Prado v3.0 Quick Start Tutorial [1]

Qiang Xue, Wei Zhuo

February 7, 2006

Contents

Preface

Prado quick start doc

License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the developer nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Chapter 1

Getting Started

1.1 Welcome to the PRADO QuickStart Tutorial

This QuickStart tutorial is provided to help you quickly start building your own Web applications based on PRADO version 3.0.

You may refer to the following resources if you find this tutorial does not fulfill all your needs.

- PRADO API Documentation
- PRADO Forum
- PRADO Wiki
- PRADO Trac

1.2 What is PRADO?

PRADO is a component-based and event-driven programming framework for developing Web applications in PHP 5. PRADO stands for **P**HP **R**apid **A**pplication **D**evelopment **O**bject-oriented.

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.

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

1.2.1 Why PRADO?

A primary goal of PRADO is to enable maximum reusability in Web programming. By reusability, we mean not only reusing one's own code, but also reusing other people's code in an easy way. The latter is more important as it saves the effort of reinventing the wheels and may cut off development time dramatically. The introduction of the concept of component is for this purpose.

Reusing existing components is very easy. It merely involves getting and setting component properties, and sometimes responding to component events. PRADO provides a complete set of components that deal with common Web programming tasks, such as collecting and validating user inputs through generic HTML elements, manipulating tabular data, etc. These components can be rapidly glued together and form Web pages that are highly user interactive. For example, using the datagrid component, one only needs to write a few lines of PHP code (mainly to populate the data into the datagrid), and he can create a page presenting a data table which allows paging, sorting, editting, and deleting rows of data.

Developing new components can also be very easy. A new component class can be created by composing together several existing components in a template which specifies the layout of the constituent components. The page that you are reading now is such an example. It is the presentation of a component without a single line of PHP code.

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.

1.2.2 What Is PRADO Best For?

PRADO is best suitable for creating Web front-ends that are highly user-interactive and require small to medium traffic. It can be used to develop systems as simple as a blog system to systems as complex as a content management system (CMS) or a complete e-commerce solution. PRADO can help you cut your development time significantly.

PRADO does not exclude other back-end solutions such as most DB abstraction layers. In fact, they can be used like what you usually do with traditional PHP programming.

Without server caching techniques, PRADO may not be suitable for developing extremely high-traffic Web applications, such as popular portals, forums, etc. In these applications, every niche of potential performance gain must be exploited and server caching (e.g. Zend optimizer) is almost a must.

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,

1. Go to pradosoft.com to grab a latest version of PRADO.
2. Unpack the PRADO release file using *unzip* on Linux or *winzip* on Windows. A directory named *prado* will be created under the working directory.
3. Copy or upload everything under the *prado* directory to the DocumentRoot directory (or a subdirectory) of the Web server.
4. 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,

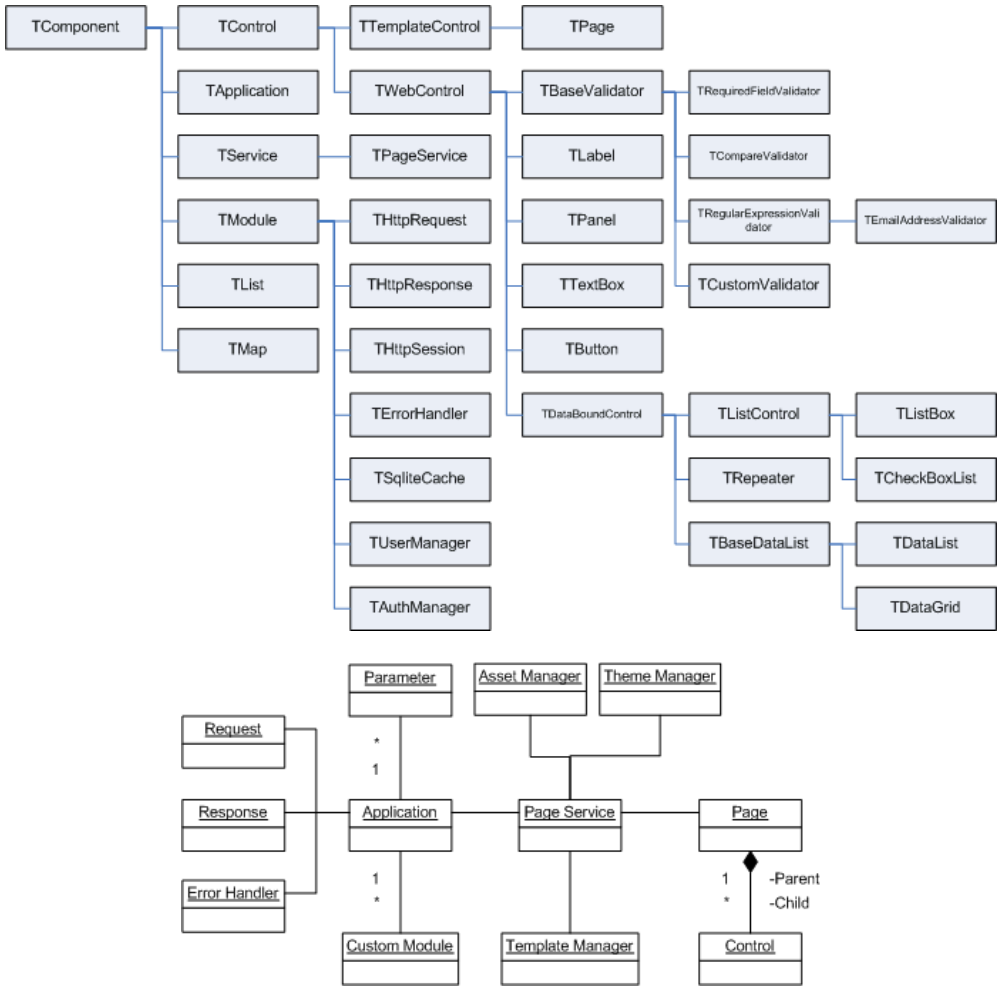
When a PRADO application is processing a page request, its static object diagram can be shown as follows,

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
http://hostname/index.php?page=About
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,

- `wwwroot` - Web document root or sub-directory.
- `index.php` - entry script of the PRADO application.
- `assets` - directory storing published private files. See assets section.
- `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.
- `runtime` - application runtime storage path. This directory is used by PRADO to store application runtime information, such as application state, cached data, etc.
- `pages` - base path storing all PRADO pages. See services section.
- `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,

1. Copy the content under `wwwroot` to a Web-accessible directory on the new server.
2. Modify the entry script file `index.php` so that it includes correctly the `prado.php` file.
3. 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";
}
```

The following line in the page template attaches the `buttonClicked()` method to the `OnClick` event of the button,

