effective only when a page under that directory or a sub-directory is requested. For more details, see the page configuration section.

Service parameter for the page service refers to the page being requested. A parameter like `Fundamentals.Services` refers to the `Services` page under the `<BasePath>/Fundamentals` directory. If such a parameter is absent in a request, a default page named `Home` is assumed. Using `THttpRequest` as the request module (default), the following URLs will request for `Home`, `About` and `Register` pages, respectively,

```
http://hostname/index.php<br/>
http://hostname/index.php?page=About<br/>
http://hostname/index.php?page=Users.Register
```

where the first example takes advantage of the fact that the page service is the default service and `Home` is the default page.

## 2.7 Applications

An application is an instance of `TApplication` or its derived class. It manages modules that provide different functionalities and are loaded when needed. It provides services to end-users. It is the central place to store various parameters used in an application. In a PRADO application, the application instance is the only object that is globally accessible via `Prado::getApplication()` function call.

Applications are configured via application configurations. They are usually created in entry scripts like the following,

```
require_once('/path/to/prado.php');
$application = new TApplication;
$application->run();
```

where the method `run()` starts the application to handle user requests.

### 2.7.1 Directory Organization

A minimal PRADO application contains two files: an entry file and a page template file. They must be organized as follows,

1. `wwwroot` - Web document root or sub-directory.
2. `index.php` - entry script of the PRADO application.
3. `assets` - directory storing published private files. See assets section.
4. `protected` - application base path storing application data and private script files. This directory should be configured inaccessible to Web-inaccessible, or it may be located outside of Web directories.
5. `runtime` - application runtime storage path. This directory is used by PRADO to store application runtime information, such as application state, cached data, etc.
6. `pages` - base path storing all PRADO pages. See services section.
7. `Home.page` - default page returned when users do not explicitly specify the page requested. This is a page template file. The file name without suffix is the page name. The page class is `TPage`. If there is also a class file `Home.php`, the page class becomes `Home`.

A product PRADO application usually needs more files. It may include an application configuration file named `application.xml` under the application base path `protected`. The pages may be organized in directories, some of which may contain page configuration files named `config.xml`. Fore more details, please see configurations section.

### 2.7.2 Application Deployment

Deploying a PRADO application mainly involves copying directories. For example, to deploy the above minimal application to another server, follow the following steps,

- Copy the content under `wwwroot` to a Web-accessible directory on the new server.
- Modify the entry script file `index.php` so that it includes correctly the `prado.php` file.
- Remove all content under `assets` and `runtime` directories and make sure both directories are writable by the Web server process.

### 2.7.3 Application Lifecycles

Like page lifecycles, an application also has lifecycles. Application modules can register for the lifecycle events. When the application reaches a particular lifecycle and raises the corresponding event, the registered module methods are invoked automatically. Modules included in the PRADO release, such as `TAuthManager`, are using this way to accomplish their goals.

The application lifecycles can be depicted as follows,

## 2.8 Sample: Hello World

"Hello World" perhaps is the simplest *interactive* PRADO application that you can build. It displays to end-users a page with a submit button whose caption is *Click Me*. When the user clicks on the button, the button changes the caption to *Hello World*.

There are many approaches that can achieve the above goal. One can submit the page to the server, examine the POST variable, and generate a new page with the button caption updated. Or one can simply use JavaScript to update the button caption upon its *onclick* event.

PRADO promotes component-based and event-driven Web programming. The button is represented by a *TButton* object. It encapsulates the button caption as the *Text* property and associates the user button click action with a server-side *Click* event. Therefore, the "Hello World" task can be handled intuitively and easily. One simply needs to attach a function to the button's *Click* event. Within the function, the button's *Text* property is modified as "Hello World". The following diagram shows the above sequence,

The code that a developer needs to write is merely the following event handler function, where `$sender` refers to the button object.

```
public function buttonClicked($sender,$param)
{
$sender->Text = "Hello World";
}
```

Try, `http://../quickstart/index.php?page=Fundamentals.Samples.HelloWorld.Home`

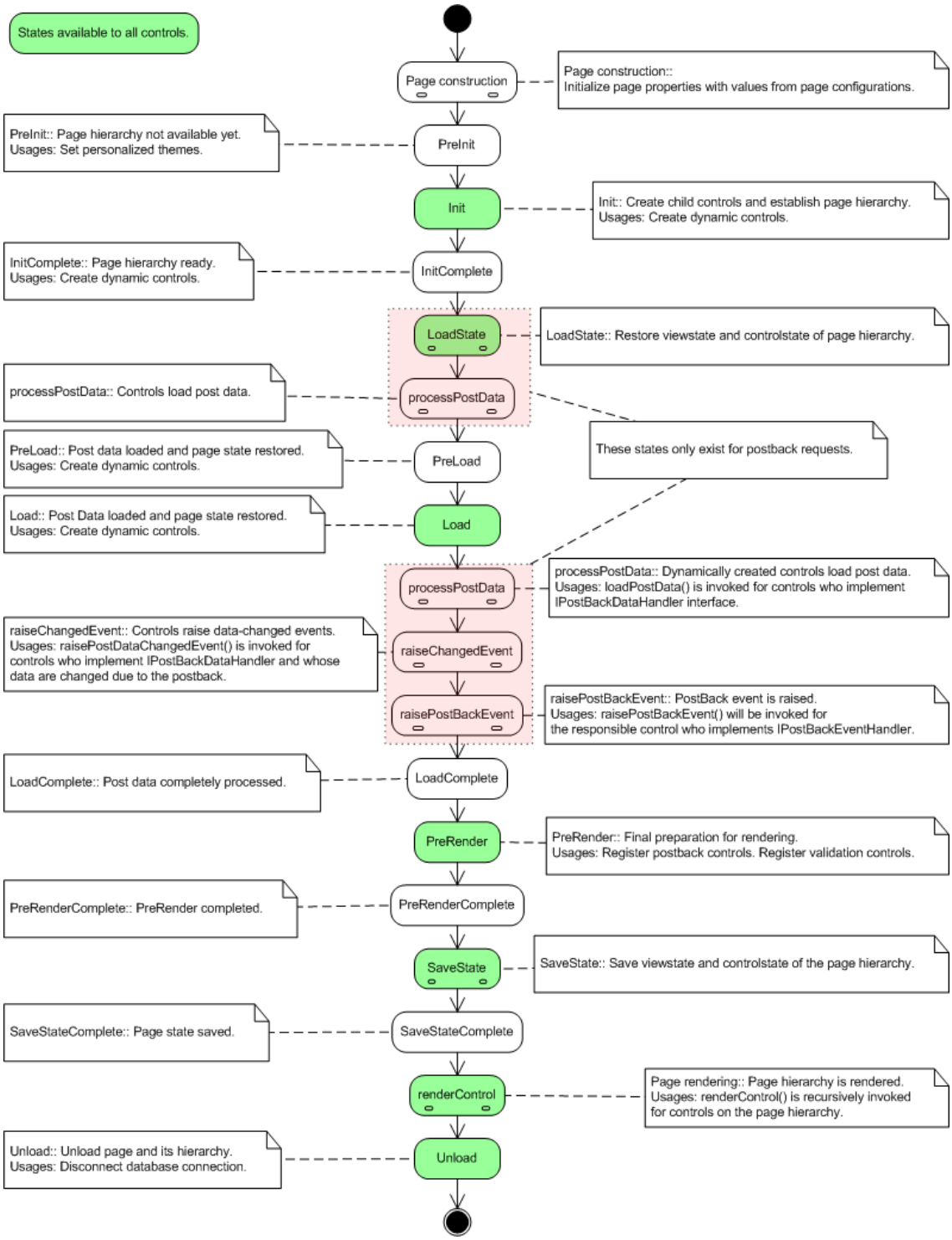
## 2.9 Sample: Hangman Game

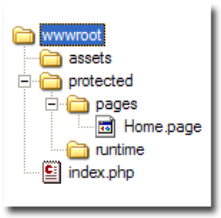
Having seen the simple "Hello World" application, we now build a more complex application called "Hangman Game". In this game, the player is asked to guess a word, a letter at a time. If he guesses a letter right, the letter will be shown in the word. The player can continue to guess as long as the number of his misses is within a prespecified bound. The player wins the game if he finds out the word within the miss bound, or he loses.