```
<com:TButton Text="Click Me" OnClick="buttonClicked" />
```

Try, Fundamentals.Samples.HelloWorld.Home

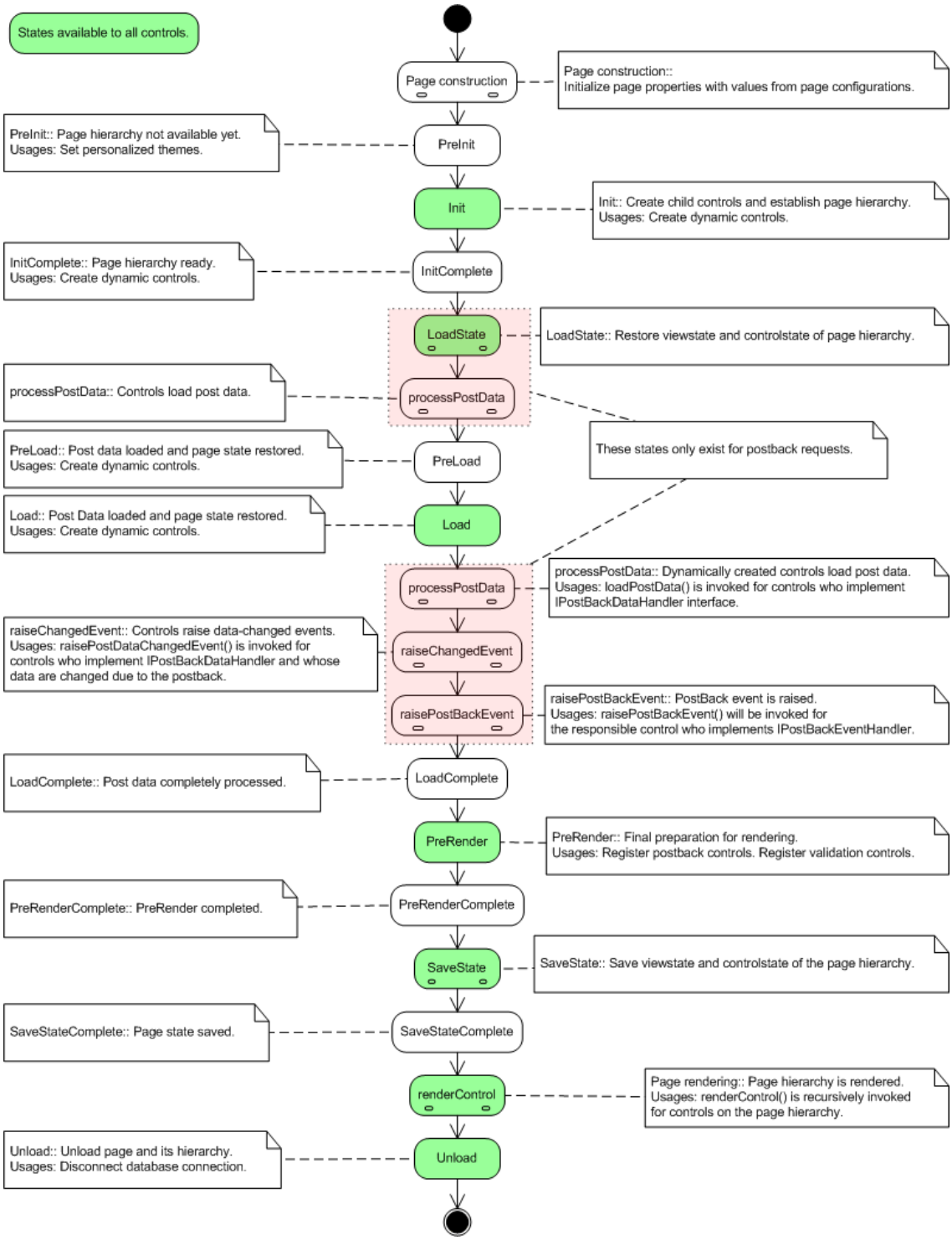
2.9 Sample: Hangman Game

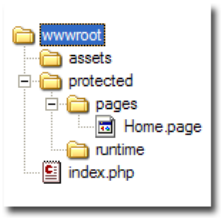
Having seen the simple "Hello World" application, we now build a more complex application called "Hangman Game". In this game, the player is asked to guess a word, a letter at a time. If he guesses a letter right, the letter will be shown in the word. The player can continue to guess as long as the number 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.

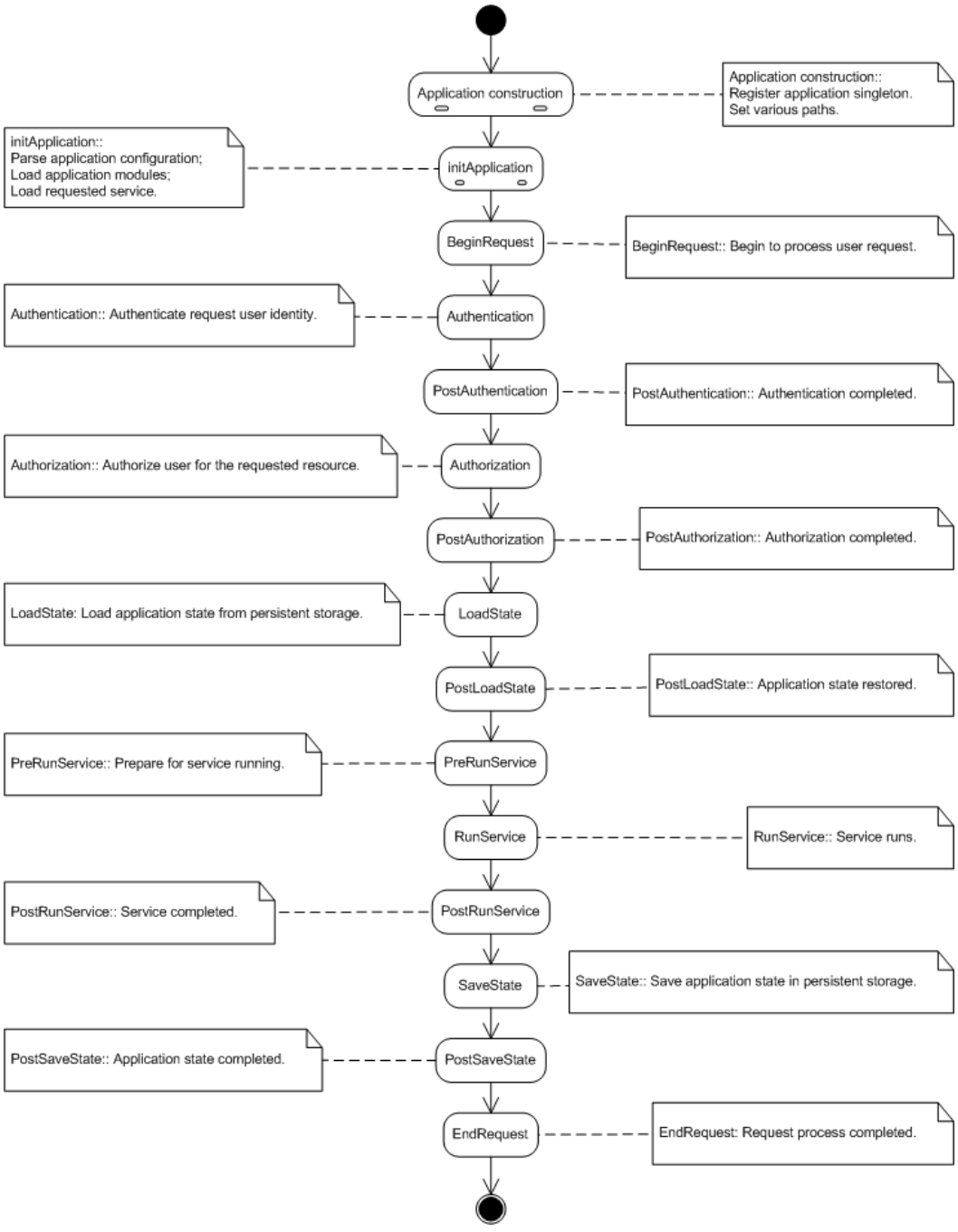
To facilitate the building of this game, we show the state transition diagram of the gaming process in the following,

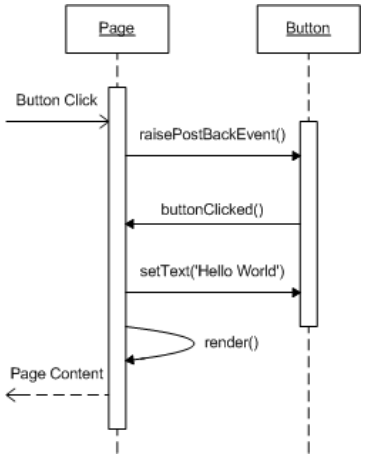
To be continued...

Try, 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 template shows a component tag specifying the `Text` property and `OnClick` event of a button control,

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

Note, property names and event names are all case-insensitive, while component type names are case-sensitive. Event names always begin with `On`.

Also note, initial values for properties whose name ends with `Template` are specially processed. In particular, the initial values are parsed as `TTemplate` objects. The `ItemTemplate` property of the `TRepeater` control is such an example.

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.

Component IDs

When specified in templates, component `ID` property has special meaning in addition to its normal property definition. A component tag specified with an ID value in template will register the corresponding component to the template owner control. The component can thus be directly accessed from the template control with its ID value. For example, in `Home` page's template, the following component tag

```
<com:TTextBox ID="TextBox" Text="First Name" />
```

makes it possible to get the textbox object in code using `$page->TextBox`.

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>
```

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

- 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.

- 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.

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

- 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" value="ParameterValue" />
  ```

  Note, if the `value` attribute is not specified, the whole parameter XML node (of type `TXmlElement`) will be returned as the parameter value.

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>`, `<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. Here `<BasePath>` is as defined in page service.

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.

TBW: Control ID, Parent, NamingContainer, ViewState, ControlState, TWebControl, DataBound Control, ActiveControls

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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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,

- `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>
```

- `SelectedIndex` - the zero-based index of the first selected item in the item list.
- `SelectedIndices` - the indices of all selected items.
- `SelectedItem` - the first selected item in the item list.
- `SelectedValue` - the value of the first selected item in the item list.
- `AutoPostBack` - whether changing the selection of the control should trigger postback.
- `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 `Items` can be populated from preexisting data specified by `DataSource` or `DataSourceID`. A function call to `dataBind()` will cause the data population. For list controls, data can be specified in three kinds of format:

- integer-indexed array, `TList` or traversable : each array element value will be used as the value and text for a list item. For example

```
$listbox->DataSource=array('item 1','item 2','item 3');
$listbox->dataBind();
```

- associative array, `TMap` or traversable : array keys will be used as list item values, and array values will be used as list item texts. For example

```
$listbox->DataSource=array(
'key 1'=>'item 1',
'key 2'=>'item 2',
'key 3'=>'item 3');
$listbox->dataBind();
```

- tabular (two-dimensional) data : each row of data populates a single list item. The list item value is specified by the data member indexed with `DataValueField`, and the list item text by `DataTextField`. For example,

```
$listbox->DataTextField='name';
$listbox->DataValueField='id';
$listbox->DataSource=array(
array('id'=>'001','name'=>'John','age'=>31),
array('id'=>'002','name'=>'Mary','age'=>30),
array('id'=>'003','name'=>'Cary','age'=>20));
$listbox->dataBind();
```

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 number of rows displayed in the box is specified via the `Rows` property value.

Try, 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.

Try, Controls.Samples.TDropDownList.Home

4.3.3 TCheckBoxList

`TCheckBoxList` displays a list of checkboxes on a Web page. The alignment of the text besides each checkbox can be specified `TextAlign`. The layout of the checkboxes can be controlled by the following properties:

- `RepeatLayout` - can be either `Table` or `Flow`. A `Table` uses HTML table cells to organize the checkboxes, while a `Flow` uses HTML span tags and breaks for the organization. With `Table` layout, you can set `CellPadding` and `CellSpacing`.
- `RepeatColumns` - how many columns the checkboxes should be displayed in.
- `RepeatDirection` - how to traverse the checkboxes, in a horizontal way or a vertical way (because the checkboxes are displayed in a matrix-like layout).

Try, Controls.Samples.TCheckBoxList.Home

4.3.4 TRadioButtonList

`TRadioButtonList` is similar to `TCheckBoxList` in every aspect except that each `TRadioButtonList` displays a group of radiobuttons. Only one of the radiobuttions can be selected (`TCheckBoxList` allows multiple selections.)

Try, Controls.Samples.TRadioButtonList.Home

4.3.5 TBulletList

`TBulletedList` displays items in a bullet format on a Web page. The style of the bullets can be specified by `BulletStyle`. When the style is `CustomImage`, the bullets are displayed as images, which is specified by `BulletImageUrl`.

`TBulletedList` displays the item texts in three different modes,

- `Text` - the item texts are displayed as static texts;
- `HyperLink` - each item is displayed as a hyperlink whose URL is given by the item value, and `Target` property can be used to specify the target browser window;

- `LinkButton` - each item is displayed as a link button which posts back to the page if a user clicks on that, and the event `OnClick` will be raised under such a circumstance.

Try, Controls.Samples.TBulletedList.Home

4.4 Validation Controls

Validation is performed when a postback control, such as a `TButton`, a `TLinkButton` or a `TTextBox` (under AutoPostBack mode) is submitting the page and its `CausesValidation` property is true.

Validator controls always validate the associated input control on the serve side. In addition, if `EnableClientScript` is true, validation will also be performed on the client-side using javascript. Client-side validation will validate user input before it is sent to the server. The form data will not be submitted if any error is detected. This avoids the round-trip of information necessary for server-side validation.

Every validator component has the following properties, defined in the `TBaseValidator` class.

ControlToValidate The `ID` of the component for this validator. This property must be set to the `ID` path of an input component. The `ID` path is the dot-connected IDs of the components reaching from the validator's parent component to the target component.

ErrorMessage The text for the error message when the input component failed to validate.

ValidationGroup If the control causing the validation also sets its ValidationGroup property, only those validators having the same ValidationGroup value will do input validation.

Display The display behavior of the error message in a validation component. The allowed values are: `None`, `Static` and `Dynamic`. The default is `Static`.