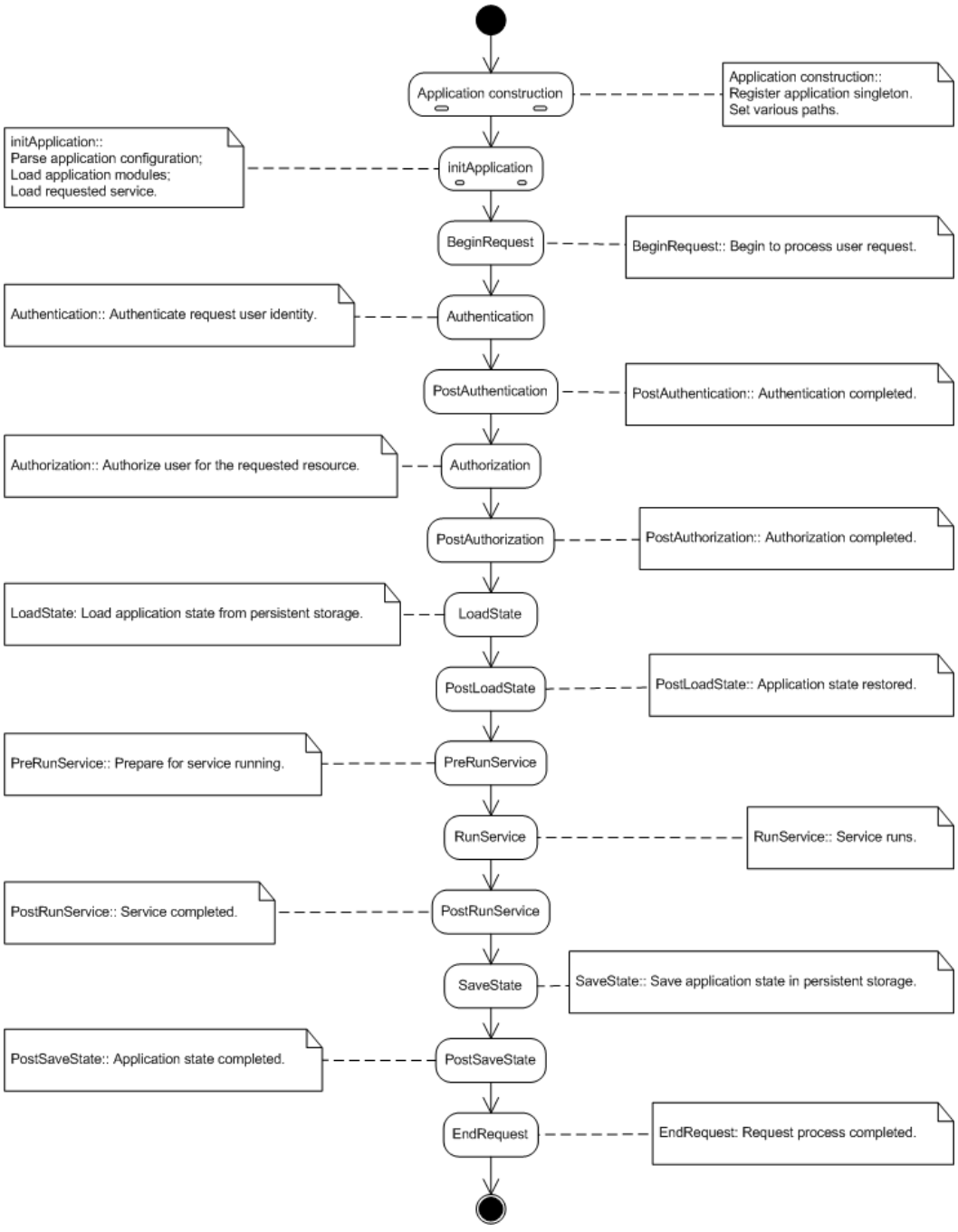
To facilitate the building of this game, we show the state transition diagram of the gaming process in the following,

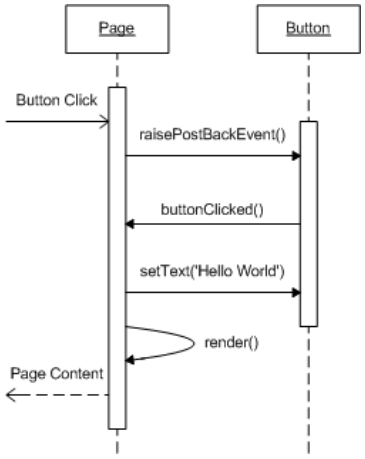
To be continued...

Try, `http://../quickstart/index.php?page=Fundamentals.Samples.Hangman.Home`


States available to all controls.
Page construction
Page construction::
Initialize page properties with values from page configurations.
PreInit:: Page hierarchy not available yet.
Usages: Set personalized themes.
PreInit
Init
Init:: Create child controls and establish page hierarchy.
Usages: Create dynamic controls.
InitComplete:: Page hierarchy ready.
Usages: Create dynamic controls.
InitComplete
LoadState
LoadState:: Restore viewstate and controlstate of page hierarchy.
processPostData:: Controls load post data.
processPostData
These states only exist for postback requests.
PreLoad:: Post data loaded and page state restored.
Usages: Create dynamic controls.
PreLoad
Load:: Post Data loaded and page state restored.
Usages: Create dynamic controls.
Load
processPostData
processPostData:: Dynamically created controls load post data.
Usages: loadPostData() is invoked for controls who implement
IPostBackDataHandler interface.
raiseChangedEvent:: Controls raise data-changed events.
Usages: raisePostDataChangedEvent() is invoked for
controls who implement IPostBackDataHandler and whose
data are changed due to the postback.
raiseChangedEvent
raisePostBackEvent
raisePostBackEvent:: PostBack event is raised.
Usages: raisePostBackEvent() will be invoked for
the responsible control who implements IPostBackEventHandler.
LoadComplete:: Post data completely processed.
LoadComplete
PreRender
PreRender:: Final preparation for rendering.
Usages: Register postback controls. Register validation controls.
PreRenderComplete:: PreRender completed.
PreRenderComplete
SaveState
SaveState:: Save viewstate and controlstate of the page hierarchy.
SaveStateComplete:: Page state saved.
SaveStateComplete
renderControl
Page rendering:: Page hierarchy is rendered.
Usages: renderControl() is recursively invoked
for controls on the page hierarchy.
Unload:: Unload page and its hierarchy.
Usages: Disconnect database connection.
Unload


wwwroot
assets
protected
pages
Home.page
runtime
index.php


Application construction
Application construction::
Register application singleton.
Set various paths.
initApplication::
Parse application configuration;
Load application modules;
Load requested service.
initApplication
BeginRequest
BeginRequest:: Begin to process user request.
Authentication:: Authenticate request user identity.
Authentication
PostAuthentication
PostAuthentication:: Authentication completed.
Authorization:: Authorize user for the requested resource.
Authorization
PostAuthorization
PostAuthorization:: Authorization completed.
LoadState: Load application state from persistent storage.
LoadState
PostLoadState
PostLoadState:: Application state restored.
PreRunService:: Prepare for service running.
PreRunService
RunService
RunService:: Service runs.
PostRunService:: Service completed.
PostRunService
SaveState
SaveState:: Save application state in persistent storage.
PostSaveState:: Application state completed.
PostSaveState
EndRequest
EndRequest: Request process completed.


Page
Button
Button Click
raisePostBackEvent()
buttonClicked()
setText('Hello World')
render()
Page Content

# Chapter 3

# Configurations

## 3.1 Configuration Overview

PRADO uses configurations to glue together components into pages and applications. There are application configurations, page configurations, and templates.

Application and page configurations are optional if default values are used. Templates are mainly used by pages and template controls. They are optional, too.

## 3.2 Templates: Part I

Templates are used to specify the presentational layout of controls. A template can contain static text, components, or controls that contribute to the ultimate presentation of the associated control. By default, an instance of `TTemplateControl` or its subclass may automatically load and instantiate a template from a file whose name is the same as the control class name. For page templates, the file name suffix must be `.page`; for other regular template controls, the suffix is `.tpl`.

The template format is like HTML, with a few PRADO-specifc tags, including component tags, template control tags, comment tags, dynamic content tags, and dynamic property tags. .

### 3.2.1 Component Tags

A component tag specifies a component as part of the body content of the template control. If the component is a control, it usually will become a child or grand child of the template control, and its rendering result will be inserted at the place where it is appearing in the template.

The format of a component tag is as follows,

```
<com:ComponentType PropertyName="PropertyValue" ... EventName="EventHandler" ...>
body content
</com:ComponentType>
```

`ComponentType` can be either the class name or the dotted type name (e.g. `System.Web.UI.TControl`) of the component. `PropertyName` and `EventName` are both case-insensitive. `PropertyName` can be a property or subproperty name (e.g. `Font.Name`). Note, `PropertyValue` will be HTML-decoded when assigned to the corresponding property. Content enclosed between the opening and closing component tag are normally treated the body of the component.

It is required that component tags nest properly with each other and an opening component tag be paired with a closing tag, similar to that in XML. The following shows a component tag specifying the `Text` property and `Click` event of a button control,

```
<com:TButton Text="Register" OnClick="registerUser" />
```

To deal conveniently with properties taking take big trunk of initial data, the following property initialization tag is introduced,

```
<prop:PropertyName>
PropertyValue
</prop:PropertyName>
```

It is equivalent to `...PropertyName="PropertyValue"...` in every aspect. Property initialization tags must be directly enclosed between the corresponding opening and closing component tag.

### 3.2.2 Template Control Tags

A template control tag is used to configure the initial property values of the control owning the template. Its format is as follows,

```
<%@ PropertyName="PropertyValue" ... %>
```

Like in component tags, `PropertyName` is case-insensitive and can be a property or subproperty name.

Initial values specified via the template control tag are assigned to the corresponding properties when the template control is being constructed. Therefore, you may override these property values in a later stage, such as the `Init` stage of the control.

Template control tag is optional in a template. Each template can contain at most one template control tag. You can place the template control tag anywhere in the template. It is recommended that you place it at the beginning of the template for better visibility.

### 3.2.3 Comment Tags

Comment tags are used to put comments in the template or the ultimate rendering result. There are two types of comment tags. One is like that in HTML and will be displayed to the end-users. The other only appear in a template and will be stripped out when the template is instantiated and displayed to the end-users. The format of these two comment tags is as follows,

```
<!--
Comments VISIBLE to end-users
-->

<!
Comments INVISIBLE to end-users
!>
```

## 3.3 Templates: Part II

### 3.3.1 Dynamic Content Tags

Dynamic content tags are introduced as shortcuts to some commonly used component tags. These tags are mainly used to render contents resulted from evaluating some PHP expressions or statements. They include expression tags, statement tags, databind tags, parameter tags, asset tags and localization tags.