- `None` – the validator component and the error message will not be displayed.
- `Dynamic` – CSS for the error is constructed in such a way that space for the error message on the page is NOT reserved. When the user hits the "submit" button, applicable error messages will show up shifting the layout of your page around (usually down).
- `Static` – CSS for the error is constructed in such a way that space for the error message on the page is always reserved. When the user hits the "submit" button, applicable error messages will just show up, not altering the layout of your page.

EnableClientScript Indicating whether client-side validation is enabled. Default is true.

4.4.1 TRequiredFieldValidator

This is the simplest validator, ensuring that the input field has some sort of value. To ensure that all of our input fields are required, add a `TRequiredFieldValidator` component for each of the input fields. The `TRequiredFieldValidator` also has the following property.

InitialValue The associated input component fails validation if its value does not change from the `InitialValue` upon losing focus.

Try, Controls.Samples.TRequiredFieldValidator.Home

4.4.2 TRegularExpressionValidator

The TRegularExpressionValidator has the following property in addition to the parent `TBaseValidator` properties.

RegularExpression The regular expression that determines the pattern used to validate a field. Some commonly used regular expressions include:

- At least 6 characters: `[\w]{6,}`
- Internal URL: `http://([\w-]+\.)+[\w-]+(/[\w- ./?%&=]*)?`
- Japanese Phone Number: `(0\d{1,4}-|\(0\d{1,4}\) ?)?\d{1,4}-\d{4}`
- Japanese Postal Code: `\d{3}(-(\d{4}|\d{2}))?`
- P.R.C. Phone Number: `(\(\d{3}\)|\d{3}-)?\d{8}`
- P.R.C. Postal Code: `\d{6}`
- P.R.C. Social Security Number: `\d{18}|\d{15}`
- U.S. Phone Number: `((\(\d{3}\) ?)|(\d{3}-))?\d{3}-\d{4}`
- U.S. ZIP Code: `\d{5}(-\d{4})?`
- U.S. Social Security Number: `\d{3}-\d{2}-\d{4}`

More regular expression patterns can be found on the Internet, e.g. http://regexlib.com/.

Try, Controls.Samples.TRegularExpressionValidator.Home

4.4.3 TEmailAddressValidator

`TEmailAddressValidator` validates whether the value of an associated input component is a valid email address. It will check MX record if `checkdnsrr()` is available in the installed PHP.

Try, Controls.Samples.TEmailAddressValidator.Home

4.4.4 TCompareValidator

The validator `TCompareValidator` is used to compare two input fields, the comparison can be made in many ways. The following are the properties of the `TCompareValidator` in addition to the parent `TBaseValidator`.

ControlToCompare The input component to compare with the input control being validated.

ValueToCompare A constant value to compare with the value entered by the user into the input component being validated.

ValueType The data type (`Integer`, `Double`, `Currency`, `Date`, `String`) that the values being compared are converted to before the comparison is made.

Operator The comparison operation to perform (`Equal`, `NotEqual`, `GreaterThan`, `GreaterThanEqual`, `LessThan`, `LessThanEqual`, `DataTypeCheck`).

DateFormat The date format to use during comparision.

To specify the input component to validate, set the `ControlToValidate` property to the `ID` of the input component. To compare the associated input component with another input component, set the `ControlToCompare` property to the `ID` of the component to compare with.

To compare the associated input component with a constant value, specify the constant value to compare with by setting the `ValueToCompare` property.

The `ValueType` property is used to specify the data type of both comparison values. Both values are automatically converted to this data type before the comparison operation is performed. The following value types are supported.

Integer A 32-bit signed integer data type.

Double A double-precision floating point number data type.

Currency A decimal data type that can contain currency symbols.

Date A date data type, the date format depends on the `DateFormat` property.

String A string data type.

Use the `Operator` property to specify the type of comparison to perform. If you set the `Operator` property to `DataTypeCheck`, the `TCompareValidator` component will ignore the `ControlToCompare` and `ValueToCompare` properties and simply indicates whether the value entered into the input component can be converted to the data type specified by the `ValueType` property.

Note that if the input control is empty, no validation functions are called and validation succeeds. Use a `RequiredFieldValidator` control to require the user to enter data into the input control.

Try, Controls.Samples.TCompareValidator.Home

4.4.5 TCustomValidator

Try, Controls.Samples.TCustomValidator.Home

4.4.6 TValidationSummary

Try, Controls.Samples.TValidationSummary.Home

4.5 TRepeater

TRepeater displays its content defined in templates repeatedly based on the given data specified by the `DataSource` or `DataSourceID` property. The repeated contents can be retrieved from the `Items` property. Each item is created by instantiating a template and each is a child control of the repeater.

Like normal control templates, the repeater templates can contain static text, controls and special tags, which after instantiation, become child contents of the corresponding item. TRepeater defiens five templates for different purposes,

- `HeaderTemplate` - the template used for displaying content at the beginning of a repeater;
- `FooterTemplate` - the template used for displaying content at the end of a repeater;
- `ItemTemplate` - the template used for displaying every repeater item. If `AlternatingItemTemplate` is also defined, `ItemTemplate` will be used for displaying item 1, 3, 5, etc.
- `AlternatingItemTemplate` - the template used for displaying every alternating repeater item (i.e., item 2, 4, 6, etc.)
- `SeparatorTemplate` - the template used for displaying content between items.

To populate data into the repeater items, set `DataSource` to a valid data object, such as array, `TList`, `TMap`, or a database table, and then call `dataBind()` for the repeater. That is,

```
class MyPage extends TPage {
    protected function onLoad($param) {
        parent::onLoad($param);
        if(!$this->IsPostBack) {
            $this->Repeater->DataSource=$data;
            $this->Repeater->dataBind();
        }
    }
}
```

Normally, you only need to do this when the page containing the repeater is initially requested. In postbacks, TRepeater is smart enough to remember the previous state, i.e., contents populated with datasource information.The following sample displays tabular data using TRepeater.

TRepeater provides several events to facilitate manipulation of its items,

- `OnItemCreated` - raised each time an item is newly created. When the event is raised, data and child controls are both available for the new item.
- `OnItemDataBound` - raised each time an item just completes databinding. When the event is raised, data and child controls are both available for the item, and the item has finished databindings of itself and all its child controls.
- `OnItemCommand` - raised when a child control of some item (such as a `TButton`) raises an `OnCommand` event.

The following example shows how to use TRepeater to display tabular data.

Try, Controls.Samples.TRepeater.Sample1

TRepeater can be used in more complex situations. As an example, we show in the following how to use nested repeaters, i.e., repeater in repeater. This is commonly seen in presenting master-detail data. To use a repeater within another repeater, for an item for the outer repeater is created, we need to set the detail data source for the inner repeater. This can be achieved by responding to the `OnItemDataBound` event of the outer repeater. An `OnItemDataBound` event is raised each time an outer repeater item completes databinding. In the following example, we exploit another event of repeater called `OnItemCreated`, which is raised each time a repeater item (and its content) is newly created. We respond to this event by setting different background colors for repeater items to achieve alternating item background display. This saves us from writing an `AlternatingItemTemplate` for the repeaters.

Try, Controls.Samples.TRepeater.Sample2

Besides displaying data, TRepeater can also be used to collect data from users. Validation controls can be placed in TRepeater templates to verify that user inputs are valid.

The PRADO component composer demo is a good example of such usage. It uses a repeater to collect the component property and event definitions. Users can also delete or adjust the order of the properties and events, which is implemented by responding to the `OnItemCommand` event of repeater.

See in the following yet another example showing how to use repeater to collect user inputs.

Try, Controls.Samples.TRepeater.Sample3

4.6 TDataList

TDataList is used to display or modify a list of data items specified by its `DataSource` or `DataSourceID` property. Each data item is displayed by a data list item which is a child control of the data list. The `Items` property contains the list of all data list items.

TDataList displays its items in either a `Table` or `Flow` layout, which is specified by the `RepeatLayout` property. A table layout uses HTML table cells to organize the items while a flow layout uses line breaks to organize the items. When the layout is `Table`, the table's cellpadding and cellspacing

can be adjusted by `CellPadding` and `CellSpacing` properties, respectively. And `Caption` and `CaptionAlign` can be used to add a table caption with the specified alignment. The number of columns used to display the data list items is specified via `RepeatColumns` property, while the `RepeatDirection` governs the order of the items being rendered.

Each data list item is created according to one of the seven kinds of templates that developers may specified for a TDataList,

- `HeaderTemplate` - the template used for displaying content at the beginning of a data list;
- `FooterTemplate` - the template used for displaying content at the end of a data list;
- `ItemTemplate` - the template used for displaying every data list item. If `AlternatingItemTemplate` is also defined, `ItemTemplate` will be used for displaying item 1, 3, 5, etc.
- `AlternatingItemTemplate` - the template used for displaying every alternating data list item (i.e., item 2, 4, 6, etc.)
- `SeparatorTemplate` - the template used for displaying content between items.
- `EditItemTemplate` - the template used for displaying items in edit mode.
- `SelectedItemTemplate` - the template used for displaying items in selected mode.

Each of the above templates is associated with a style property that is applied to the items using the template. For example, `ItemTemplate` is associated with a property named `AlternatingItemStyle`. Through this property, one can set CSS style fields or CSS classes for the data list items.

Item styles are applied in a hierarchical way. Style in higher hierarchy will inherit from styles in lower hierarchy. Starting from the lowest hierarchy, the item styles include item's own style, `ItemStyle`, `AlternatingItemStyle`, `SelectedItemStyle`, and `EditItemStyle`. Therefore, if background color is set as red in `ItemStyle`, `EditItemStyle` will also have red background color, unless it is explicitly set to a different value.

A data list item can be in normal mode, edit mode or selected mode. Different templates will apply to items of different modes. To change an item's mode, modify `EditItemIndex` or `SelectedItemIndex`. Note, if an item is in edit mode, then selecting this item will have no effect.

TDataList provides several events to facilitate manipulation of its items,

- `OnItemCreated` - raised each time an item is newly created. When the event is raised, data and child controls are both available for the new item.

- `OnItemDataBound` - raised each time an item just completes databinding. When the event is raised, data and child controls are both available for the item, and the item has finished databindings of itself and all its child controls.
- `OnItemCommand` - raised when a child control of some item (such as a `TButton`) raises an `OnCommand` event.
- command events - raised when a child control's `OnCommand` event has a specific command name,
  - `OnSelectedIndexChanged` - if the command name is `select`.
  - `OnEditCommand` - if the command name is `edit`.
  - `OnDeleteCommand` - if the command name is `delete`.
  - `OnUpdateCommand` - if the command name is `update`.
  - `OnCancelCommand` - if the command name is `cancel`.