#### Expression Tags

An expression tag represents a PHP expression that is evaluated when the template control is being rendered. The expression evaluation result is inserted at the place where the tag resides in the template. Its format is as follows,

```
<%= PhpExpression %>
```

Inernally, an expression tag is represented by a `TExpression` control. Therefore, in the expression `$this` refers to the `TExpression` control. For example, the following expression tag will display the current page title at the place,

```
<%= $this->Page->Title %>
```

#### Statement Tags

Statement tags are similar to expression tags, except that statement tags contain PHP statements rather than expressions. The output of the PHP statements (using for example `echo` or `print` in PHP) are displayed at the place where the statement tag resides in the template. Inernally, a statement tag is represented by a `TStatements` control. Therefore, in the statements `$this` refers to the `TStatements` control. The format of statement tags is as follows,

```
<%
PHP Statements
%>
```

The following example displays the current time in Dutch at the place,

```
<%%
setlocale(LC_ALL, 'nl_NL');
echo strftime("%A %e %B %Y",time());
%>
```

### Databind Tags

Databind tags are similar to expression tags, except that the expressions are evaluated only when a `dataBind()` call is invoked on the controls representing the databind tags. Internally, a `TLiteral` control is used to represent a databind tag and `$this` in the expression would refer to the control. The format of databind tags is as follows,

```
<%# PhpExpression %>
```

### Parameter Tags

Parameter tags are used to insert application parameters at the place where they appear in the template. The format of parameter tags is as follows,

```
<%$ ParameterName %>
```

Note, application parameters are usually defined in application configurations or page directory configurations. The parameters are evaluated when the template is instantiated.

### Asset Tags

Asset tags are used to publish private files and display the corresponding the URLs. For example, if you have an image file that is not Web-accessible and you want to make it visible to end-users, you can use asset tags to publish this file and show the URL to end-users so that they can fetch the published image.

The format of asset tags is as follows,

```
<%~ LocalFileName %>
```

where `LocalFileName` refers to a file path that is relative to the directory containing the current template file. The file path can be a single file or a directory. If the latter, the content in the whole directory will be made accessible by end-users.

BE VERY CAUTIOUS when you are using asset tags as it may expose to end-users files that you probably do not want them to see.

#### Localization Tags

Localization tags represent localized texts. They are in the following format,

```
<%[string]%>
```

where `string` will be translated to different languages according to the end-user's language preference.

## 3.4 Templates: Part III

### 3.4.1 Dynamic Property Tags

Dynamic property tags are very similar to dynamic content tags, except that they are applied to component properties. The purpose of dynamic property tags is to allow more versatile component property configuration. Note, you are not required to use dynamic property tags because what can be done using dynamic property tags can also be done in PHP code. However, using dynamic property tags bring you much more convenience at accomplishing the same tasks. The basic usage of dynamic property tags is as follows,

```
<com:ComponentType PropertyName=DynamicPropertyTag ...>
body content
</com:ComponentType>
```

where you may enclose `DynamicPropertyTag` within single or double quotes for better readability.

Like dynamic content tags, we have expression tags, databind tags, parameter tags, asset tags and localization tags. (Note, there is no statement tag here.)

### Expression Tags

An expression tag represents a PHP expression that is evaluated when the template is being instantiated. The expression evaluation result is assigned to the corresponding component property. The format of expression tags is as follows,

```
<%= PhpExpression %>
```

In the expression, `$this` refers to the component specified by the component tag. The following example specifies a `TLabel` control whose `Text` property is initialized as the current page title when the `TLabel` control is being constructed,

```
<com:TLabel Text=<%= $this->Page->Title %> />
```

Note, unlike dynamic content tags, the expressions tags for component properties are evaluated when the components are being constructed, while for the dynamic content tags, the expressions are evaluated when the controls are being rendered.

### Databind Tags

Databind tags are similar to expression tags, except that they can only be used with control properties and the expressions are evaluated only when a `dataBind()` call is invoked on the controls represented by the component tags. In the expression, `$this` refers to the control itself. Databind tags do not apply to all components. They can only be used for controls.

The format of databind tags is as follows,

```
<%# PhpExpression %>
```

### Parameter Tags

Parameter tags are used to assign application parameter values to the corresponding component properties. The format of parameter tags is as follows,

```
<%$ ParameterName %>
```

Note, application parameters are usually defined in application configurations or page directory configurations. The parameters are evaluated when the template is instantiated.

#### Asset Tags

Asset tags are used to publish private files and assign the corresponding the URLs to the component properties. For example, if you have an image file that is not Web-accessible and you want to make it visible to end-users, you can use asset tags to publish this file and show the URL to end-users so that they can fetch the published image.

The format of asset tags is as follows,

```
<%~ LocalFileName %>
```

where `LocalFileName` refers to a file path that is relative to the directory containing the current template file. The file path can be a single file or a directory. If the latter, the content in the whole directory will be made accessible by end-users.

BE VERY CAUTIOUS when you are using asset tags as it may expose to end-users files that you probably do not want them to see.

#### Localization Tags

Localization tags represent localized texts. They are in the following format,

```
<%[string]>
```

where `string` will be translated to different languages according to the end-user's language preference.

## 3.5 Application Configurations

Application configurations are used to specify the global behavior of an application. They include specification of path aliases, namespace usages, module and service configurations, and parameters.

Configuration for an application is stored in an XML file named `application.xml`, which should be located under the application base path. Its format is shown in the following,

```
<application PropertyName="PropertyValue" ...>
  <paths>
    <alias id="AliasID" path="AliasPath" />
    <using namespace="Namespace" />
  </paths>
  <modules>
    <module id="ModuleID" class="ModuleClass"  PropertyName="PropertyValue" ... />
  </modules>
  <services>
    <service id="ServiceID" class="ServiceClass" PropertyName="PropertyValue" ... />
  </services>
  <parameters>
    <parameter id="ParameterID" class="ParameterClass" PropertyName="PropertyValue" ... />
  </parameters>
</application>
```

1. The outermost element `<application>` corresponds to the `TApplication` instance. The `PropertyName="PropertyValue"` pairs specify the initial values for the properties of `TApplication`.

2. The `<paths>` element contains the definition of path aliases and the PHP inclusion paths for the application. Each path alias is specified via an `<alias>` whose `path` attribute takes an absolute path or a path relative to the directory containing the application configuration file. The `<using>` element specifies a particular path (in terms of namespace) to be appended to the PHP include paths when the application runs. PRADO defines two default aliases: `System` and `Application`. The former refers to the PRADO framework root directory, and the latter refers to the directory containing the application configuration file.

3. The `<modules>` element contains the configurations for a list of modules. Each module is specified by a `<module>` element. Each module is uniquely identified by the `id` attribute and is of type `class`. The `PropertyName="PropertyValue"` pairs specify the initial values for the properties of the module.

4. The `<services>` element is similar to the `<modules>` element. It mainly specifies the services provided by the application.

5. The `<parameters>` element contains a list of application-level parameters that are accessible from anywhere in the application. You may specify component-typed parameters like specifying modules, or you may specify string-typed parameters which take a simpler format as follows,

   ```
   <parameter id="ParameterID">ParameterValue</parameter>
   ```

By default without explicit configuration, a PRADO application when running will load a few core modules, such as `THttpRequest`, `THttpResponse`, etc. It will also provide the `TPageService` as a default service. Configuration and usage of these modules and services are covered in individual sections of this tutorial. Note, if your application takes default settings for these modules and service, you do not need to provide an application configuration. However, if these modules or services are not sufficient, or you want to change their behavior by configuring their property values, you will need an application configuration.

## 3.6 Page Configurations

Page configurations are mainly used by `TPageService` to modify or append the application configuration. As the name indicates, a page configuration is associated with a directory storing some page files. It is stored as an XML file named `config.xml`.

When a user requests a page stored under `<BasePath>/dir1/dir2`, the `TPageService` will try to parse and load `config.xml` files under `<BasePath>/dir1` and `<BasePath>/dir1/dir2`. Paths, modules, and parameters specified in these configuration files will be appended or merged into the existing application configuration.

The format of a page configuration file is as follows,

```
<configuration>
  <paths>
    <alias id="AliasID" path="AliasPath" />
    <using namespace="Namespace" />
  </paths>
  <modules>
    <module id="ModuleID" class="ModuleClass"  PropertyName="PropertyValue" ... />
  </modules>
```

```
  <authorization>
    <allow pages="PageID1,PageID2" users="User1,User2" roles="Role1,Role2" verb="get" />
    <deny pages="PageID1,PageID2" users="User1,User2" roles="Role1,Role2" verb="post" />
  </authorization>
  <pages PropertyName="PropertyValue" ...>
    <page id="PageID" PropertyName="PropertyValue" ... />
  </pages>
  <parameters>
    <parameter id="ParameterID" class="ParameterClass" PropertyName="PropertyValue" ... />
  </parameters>
</configuration>
```

The `<paths>`, `<modules>` and `<parameters>` are similar to those in an application configuration. The `<authorization>` specifies the authorization rules that apply to the current page directory and all its subdirectories. It will be explained in more detail in future sections. The `<pages>` element specifies the initial values for the properties of pages. Each `<page>` element specifies the initial property values for a particular page identified by the `id` attribute. Initial property values given in the `<pages>` element apply to all pages in the current directory and all its subdirectories.