The following example shows how to use TDataList to display tabular data, with different layout and styles.

Try, Controls.Samples.TDataList.Sample1

A common use of TDataList is for maintaining tabular data, including browsing, editting, deleting data items. This is enabled by the command events and various item templates of TDataList.

The following example displays a computer product information. Users can add new products, modify or delete existing ones. In order to locate the data item for updating or deleting, `DataKeys` property is used.

Be aware, for simplicity, this application does not do any input validation. In real applications, make sure user inputs are valid before saving them into databases.

Try, Controls.Samples.TDataList.Sample2

4.7 TDataGrid : Part I

TDatagrid is an important control in building complex Web applications. It displays data in a tabular format with rows (also called items) and columns. A row is composed by cells, while

columns govern how cells should be displayed according to their association with the columns. Data specified via `DataSource` or `DataSourceID` are bound to the rows and feed contents to cells.

TDataGrid is highly interactive. Users can sort the data along specified columns, navigate through different pages of the data, and perform actions, such as editting and deleting, on rows of the data.

Rows of TDataGrid can be accessed via its `Items` property. A row (item) can be in one of several modes: browsing, editting and selecting, which affects how cells in the row are displayed. To change an item's mode, modify `EditItemIndex` or `SelectedItemIndex`. Note, if an item is in edit mode, then selecting this item will have no effect.

4.7.1 Columns

Columns of a data grid determine how the associated cells are displayed. For example, cells associated with a `TBoundColumn` are displayed differently according to their modes. A cell is displayed as a static text if the cell is in browsing mode, a text box if it is in editting mode, and so on.

PRADO provides five types of columns:

- `TBoundColumn` associates cells with a specific field of data and displays the cells according to their modes.
- `TCheckBoxColumn` associates cells with a specific field of data and displays in each cell a checkbox whose check state is determined by the data field value.
- `THyperLinkColumn` displays in the cells a hyperlink whose caption and URL can be either statically specified or bound to some fields of data.
- `TEditCommandColumn` displays in the cells edit/update/cancel command buttons according to the state of the item that a cell resides in.
- `TButtonColumn` displays in the cells a command button.
- `TTemplateColumn` displays the cells according to different templates defined for it.

4.7.2 Item Styles

TDataGrid defines different styles applied to its items. For example, `AlternatingItemStyle` is applied to alternating items (item 2, 4, 6, etc.) Through these properties, one can set CSS style fields or CSS classes for the items.

Item styles are applied in a hierarchical way. Styles in higher hierarchy will inherit from styles in lower hierarchy. Starting from the lowest hierarchy, the item styles include item's own style, `ItemStyle`, `AlternatingItemStyle`, `SelectedItemStyle`, and `EditItemStyle`. Therefore, if background color is set as red in `ItemStyle`, `EditItemStyle` will also have red background color, unless it is explicitly set to a different value.

4.7.3 Events

TDataGrid provides several events to facilitate manipulation of its items,

- `OnItemCreated` - raised each time an item is newly created. When the event is raised, data and child controls are both available for the new item.
- `OnItemDataBound` - raised each time an item just completes databinding. When the event is raised, data and child controls are both available for the item, and the item has finished databindings of itself and all its child controls.
- `OnItemCommand` - raised when a child control of some item (such as a `TButton`) raises an `OnCommand` event.
- command events - raised when a child control's `OnCommand` event has a specific command name,
  - `OnSelectedIndexChanged` - if the command name is `select`.
  - `OnEditCommand` - if the command name is `edit`.
  - `OnDeleteCommand` - if the command name is `delete`.
  - `OnUpdateCommand` - if the command name is `update`.
  - `OnCancelCommand` - if the command name is `cancel`.
  - `OnSortCommand` - if the command name is `sort`.
  - `OnPageIndexChanged` - if the command name is `page`.

4.7.4 Using TDataGrid

Automatically Generated Columns

TDataGrid by default will create a list of columns based on the structure of the bound data. TDataGrid will read the first row of the data, extract the field names of the row, and construct a column for each field. Each column is of type `TBoundColumn`.

The following example displays a list of computer product information using a TDataGrid. Columns are automatically generated. Pay attention to how item styles are specified and inherited. The data are populated into the datagrid using the follow code, which is common among most datagrid applications,

```
public function onLoad($param) {
    parent::onLoad($param);
    if(!$this->IsPostBack) {
        $this->DataGrid->DataSource=$this->Data;
        $this->DataGrid->dataBind();
    }
}
```

Try, Controls.Samples.TDataGrid.Sample1

Manually Specified Columns

Using automatically generated columns gives a quick way of browsing tabular data. In real applications, however, automatically generated columns are often not sufficient because developers have no way customizing their appearance. Manually specified columns are thus more desirable.

To manually specify columns, set `AutoGenerateColumns` to false, and specify the columns in a template like the following,

```
<com:TDataGrid ...>
  <com:TBoundColumn DataField="name" .../>
  <com:TBoundColumn DataField="price" .../>
  <com:TEditCommandColumn ...>
```

```
  ...
</com:TDataGrid>
```

Note, if `AutoGenerateColumns` is true and there are manually specified columns, the automatically generated columns will be appended to the manually specified columns. Also note, the datagrid's `Columns` property contains only manually specified columns and no automatically generated ones.

The following example uses manually specified columns to show a list of book information,

- Book title - displayed as a hyperlink pointing to the corresponding amazon.com book page. `THyperLinkColumn` is used.
- Publisher - displayed as a piece of text using `TBoundColumn`.
- Price - displayed as a piece of text using `TBoundColumn` with output formatting string and customized styles.
- In-stock or not - displayed as a checkbox using `TCheckBoxColumn`.
- Rating - displayed as an image using `TTemplateColumn` which allows maximum freedom in specifiying cell contents.

Pay attention to how item (row) styles and column styles cooperate together to affect the appearance of the cells in the datagrid. Try, Controls.Samples.TDataGrid.Sample2

4.8 TDataGrid : Part II

4.8.1 Interacting with TDataGrid

Besides the rich data presentation functionalities as demonstrated in previous section, TDataGrid is also highly user interactive. An import usage of TDataGrid is editting or deleting rows of data. The `TBoundColumn` can adjust the associated cell presentation according to the mode of datagrid items. When an item is in browsing mode, the cell is displayed with a static text; when the item is in editting mode, a textbox is displayed to collect user inputs. TDataGrid provides `TEditCommandColumn` for switching item modes. In addition, `TButtonColumn` offers developers the flexibility of creating arbitrary buttons for various user interactions.

The following example shows how to make the previous book information table an interactive one. It allows users to edit and delete book items from the table. Two additional columns are used in the example to allow users interact with the datagrid: `TEditCommandColumn` and `TButtonColumn`.

Try, Controls.Samples.TDataGrid.Sample3

4.8.2 Sorting

TDataGrid supports sorting its items according to specific columns. To enable sorting, set `AllowSorting` to true. This will turn column headers into clickable buttons if their `SortExpression` property is not empty. When users click on the header buttons, an `OnSortCommand` event will be raised. Developers can write handlers to respond to the sort command and sort the data according to `SortExpression` which is specified in the corresponding column.

The following example turns the datagrid in Example 2 into a sortable one. Users can click on the link button displayed in the header of any column, and the data will be sorted in ascending order along that column.

Try, Controls.Samples.TDataGrid.Sample4

4.8.3 Paging

When dealing with large datasets, paging is helpful in reducing the page size and complexity. TDataGrid has an embedded pager that allows users to specify which page of data they want to see. The pager can be customized via `PagerStyle`. For example, `PagerStyle.Visible` determines whether the pager is visible or not; `PagerStyle.Position` indicates where the pager is displayed; and `PagerStyle.Mode` specifies what type of pager is displayed, a numeric one or a next-prev one.

To enable paging, set `AllowPaging` to true. The number of rows of data displayed in a page is specified by `PageSize`, while the index (zero-based) of the page currently showing to users is by `CurrentPageIndex`. When users click on a pager button, TDataGrid raises `OnPageIndexChanged` event. Typically, the event handler is written as follows,

```
public function pageIndexChanged($sender,$param) {
    $this->DataGrid->CurrentPageIndex=$param->NewPageIndex;
    $this->DataGrid->DataSource=$this->Data;
    $this->DataGrid->dataBind();
```

```
}
```

The following example enables the paging functionality of the datagrid shown in Example 1. In this example, you can set various pager styles interactively to see how they affect the pager display.

Try, Controls.Samples.TDataGrid.Sample5

Custom Paging

The paging functionality shown above requires loading all data into memory, even though only a portion of them is displayed in a page. For large datasets, this is inefficient and may not always be feasible. TDataGrid provides custom paging to solve this problem. Custom paging only requires the portion of the data to be displayed to end users.

To enable custom paging, set both `AllowPaging` and `AllowCustomPaging` to true. Notify TDataGrid the total number of data items (rows) available by setting `VirtualItemCount`. And respond to the `OnPageIndexChanged` event. In the event handler, use the `NewPageIndex` property of the event parameter to fetch the new page of data from data source. For MySQL database, this can be done by using `LIMIT` clause in an SQL select statement.

Try, Controls.Samples.TDataGrid.Sample6

4.8.4 Extending TDataGrid

Besides traditional class inheritance, extensibility of TDataGrid is mainly through developing new datagrid column components. For example, one may want to display an image column. He may use `TTemplateColumn` to accomplish this task. A better solution is to develop an image column component so that the work can be reused easily in other projects.

All datagrid column components must inherit from `TDataGridColumn`. The main method that needs to be overridden is `initializeCell()` which creates content for cells in the corresponding column. Since each cell is also in an item (row) and the item can have different types (such as `Header`, `AltneratingItem`, etc.), different content may be created according to the item type. For the image column example, one may want to create a `TImage` control within cells residing in items of `Item` and `AlterantingItem` types.

```
class ImageColumn extends TDataGridColumn {
```

```
    ...
    public function initializeCell($cell,$columnIndex,$itemType) {
        parent::initializeCell($cell,$columnIndex,$itemType);
        if($itemType==='Item' || $itemType==='AlternatingItem') {
            $image=new TImage;
            // ... customization of the image
            $cell->Controls[]=$image;
        }
    }
}
```

In `initializeCell()`, remember to call the parent implementation, as it initializes cells in items of `Header` and `Footer` types.

4.9 Writing New Controls

Writing new controls is often desired by advanced programmers, because they want to reuse the code that they write for dealing with complex presentation and user interactions.

In general, there are two ways of writing new controls: composition of existing controls and extending existing controls. They all require that the new control inherit from `TControl` or its child classes.

4.9.1 Composition of Existing Controls

Composition is the easiest way of creating new controls. It mainly involves instantiating existing controls, configuring them and making them the constituent components. The properties of the constituent components are exposed through subproperties.

One can compose a new control in two ways. One is to override the `TControl::createChildControls()` method. The other is to extend `TTemplateControl` (or its child classes) and write a control template. The latter is easier to use and can organize the layout constituent compoents more intuitively, while the former is more efficient because it does not require parsing of the template.

As an example, we show how to create a labeled textbox called `LabeledTextBox` using the above two approaches. A labeled textbox displays a label besides a textbox. We want reuse the PRADO

provided `TLabel` and `TTextBox` to accomplish this task.

Composition by Writing Templates

We need two files: a control class file named `LabeledTextBox.php` and a control template file named `LabeledTextBox.tpl`. Both must reside under the same directory.

Like creating a PRADO page, we can easily write down the content in the control template file.

```
<com:TLabel ID="Label" ForControl="TextBox" />
<com:TTextBox ID="TextBox" />
```

The above template specifies a `TLabel` control named `Label` and a `TTextBox` control named `TextBox`. We would to expose these two controls. This can be done by defining a property for each control in the `LabeledTextBox` class file. For example, we can define a `Label` property as follows,

```
class LabeledTextBox extends TTemplateControl {
    public function getLabel() {
        $this->ensureChildControls();
        return $this->getRegisteredObject('Label');
    }
}
```

In the above, the method call to `ensureChildControls()` ensures that both the label and the textbox controls are created (from template) when the `Label` property is accessed. The `TextBox` property can be implemented similarly.

Try, Controls.Samples.LabeledTextBox1.Home

Composition by Overriding `createChildControls()`

For a composite control as simple as `LabeledTextBox`, it is better to create it by extending `TControl` and overriding the `createChildControls()` method, because it does not use templates and thus saves template parsing time. Note, the new control class must implement the `INamingContainer` interface to ensure the global uniqueness of the ID of its constituent controls.

Complete code for `LabeledTextBox` is shown as follows,

```
class LabeledTextBox extends TControl implements INamingContainer {
    private $_label;
    private $_textbox;
    protected function createChildControls() {
        $this->_label=new TLabel;
        $this->_label->setID('Label');
        // add the label as a child of LabeledTextBox
        $this->getControls()->add($label);
        $this->_textbox=new TTextBox;
        $this->_textbox->setID('TextBox');
        // add the textbox as a child of LabeledTextBox
        $this->getControls()->add($textbox);
    }
    public function getLabel() {
        $this->ensureChildControls();
        return $this->_label;
    }
    public function getTextBox() {
        $this->ensureChildControls();
        return $this->_textbox;
    }
}
```

Try, Controls.Samples.LabeledTextBox2.Home

Using `LabeledTextBox`

To use `LabeledTextBox` control, first we need to include the corresponding class file. Then in a page template, we can write lines like the following,

```
<com:LabeledTextBox ID="Input" Label.Text="Username" />
```

In the above, `Label.Text` is a subproperty of `LabeledTextBox`, which refers to the `Text` property of the `Label` property. For other details of using `LabeledTextBox`, see the above online examples.

4.9.2 Extending Existing Controls

Extending existing controls is the same as conventional class inheritance. It allows developers to customize existing control classes by overriding their properties, methods, events, or creating new ones.

The difficulty of the task depends on how much an existing class needs to be customized. For example, a simple task could be to customize `TLabel` control, so that it displays a red label by default. This would merely involves setting the `ForeColor` property to `"red"` in the constructor. A difficult task would be to create controls that provide completely innovative functionalities. Usually, this requires the new controls extend from "low level" control classes, such as `TControl` or `TWebControl`.

In this section, we mainly introduce the base control classes `TControl` and `TWebControl`, showing how they can be customized. We also introduce how to write controls with specific functionalities, such as loading post data, raising post data and databinding with data source.

Extending `TControl`

`TControl` is the base class of all control classes. Two methods are of the most importance for derived control classes:

- `addParsedObject()` - this method is invoked for each component or text string enclosed within the component tag specifying the control in a template. By default, the enclosed components and text strings are added into the `Controls` collection of the control. Derived controls may override this method to do special processing about the enclosed content. For example, `TListControl` only accepts `TListItem` components to be enclosed within its component tag, and these components are added into the `Items` collection of `TListControl`.
- `render()` - this method renders the control. By default, it renders items in the `Controls` collection. Derived controls may override this method to give customized presentation.

Other important properties and methods include:

- `ID` - a string uniquely identifying the control among all controls of the same naming container. An automatic ID will be generated if the ID property is not set explicitly.

- `UnqiueID` - a fully qualified ID uniquely identifying the control among all controls on the current page hierarchy. It can be used to locate a control in the page hierarchy by calling `TControl::findControl()` method. User input controls often use it as the value of the name attribute of the HTML input element.
- `ClientID` - similar to `UniqueID`, except that it is mainly used for presentation and is commonly used as HTML element id attribute value. Do not rely on the explicit format of `ClientID`.
- `Enabled` - whether this control is enabled. Note, in some cases, if one of the control's ancestor controls is disabled, the control should also be treated as disabled, even if its `Enabled` property is true.
- `Parent` - parent control of this control. The parent control is in charge of whether to render this control and where to place the rendered result.
- `Page` - the page containing this control.
- `Controls` - collection of all child controls, including static texts between them. It can be used like an array, as it implements `Traversable` interface. To add a child to the control, simply insert it into the `Controls` collection at appropriate position.
- `Attributes` - collection of custom attributes. This is useful for allowing users to specify attributes of the output HTML elements that are not covered by control properties.
- `getViewState()` and `setViewState()` - these methods are commonly used for defining properties that are stored in viewstate.
- `saveState()` and `loadState()` - these two methods can be overriden to provide last step state saving and loading.
- Control lifecycles - Life page lifecycles, controls also have lifecycles. Each control undergoes the following lifecycles in order: constructor, `onInit()`, `onLoad()`, `onPreRender()`, `render()`, and `onUnload`. More details can be found in the page section.

Extending `TWebControl`

`TWebControl` is mainly used as a base class for controls representing HTML elements. It provides a set of properties that are common among HTML elements. It breaks the `TControl::render()` into the following methods that are more suitable for rendering an HTML element:

- `addAttributesToRender()` - adds attributes for the HTML element to be rendered. This method is often overriden by derived classes as they usually have different attributes to be rendered.
- `renderBeginTag()` - renders the openning HTML tag.
- `renderContents()` - renders the content enclosed within the HTML element. By default, it displays the items in the `Controls` collection of the control. Derived classes may override this method to render customized contents.
- `renderEndTag()` - renders the closing HTML tag.

When rendering the openning HTML tag, `TWebControl` calls `getTagName()` to obtain the tag name. Derived classes may override this method to render different tag names.

Creating Controls with Special Functionalities

If a control wants to respond to client-side events and translate them into server side events (called postback events), such as `TButton`, it has to implement the `IPostBackEventHandler` interface.

If a control wants to be able to load post data, such as `TTextBox`, it has to implement the `IPostBackDataHandler` interface.

If a control wants to get data from some external data source, it can extend `TDataBoundControl`. `TDataBoundControl` implements the basic properties and methods that are needed for populating data via databinding. In fact, controls like `TListControl`, `TRepeater` are `TDataGrid` are all derived from it.

PostBackHandler PostBackEvnetHandler DataBoundControl

Chapter 5

Security

5.1 Authentication and Authorization

Authentication is a process of verifying whether someone is who he claims he is. It usually involves a username and a password, but may include any other methods of demonstrating identity, such as a smart card, fingerprints, etc.

Authorization is finding out if the person, once identified, is permitted to manipulate specific resources. This is usually determined by finding out if that person is of a particular role that has access to the resources.

5.1.1 How PRADO Auth Framework Works

PRADO provides an extensible authentication/authorization framework. As described in application lifecycles, `TApplication` reserves several lifecycles for modules responsible for authentication and authorization. PRADO provides the `TAuthManager` module for such purposes. Developers can plug in their own auth modules easily. `TAuthManager` is designed to be used together with `TUserManager` module, which implements a read-only user database.

When a page request occurs, `TAuthManager` will try to restore user information from session. If no user information is found, the user is considered as an anonymous or guest user. To facilitate user identity verification, `TAuthManager` provides two commonly used methods: `login()` and

`logout()`. A user is logged in (verified) if his username and password entries match a record in the user database managed by `TUserManager`. A user is logged out if his user information is cleared from session and he needs to re-login if he makes new page requests.

During `Authorization` application lifecycle, which occurs after `Authentication` lifecycle, `TAuthManager` will verify if the current user has access to the requested page according to a set of authorization rules. The authorization is role-based, i.e., a user has access to a page if 1) the page explicitly states that the user has access; 2) or the user is of a particular role that has access to the page. If the user does not have access to the page, `TAuthManager` will redirect user browser to the login page which is specified by `LoginPage` property.

5.1.2 Using PRADO Auth Framework

To enable PRADO auth framework, add the `TAuthManager` module and `TUserManager` module to application configuration,

```
<service id="page" class="TPageService">
  <modules>
    <module id="auth" class="System.Security.TAuthManager"
              UserManager="users" LoginPage="UserLogin" />
    <module id="users" class="System.Security.TUserManager"
              PasswordMode="Clear">
      <user name="demo" password="demo" />
      <user name="admin" password="admin" />
    </module>
  </modules>
</service>
```

In the above, the `UserManager` property of `TAuthManager` is set to the `users` module which is `TUserManager`. Developers may replace it with a different user management module that is derived from `TUserManager`.

Authorization rules for pages are specified in page configurations as follows,

```
<authorization>
    <allow pages="PageID1,PageID2"
```

```
            users="User1,User2"
            roles="Role1" />
    <deny pages="PageID1,PageID2"
            users="?"
            verb="post" />
</authorization>
```

An authorization rule can be either an `allow` rule or a `deny` rule. Each rule consists of four optional properties:

- `pages` - list of comma-separated page names that this rule applies to. If empty or not set, this rule will apply to all pages under the current directory and all its subdirectories recursively.
- `users` - list of comma-separated user names that this rule applies to. A character * refers to all users including anonymous/guest user. And a character ? refers to anonymous/guest user.
- `roles` - list of comma-separated user roles that this rule applies to.
- `verb` - page access method that this rule applies to. It can be either `get` or `post`. If empty or not set, the rule applies to both methods.