# Chapter 4

# Controls

## 4.1 Controls Overview

Control are components defined in addition with user interface. Control classes constitute a major part of the PRADO framework. Nearly every generic HTML element can find its representation in terms of a PRADO control. Mastering these controls becomes extremely important for developers to compose effectively and efficiently applications using PRADO.

To be continued...

## 4.2 Simple HTML Controls

### 4.2.1 TLabel

`TLabel` displays a piece of text on a Web page. The text to be displayed is set via its `Text` property. If `Text` is empty, content enclosed within the `TLabel` component tag will be displayed. `TLabel` may also be used as a form label associated with some control on the form. Since `Text` is not HTML-encoded when being rendered, make sure it does not contain dangerous characters that you want to avoid.

Try, `http://../quickstart/index.php?page=Controls.Samples.TLabel.Home`

### 4.2.2 THyperLink

`THyperLink` displays a hyperlink on a page. The hyperlink URL is specified via the `NavigateUrl` property, and link text is via the `Text` property. The link target is specified via the `Target` property. It is also possible to display an image by setting the `ImageUrl` property. In this case, `Text` is displayed as the alternate text of the image. If both `ImageUrl` and `Text` are empty, the content enclosed within the control tag will be rendered.

Try, `http://../quickstart/index.php?page=Controls.Samples.THyperLink.Home`

### 4.2.3 TImage

`TImage` displays an image on a page. The image is specified via the `ImageUrl` property which takes a relative or absolute URL to the image file. The alignment of the image displayed is set by the `ImageAlign` property. To set alternate text or long description of the image, use `AlternateText` or `DescriptionUrl`, respectively.

Try, `http://../quickstart/index.php?page=Controls.Samples.TImage.Home`

### 4.2.4 TPanel

`TPanel` acts as a presentational container for other control. It displays a ¡div¿ element on a page. The property `Wrap` specifies whether the panel's body content should wrap or not, while `HorizontalAlign` governs how the content is aligned horizontally and `Direction` indicates the content direction (left to right or right to left). You can set `BackImageUrl` to give a background image to the panel, and you can ste `GroupingText` so that the panel is displayed as a field set with a legend text. Finally, you can specify a default button to be fired when users press 'return' key within the panel by setting the `DefaultButton` property.

Try, `http://../quickstart/index.php?page=Controls.Samples.TPanel.Home`

### 4.2.5 TTable

`TTable` displays an HTML table on a page. It is used together with `TTableRow` and `TTableCell` to allow programmatically manipulating HTML tables. The rows of the table is stored in `Rows` property. You may set the table cellspacing and cellpadding via the `CellSpacing` and `CellPadding`

properties, respectively. The table caption can be specified via `Caption` whose alignment is specified by `CaptionAlign`. The `GridLines` property indicates how the table should display its borders, and the `BackImageUrl` allows the table to have a background image.

Try, `http://../quickstart/index.php?page=Controls.Samples.TTable.Home`

### 4.2.6 TTextBox

`TTextBox` displays a text box on a Web page. The content in the text box is determined by the `Text` property. You can create a `SingleLine`, a `MultiLine`, or a `Password` text box by setting the `TextMode` property. The `Rows` and `Columns` properties specify their dimensions. If `AutoPostBack` is true, changing the content in the text box and then moving the focus out of it will cause postback action.

Try, `http://../quickstart/index.php?page=Controls.Samples.TTextBox.Home`

### 4.2.7 TButton

`TButton` creates a click button on a Web page. The button's caption is specified by `Text` property. A button is used to submit data to a page. `TButton` raises two server-side events, `Click` and `Command`, when it is clicked on the client-side. The difference between `Click` and `Command` events is that the latter event is bubbled up to the button's ancestor controls. A `Command` event handler can use `CommandName` and `CommandParameter` associated with the event to perform specific actions.

Clicking on button can trigger form validation, if `CausesValidation`is true. And the validation may be restricted within a certain group of validator controls according to `ValidationGroup`.

Try, `http://../quickstart/index.php?page=Controls.Samples.TButton.Home`

### 4.2.8 TLinkButton

`TLinkButton` is similar to `TButton` in every aspect except that `TLinkButton` is displayed as a hyperlink. The link text is determined by its `Text` property. If the `Text` property is empty, then the body content of the button is displayed (therefore, you can enclose a ¡img¿ tag within the button body and get an image button.

Try, `http://../quickstart/index.php?page=Controls.Samples.TLinkButton.Home`

### 4.2.9 TImageButton

`TImageButton` is also similar to `TButton`, except that `TImageButton` displays the button as an image. The image is specified via `ImageUrl`, and the alternate text is specified by `Text`. In addition, it is possible to obtain the coordinate of the point where the image is clicked. The coordinate information is contained in the event parameter of the `Click` event (not `Command`).