When a page request is being processed, a list of authorization rules may be available. However, only the *first effective* rule *matching* the current user will render the authorization result.

- Rules are ordered bottom-up, i.e., the rules contained in the configuration of current page folder go first. Rules in configurations of parent page folders go after.
- A rule is effective if the current page is in the listed pages of the rule AND the current user action (`get` or `post`) is in the listed actions.
- A rule matching occurs if the current user name is in the listed user names of an *effective* rule OR if the user's role is in the listed roles of that rule.
- If no rule matches, the user is authorized.

In the above example, anonymous users will be denied from posting to `PageID1` and `PageID2`, while `User1` and `User2` and all users of role `Role1` can access the two pages (in both `get` and `post` methods).

5.1.3 Using `TUserManager`

As aforementioned, `TUserManager` implements a read-only user database. The user information are specified in either application configuration or an external XML file.

We have seen in the above example that two users are specified in the application configuration. Complete syntax of specifying the user and role information is as follows,

```
<user name="demo" password="demo" roles="demo,admin" />
<role name="admin" users="demo,demo2" />
```

where the `roles` attribute in `user` element is optional. User roles can be specified in either the `user` element or in a separate `role` element.

5.2 Viewstate Protection

Viewstate lies at the heart of PRADO. Viewstate represents data that can be used to restore pages to the state that is last seen by end users before making the current request. By default, PRADO uses hidden fields to store viewstate information.

It is extremely important to ensure that viewstate is not tampered by end users. Without protection, malicious users may inject harmful code into viewstate and unwanted instructions may be performed when page state is being restored on server side.

To prevent viewstate from being tampered, PRADO enforces viewstate HMAC (Keyed-Hashing for Message Authentication) check before restoring viewstate. Such a check can detect if the viewstate has been tampered or not by end users. Should the viewstate modifies, PRADO simply stops restoring the viewstate and returns an error message.

HMAC check requires a private key that should be secret to end users. Developers can either manually specify a key or let PRADO automatically generate a key. Manually specified key is useful when the application runs on a server farm. To do so, configure `TPageStatePersister` in application configuration,

```
<service id="page" class="TPageService">
    <modules>
```

```
        <module id="state"
            class="TPageStatePersister"
            PrivateKey="my private key" />
    </modules>
</service>
```

HMAC check does not prevent end users from reading the viewstate content. An added security measure is to encrypt the viewstate information so that end users cannot decipher it. Work on supporting viewstate encryption is ongoing.

Another strategy to protect viewstate is to store it on server side rather than using hidden fields. The relevant work is also ongoing.

5.3 Cross Site Scripting Prevention

Cross site scripting (also known as XSS) occurs when a web application gathers malicious data from a user. Often attackers will inject JavaScript, VBScript, ActiveX, HTML, or Flash into a vulnerable application to fool other application users and gather data from them. For example, a poorly design forum system may display user input in forum posts without any checking. An attacker can then inject a piece of malicious JavaScript code into a post so that when other users read this post, the JavaScript runs unexpectedly on their computers.

One of the most important measures to prevent XSS attacks is to check user input before displaying them. One can do HTML-encoding with the user input to achieve this goal. However, in some situations, HTML-encoding may not be preferrable because it disables all HTML tags.

PRADO incorporates the work of SafeHTML and provides developers with a useful component called `TSafeHtml`. By enclosing content within a `TSafeHtml` component tag, the enclosed content are ensured to be safe to end users. In addition, the commonly used `TTextBox` has a `SafeText` property which contains user input that are ensured to be safe if displayed directly to end users.

Chapter 6

Advanced Topics

6.1 Assets

Assets are resource files (such as images, sounds, videos, CSS stylesheets, javascripts, etc.) that belong to specific component classes. Assets are meant to be provided to Web users. For better reusability and easier deployment of the corresponding component classes, assets should reside together with the component class files . For example, a toggle button may use two images, stored in file `down.gif` and `up.gif`, to show different toggle states. If we require the image files be stored under `images` directory under the Web server document root, it would be inconvenient for the users of the toggle button component, because each time they develop or deploy a new application, they would have to manually copy the image files to that specific directory. To eliminate this requirement, a directory relative to the component class file should be used for storing the image files. A common strategy is to use the directory containing the component class file to store the asset files.

Because directories containing component class files are normally inaccessible by Web users, PRADO implements an asset publishing scheme to make available the assets to Web users. An asset, after being published, will have a URL by which Web users can retrieve the asset file.

6.1.1 Asset Publishing

PRADO provides several methods for publishing assets or directories containing assets:

- In a template file, you can use asset tags to publish assets and obtain their URLs. Note, the assets must be relative to the directory containing the template file.
- In PHP code, you can call `TControl::getAsset($relativePath)` to publish an asset and get its URL. The asset file or directory must be relative to the directory containing the control class file.
- If you want to publish an arbitrary asset, you need to call `TAssetManager::publishFilePath($path)`.

BE AWARE: Be very careful with assets publishing, because it gives Web users access to files that were previously inaccessible to them. Make sure that you do not publish files that do not want Web users to see.

6.1.2 Customization

Asset publishing is managed by the `System.Web.UI.TAssetManager` module. By default, all published asset files are stored under the `[AppEntryPath]/assets` directory, where `AppEntryPath` refers to the directory containing the application entry script. Make sure the `assets` directory is writable by the Web server process. You may change this directory to another by configuring the `BasePath` and `BaseUrl` properties of the `System.Web.UI.TAssetManager` module in application configuration,

```
<service id="page" class="TPageService">
    <modules>
        <module id="asset"
                class="System.Web.UI.TAssetManager"
                BasePath="images"
                BaseUrl="images" />
    </modules>
</service>
```

6.1.3 Performance

PRADO uses caching techniques to ensure the efficiency of asset publishing. Publishing an asset essentially requires file copy operation, which is expensive. To save unnecessary file copy operations, `System.Web.UI.TAssetManager` only publishes an asset when it has a newer file modification time

than the published file. When an application runs under the `Performance` mode, such timestamp checkings are also omitted.

ADVISORY: Do not overuse asset publishing. The asset concept is mainly used to help better reuse and redistribute component classes. Normally, you should not use asset publishing for resources that are not bound to any components in an application. For example, you should not use asset publishing for images that are mainly used as design elements (e.g. logos, background images, etc.) Let Web server to directly serve these images will help improve the performance of your application.

6.1.4 A Toggle Button Example

We now use the toggle button example to explain the usage of assets. The control uses two image files `up.gif` and `down.gif`, which are stored under the directory containing the control class file. When the button is in `Up` state, we would like to show the `up.gif` image. This can be done as follows,

```
class ToggleButton extends TWebControl {
    ...
    protected function addAttributesToRender($writer) {
        ...
        if($this->getState()===’Up’) {
            $url=$this->getAsset(’up.gif’);
            $writer->addAttribute(’src’,$url);
        }
        ...
    }
    ...
}
```

In the above, the call `$this->getAsset('up.gif')` will publish the `up.gif` image file and return a URL for the published image file. The URL is then rendered as the `src` attribute of the HTML image tag.

To redistribute `ToggleButton`, simply pack together the class file and the image files. Users of `ToggleButton` merely need to unpack the file, and they can use it right away, without worrying about where to copy the image files to.

6.2 Master and Content

Pages in a Web application often share common portions. For example, all pages of this tutorial application share the same header and footer portions. If we repeatedly put header and footer in every page source file, it will be a maintenance headache if in future we want to something in the header or footer. To solve this problem, PRADO introduces the concept of master and content. It is essentially a decorator pattern, with content being decorated by master.

Master and content only apply to template controls (controls extending `TTemplateControl` or its child classes). A template control can have at most one master control and one or several contents (each represented by a `TContent` control). Contents will be inserted into the master control at places reserved by `TContentPlaceHolder` controls. And the presentation of the template control is that of the master control with `TContentPlaceHolder` replaced by `TContent`.

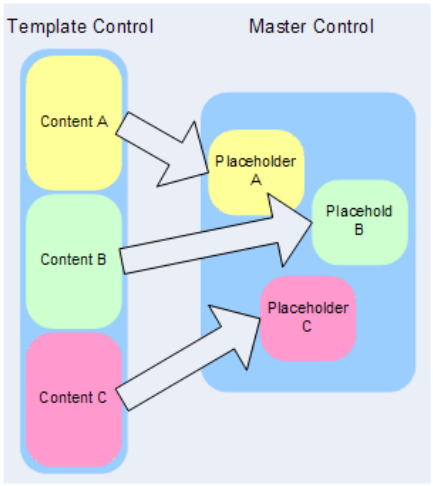
For example, assume a template control has the following template:

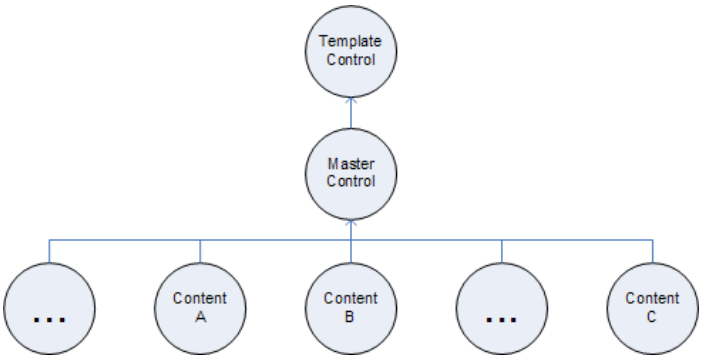
```
<%@ MasterClass="MasterControl" %>
<com:TContent ID="A" >
content A
</com:TContent >
<com:TContent ID="B" >
content B
</com:TContent >
<com:TContent ID="B" >
content B
</com:TContent >
```

which uses `MasterControl` as its master control. The master control has the following template,

```
other stuff
<com:TContentPlaceHolder ID="A" />
other stuff
<com:TContentPlaceHolder ID="B" />
other stuff
<com:TContentPlaceHolder ID="C" />
other stuff
```

Then, the contents are inserted into the master control according to the following diagram, while the resulting parent-child relationship can be shown in the next diagram. Note, the template control discards everything in the template other than the contents, while the master control keeps everything and replaces the content placeholders with the contents according to ID matching.





6.3 Themes and Skins

6.3.1 Introduction

Themes in Prado provide a way for developers to provide a consistent look-and-feel across an entire web application. A theme contains a list of initial values for properties of various control types. When applying a theme to a page, all controls on that page will receive the corresponding initial property values from the theme. This allows themes to interact with the rich property sets

of the various PRADO controls, meaning that themes can be used to specify a large range of presentational properties that other theming methods (e.g. CSS) cannot. For example, themes could be used to specify the default page size of all data grids across an application by specifying a default value for the `PageSize` property of the `TDataGrid` control.

6.3.2 Understanding Themes

A theme is a directory consists of skin files, javascript files and CSS files. Any javascript or CSS files contained in a theme will be registered with the page that the theme is applied to. A skin is a set of initial property values for a particular control type. A control type may have one or several skins, each identified by a unqiue `SkinID`. When applying a theme to a page, a skin is applied to a control if the control type and the `SkinID` value both match to those of the skin. Note, if a skin has an empty `SkinID` value, it will apply to all controls of the particular type whose `SkinID` is not set or empty. A skin file consists of one or several skins, for one or several control types. A theme is the union of skins defined in all skin files.

6.3.3 Using Themes

To use a theme, you need to set the `Theme` property of the page with the theme name, which is the theme directory name. You may set it in either page configurations or in the constructor or `onPreInit()` method of the page. You cannot set the property after `onPreInit()` because by that time, child controls of the page are already created (skins must be applied to controls right after they are created.)

To use a particular skin in the theme for a control, set `SkinID` property of the control in template like following,

```
<com:TButton SkinID="Blue" ... />
```

This will apply the 'Blue' skin to the button. Note, the initial property values specified by the 'Blue' skin will overwrite any existing property values of the button. Use stylesheet theme if you do not want them to be overwritten. To use stylesheet theme, set the `StyleSheetTheme` property of the page instead of `Theme` (you can have both `StyleSheetTheme` and `Theme`).

To use the javascript files and CSS files contained in a theme, a `THead` control must be placed on the page template. This is because the theme will register those files with the page and `THead` is

the right place to load those files.

6.3.4 Theme Storage

All themes by default must be placed under the `[AppEntryPath]/themes` directory, where `AppEntryPath` refers to the directory containing the application entry script. If you want to use a different directory, configure the `BasePath` and `BaseUrl` properties of the `System.Web.UI.TThemeManager` module in application configuration,

```
<service id="page" class="TPageService">
    <modules>
        <module id="theme"
                class="System.Web.UI.TThemeManager"
                BasePath="mythemes"
                BaseUrl="mythemes" />
    </modules>
</service>
```

6.3.5 Creating Themes

Creating a theme involves creating the theme directory and writing skin files (and possibly javascript and CSS files). The name of skin files must be terminated with `.skin`. The format of skin files are the same as that of control template files. Since skin files do not define parent-child presentational relationship among controls, you cannot place a component tag within another. And any static texts between component tags are discarded. To define the aforementioned 'Blue' skin for `TButton`, write the following in a skin file,

```
<com:TButton SkinID="Blue" BackColor="blue" />
```

As aforementioned, you can put several skins within a single skin file, or split them into several files. A commonly used strategy is that each skin file only contains skins for one type of controls. For example, `Button.skin` would contain skins only for the `TButton` control type.

6.4 Persistent State

Web applications often need to remember what an end user has done in previous page requests so that the new page request can be served accordingly. State persistence is to address this problem. Traditionally, if a page needs to keep track of user interactions, it will resort to session, cookie, or hidden fields. PRADO provides a new line of state persistence schemes, including view state, control state, and application state.

6.4.1 View State

View state lies at the heart of PRADO. With view state, Web pages become stateful and are capable of restoring pages to the state that end users interacted with before the current page request. Web programming thus resembles to Windows GUI programming, and developers can think continuously without worrying about the roundtrips between end users and the Web server. For example, with view state, a textbox control is able to detect if the user input changes the content in the textbox.

View state is only available to controls. View state of a control can be disabled by setting its `EnableViewState` property to false. To store a variable in view state, call the following,

```
$this->setViewState('Caption',$caption);
```

where `$this` refers to the control object, `Caption` is a unique key identifying the `$caption` variable stored in viewstate. To retrieve the variable back from view state, call the following,

```
$caption = $this->getViewState('Caption');
```

6.4.2 Control State

Control state is like view state in every aspect except that control state cannot be disabled. Control state is intended to be used for storing crucial state information without which a page or control may not work properly.

To store and retrieve a variable in control state, use the followign commands,

```
$this->setControlState('Caption',$caption);
```

```
$caption = $this->getControlState('Caption');
```

6.4.3 Application State

Application state refers to data that is persistent across user sessions and page requests. A typical example of application state is the user visit counter. The counter value is persistent even if the current user session terminates. Note, view state and control state are lost if the user requests for a different page, while session state is lost if the user session terminates.

To store and retrieve a variable in application state, use the followign commands,

```
$application->setGlobalState('Caption',$caption);
$caption = $application->getGlobalState('Caption');
```

6.4.4 Session State

PRADO encapsulates the traditional session management in `THttpSession` module. The module can be accessed from within any component by using `$this->Session`, where `$this` refers to the component object.

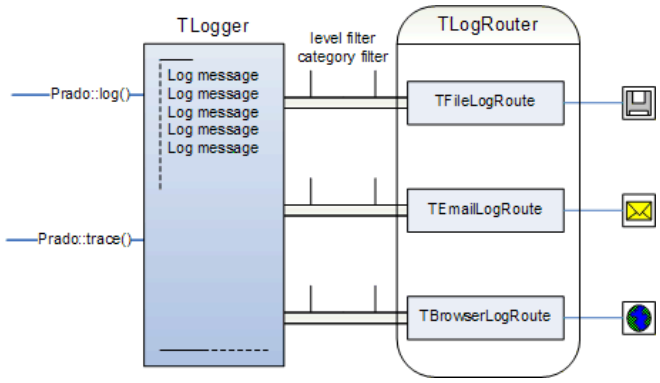
6.5 Logging

PRADO provides a highly flexible and extensible logging functionality. Messages logged can be classified according to log levels and message categories. Using level and category filters, the messages can be further routed to different destinations, such as files, emails, browser windows, etc. The following diagram shows the basic architecture of PRADO logging mechanism,

6.5.1 Using Logging Functions

The following two methods are provided for logging messages in PRADO,

```
Prado::log($message, $logLevel, $category);
Prado::trace($message, $category);
```



The difference between `Prado::log()` and `Prado::trace()` is that the latter automatically selects the log level according to the application mode. If the application is in `Debug` mode, stack trace information is appended to the messages. `Prado::trace()` is widely used in the core code of the PRADO framework.

6.5.2 Message Routing

Messages logged using the above two functions are kept in memory. To make use of the messages, developers need to route them to specific destinations, such as files, emails, or browser windows. The message routing is managed by `System.Log.TLogRouter` module. When plugged into an application, it can route the messages to different destination in parallel. Currently, PRADO provides three types of routes:

- `TFileLogRoute` - filtered messages are stored in a specified log file. By default, this file is named `prado.log` under the runtime directory of the application. File rotation is provided.
- `TEmailLogRoute` - filtered messages are sent to pre-specified email addresses.
- `TBrowserLogRoute` - filtered messages are appended to the end of the current page output.

To enable message routing, plug in and configure the `TLogRouter` module in application configuration,

```
<module id="log" class="System.Log.TLogRouter">
  <route class="TBrowserLogRoute"
      Levels="Info"
```

```
      Categories="System.Web.UI.TPage, System.Web.UI.WebControls" />
  <route class="TFileLogRoute"
      Levels="Warning, Error"
      Categories="System.Web" />
</module>
```

In the above, the `Levels` and `Categories` specify the log and category filters to selectively retrieve the messages to the corresponding destinations.

6.5.3 Message Filtering

Messages can be filtered according to their log levels and categories. Each log message is associated with a log level and a category. With levels and categories, developers can selectively retrieve messages that they are interested on.

Log levels defined in `System.Log.TLogger` include : `DEBUG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `ALERT`, `FATAL`. Messages can be filtered according log level criteria. For example, if a filter specifies `WARNING` and `ERROR` levels, then only those messages that are of `WARNING` and `ERROR` will be returned.

Message categories are hierarchical. A category whose name is the prefix of another is said to be the ancestor category of the other category. For example, `System.Web` category is the ancestor of `System.Web.UI` and `System.Web.UI.WebControls` categories. Messages can be selectively retrieved using such hierarchical category filters. For example, if the category filter is `System.Web`, then all messages in the `System.Web` are returned. In addition, messages in the childd categories, such as `System.Web.UI.WebControls`, are also returned.

By convention, the messages logged in the core code of PRADO are categorized according to the namespace of the corresponding classes. For example, messsages logged in `TPage` will be of category `System.Web.UI.TPage`.

6.6 Internationalization (I18N) and Localization (L10N)

Many web application built with PHP will not have internationalization in mind when it was first written. It may be that it was not intended for use in languages and cultures. Internationalization is an important aspect due to the increase adoption of the Internet in many non-English speaking

countries. The process of internationalization and localization will contain difficulties. Below are some general guidelines to internationalize an existing application.

6.6.1 Separate culture/locale sensitive data

Identify and separate data that varies with culture. The most obvious are text/string/message. Other type of data should also be considered. The following list categorize some examples of culture sensitive data

- Strings, Messages, Text, in relatively small units (e.g. phrases, sentences, paragraphs, but not the full text of a book).
- Labels on buttons.
- Help files, large units of text, static text.
- Sounds.
- Colors.
- Graphics,Icons.
- Dates, Times.
- Numbers, Currency, Measurements.
- Phone numbers.
- Honorifics and personal titles.
- Postal address.
- Page layout.

If possible all manner of text should be isolated and store in a persistence format. These text include, application error messages, hard coded strings in PHP files, emails, static HTML text, and text on form elements (e.g. buttons).

6.6.2 Configuration

To enable the localization features in Prado, you need to add a few configuration options in your application configuration. First you need to include the `System.I18N.*` namespace to your paths.

```
<paths>
    <using namespace="System.I18N.*" />
</paths>
```

Then, if you wish to translate some text in your application, you need to add one translation message data source.

```
<module id="globalization" class="TGlobalization">
    <translation type="XLIFF"
        source="MyApp.messages"
        autosave="true" cache="true" />
</module>
```

Where `source` in `translation` is the dot path to a directory where you are going to store your translate message catalogue. The `autosave` attribute if enabled, saves untranslated messages back into the message catalogue. With `cache` enabled, translated messages are saved in the application `runtime/i18n` directory.