Try, `http://../quickstart/index.php?page=Controls.Samples.TImageButton.Home`

### 4.2.10 TCheckBox

`TCheckBox` displays a check box on a Web page. A caption can be specified via `Text` and displayed beside the check box. It can appear either on the right or left of the check box, which is determined by `TextAlign`. You may further specify attributes applied to the text by using `LabelAttributes`.

To determine whether the check box is checked, test the `Checked` property. A `CheckedChanged` event is raised if the state of `Checked` is changed between posts to the server. If `AutoPostBack` is true, changing the check box state will cause postback action. And if `CausesValidation` is also true, upon postback validation will be performed for validators within the specified `ValidationGroup`.

Try, `http://../quickstart/index.php?page=Controls.Samples.TCheckBox.Home`

### 4.2.11 TRadioButton

`TRadioButton` is similar to `TCheckBox` in every aspect, except that `TRadioButton` displays a radio button on a Web page. The radio button can belong to a specific group specified by `GroupName` such that only one radio button within that group can be selected at most.

Try, `http://../quickstart/index.php?page=Controls.Samples.TRadioButton.Home`

## 4.3 List Controls

List controls covered in this section all inherit directly or indirectly from `TListControl`. Therefore, they share the same set of commonly used properties, including,

1. `Items` - list of items in the control. The items are of type `TListItem`. The item list can be populated via databinding or specified in templates like the following:

```
<com:TListBox>
  <com:TListItem Text="text 1" Value="value 1" />
  <com:TListItem Text="text 2" Value="value 2" Selected="true" />
  <com:TListItem Text="text 3" Value="value 3" />
</com:TListBox>
```

2. `SelectedIndex` - the zero-based index of the first selected item in the item list.
3. `SelectedIndices` - the indices of all selected items.
4. `SelectedItem` - the first selected item in the item list.
5. `SelectedValue` - the value of the first selected item in the item list.
6. `AutoPostBack` - whether changing the selection of the control should trigger postback.
7. `CausesValidation` - whether validation should be performed when postback is triggered by the list control.

Since `TListControl` inherits from `TDataBoundControl`, these list controls also share a common operation known as `databinding`. The data to be bound can be specified via either `DataSource` or `DataSourceID`. More details about databinding are covered in later chapters of this tutorial.

### 4.3.1 TListBox

`TListBox` displays a list box that allows single or multiple selection. Set the property `SelectionMode` as `Single` to make a single selection list box, and `Multiple` a multiple selection list box. The items in the list box are represented by the `Items` property. The number of rows displayed in the box is specified via the `Rows` property value.

Try, `http://../quickstart/index.php?page=Controls.Samples.TListBox.Home`

### 4.3.2 TDropDownList

`TDropDownList` displays a dropdown list box that allows users to select a single option from a few prespecified ones. The items in the list box are represented by the `Items` property. The selected item can be retrieved via `SelectedItem` property. If `AutoPostBack` is true, selection change will cause page postback.

Try, `http://../quickstart/index.php?page=Controls.Samples.TDropDownList.Home`

### 4.3.3 TCheckBoxList

Try, `http://../quickstart/index.php?page=Controls.Samples.TCheckBoxList.Home`

### 4.3.4 TRadioButtonList

Try, `http://../quickstart/index.php?page=Controls.Samples.TRadioButtonList.Home`

### 4.3.5 TBulletList

Try, `http://../quickstart/index.php?page=Controls.Samples.TBulletedList.Home`

## 4.4 Validation Controls

### 4.4.1 TRequiredFieldValidator

Try, `http://../quickstart/index.php?page=Controls.Samples.TRequiredFieldValidator.Home`

### 4.4.2 TRegularExpressionValidator

Try, `http://../quickstart/index.php?page=Controls.Samples.TRegularExpressionValidator.Home`

### 4.4.3 TEmailAddressValidator

Try, `http://../quickstart/index.php?page=Controls.Samples.TEmailAddressValidator.Home`

### 4.4.4 TEmailAddressValidator

Try, `http://../quickstart/index.php?page=Controls.Samples.TEmailAddressValidator.Home`

### 4.4.5 TCompareValidator

Try, `http://../quickstart/index.php?page=Controls.Samples.TCompareValidator.Home`

### 4.4.6 TCustomValidator

Try, `http://../quickstart/index.php?page=Controls.Samples.TCustomValidator.Home`

### 4.4.7 TValidationSummary

Try, `http://../quickstart/index.php?page=Controls.Samples.TValidationSummary.Home`

## 4.5 TDataList

TBC

Try, `http://../quickstart/index.php?page=Controls.Samples.TDataList.Sample1`

## 4.6 TDataGrid

TBC

Try, `http://../quickstart/index.php?page=Controls.Samples.TDataGrid.Sample1`