With the configuration complete, we can now start to localize your application. If you have `autosave` enabled, after running your application with some localization activity (i.e. translating some text), you will see a directory and a `messages.xml` created within your `source` directory.

6.6.3 What to do with `messages.xml`?

The translation message catalogue file, if using `type="XLIFF"`, is a standardized translation message interchange XML format. You can edit the XML file using any UTF-8 aware editor. The format of the XML is something like the following.

```
<?xml version="1.0"?>
<xliff version="1.0">
    <file original="I18N Example IndexPage"
```

```
          source-language="EN"
          datatype="plaintext"
          date="2005-01-24T11:07:53Z">
        <body>

<trans-unit id="1">
<source>Hello world.</source>
<target>Hi World!!!</target>
</trans-unit>

        </body>
    </file>
</xliff>
```

Each translation message is wrapped within a `trans-unit` tag, where `source` is the original message, and `target` is the translated message. Editors such as Heartsome XLIFF Translation Editor can help in editing these XML files.

6.6.4 Setting and Changing Culture

Once globalization is enabled, you can access the globalization settings, such as, `Culture`, `Charset`, etc, using

```
$globalization = $this->getApplication()->getGlobalization();
echo $globalization->Culture;
$globalization->Charset= "GB-2312"; //change the charset
```

You also change the way the culture is determined by changing the `class` attribute in the module configuration. For example, to set the culture that depends on the browser settings, you can use the `TGlobalizationAutoDetect` class.

```
<module id="globalization" class="TGlobalizationAutoDetect">
   ...
</module>
```

You may also provide your own globalization class to change how the application culture is set. Lastly, you can change the globalization settings on page by page basis using template control tags. For example, changing the `Culture` to "zh".

```
<%@ Application.Globalization.Culture="zh" %>
```

6.6.5 Localizing your Prado application

There are two areas in your application that may need message or string localization, in PHP code and in the templates. To localize strings within PHP, use the `localize` function detailed below. To localize text in the template, use the TTranslate component.

6.6.6 Using `localize` function to translate text within PHP

The `localize` function searches for a translated string that matches original from your translation source. First, you need to locate all the hard coded text in PHP that are displayed or sent to the end user. The following example localizes the text of the `$sender` (assuming, say, the sender is a button). The original code before localization is as follows.

```
function clickMe($sender,$param)
{
  $sender->Text="Hello, world!";
}
```

The hard coded message "Hello, world!" is to be localized using the `localize` function.

```
function clickMe($sender,$param)
{
  $sender->Text=localize("Hello, world!");
}
```

6.6.7 Compound Messages

Compound messages can contain variable data. For example, in the message "There are 12 users online.", the integer 12 may change depending on some data in your application. This is difficult

to translate because the position of the variable data may be difference for different languages. In addition, different languages have their own rules for plurals (if any) and/or quantifiers. The following example can not be easily translated, because the sentence structure is fixed by hard coding the variable data within message.

```
$num_users = 12;
$message = "There are " . $num_users . " users online.";
```

This problem can be solved using the `localize` function with string substitution. For example, the `$message` string above can be constructed as follows.

```
$num_users = 12;
$message = localize("There are {num_users} users online.", array('num_users'=>$num_users));
```

Where the second parameter in `localize` takes an associative array with the key as the substitution to find in the text and replaced it with the associated value. The `localize` function does not solve the problem of localizing languages that have plural forms, the solution is to use TChoiceFormat.

6.7 I18N Components

6.7.1 TTranslate

Messages and strings can be localized in PHP or in templates. To translate a message or string in the template, use `TTranslate`.

```
<com:TTranslate>Hello World</com:TTranslate>
<com:TTranslate Text="Goodbye" />
```

`TTranslate` can also perform string substitution. Any attributes of `TTranslate` will be substituted with {`attribute name`} in the translation. E.g.

```
<com:TTranslate time="late">
The time is {time}.
</com:TTranslate>
```

A short for `TTranslate` is also provided using the following syntax.

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

where string will be translated to different languages according to the end-user's language preference. This syntax can be used with attribute values as well.

```
<com:TLabel Text="<%[ Hello World! ]%>" />
```

6.7.2 TDateFormat

Formatting localized date and time is straight forward.

```
<com:TDateFormat Value="12/01/2005" />
```

There are of 4 localized date patterns and 4 localized time patterns. They can be used in any combination. If using a combined pattern, the first must be the date, followed by a space, and lastly the time pattern. For example, full date pattern with short time pattern.

```
<com:TDateFormat Pattern="fulldate shorttime" />
```

If the `Value` property is not specified, the current date and time is used.

6.7.3 TNumberFormat

PRADO's Internationalization framework provide localized currency formatting and number formatting. Please note that the `TNumberFormat` component provides formatting only, it does not perform current conversion or exchange.

```
<com:TNumberFormat Type="currency" Value="100" />
```

`Culture` and `Currency` properties may be specified to format locale specific numbers.

6.7.4 TTranslateParameter

Compound messages, i.e., string substitution, can be accomplished with `TTranslateParameter`. In the following example, the strings "{greeting}" and "{name}" will be replace with the values of "Hello" and "World", respectively.The substitution string must be enclose with "{" and "}". The parameters can be further translated by using `TTranslate`.

```
<com:TTranslate>
  {greeting} {name}!
  <com:TTranslateParameter Key="name">World</com:TTranslateParameter>
  <com:TTranslateParameter Key="greeting">Hello</com:TTranslateParameter>
</com:TTranslate>
```

6.7.5 TChoiceFormat

Using the `localize` function or `TTranslate` component to translate messages does not inform the translator the cardinality of the data required to determine the correct plural structure to use. It only informs them that there is a variable data, the data could be anything. Thus, the translator will be unable to determine with respect to the substitution data the correct plural, language structure or phrase to use . E.g. in English, to translate the sentence, "There are number of apples.", the resulting translation should be different depending on the `number` of apples.

The `TChoiceFormat` component performs message/string choice translation. The following example demonstrated a simple 2 choice message translation.

```
<com:TChoiceFormat Value="1"/>[1] One Apple. |[2] Two Apples</com:TChoiceFormat>
```

In the above example, the `Value` "1" (one), thus the translated string is "One Apple". If the `Value` was "2", then it will show "Two Apples".

The message/string choices are separated by the pipe "—" followed by a set notation of the form.

- `[1,2]` – accepts values between 1 and 2, inclusive.
- `(1,2)` – accepts values between 1 and 2, excluding 1 and 2.
- {`1,2,3,4`} – only values defined in the set are accepted.

- `[-Inf,0)` – accepts value greater or equal to negative infinity and strictly less than 0

Any non-empty combinations of the delimiters of square and round brackets are acceptable. The string chosen for display depends on the `Value` property. The `Value` is evaluated for each set until the `Value` is found to belong to a particular set.

6.8 Error Handling and Reporting

PRADO provides a complete error handling and reporting framework based on the PHP 5 exception mechanism.

6.8.1 Exception Classes

Errors occur in a PRADO application may be classified into three categories: those caused by PHP script parsing, those caused by wrong code (such as calling an undefined function, setting an unknown property), and those caused by improper use of the Web application by client users (such as attempting to access restricted pages). PRADO is unable to deal with the first category of errors because they cannot be caughted in PHP code. PRADO provides an exception hierarchy to deal with the second and third categories.

All errors in PRADO applications are represented as exceptions. The base class for all PRADO exceptions is `TException`. It provides the message internationalization functionality to all system exceptions. An error message may be translated into different languages according to the user browser's language preference.

Exceptions raised due to improper usage of the PRADO framework inherit from `TSystemException`, which can be one of the following exception classes:

- `TConfigurationException` - improper configuration, such as error in application configuration, control templates, etc.
- `TInvalidDataValueException` - data value is incorrect or unexpected.
- `TInvalidDataTypeException` - data type is incorrect or unexpected.
- `TInvalidDataFormatException` - format of data is incorrect.

- `TInvalidOperationException` - invalid operation request.
- `TPhpErrorException` - caughtable PHP errors, warnings, notices, etc.
- `TSecurityException` - errors related with security.
- `TIOException` - IO operation error, such as file open failure.
- `TDBException` - errors related with database operations.
- `TNotSupportedException` - errors caused by requesting for unsupported feature.
- `THttpException` - errors to be displayed to Web client users.

Errors due to improper usage of the Web application by client users inherit from `TApplicationException`.

6.8.2 Raising Exceptions

Raising exceptions in PRADO has no difference than raising a normal PHP exception. The only thing matters is to raise the right exception. In general, exceptions meant to be shown to application users should use `THttpException`, while exceptions shown to developers should use other exception classes.

6.8.3 Error Capturing and Reporting

Exceptions raised during the runtime of PRADO applications are captured by `System.Exceptions.TErrorHandler` module. Different output templates are used to display the captured exceptions. `THttpException` is assumed to contain error messages that are meant for application end users and thus uses a specific group of templates. For all other exceptions, a common template shown as follows is used for presenting the exceptions.

6.8.4 Customizing Error Display

Developers can customize the presentation of exception messages. By default, all error output templates are stored under `framework/Exceptions/templates`. The location can be changed by configuring `TErrorHandler` in application configuration,


```
<module id="error"
    class="TErrorHandler">
    ErrorTemplatePath="Application.ErrorTemplates" />
```

`THttpException` uses a set of templates that are differentiated according to different `StatusCode` property value of `THttpException`. `StatusCode` has the same meaning as the status code in HTTP protocol. For example, a status code equal to 404 means the requested URL is not found on the server. The `StatusCode` value is used to select which output template to use. The output template files use the following naming convention:

```
error<status code>-<language code>.html
```

where `status code` refers to the `StatusCode` property value of `THttpException`, and `language code` must be a valid language such as `en`, `zh`, `fr`, etc. When a `THttpException` is raised, PRADO will select an appropriate template for displaying the exception message. PRADO will first locate a template file whose name contains the status code and whose language is preferred by the client browser window. If such a template is not present, it will look for a template that has the same status code but without language code.

The naming convention for the template files used for all other exceptions is as follows,

```
exception-<language code>.html
```

Again, if the preferred language is not found, PRADO will try to use `exception.html`, instead.

CAUTION: When saving a template file, please make sure the file is saved using UTF-8 encoding. On Windows, you may use `Notepad.exe` to accomplish such saving.

6.9 Performance Tuning

Performance of Web applications is affected by many factors. Database access, file system operations, network bandwidth are all potential affecting factors. PRADO tries in every effort to reduce the performance impact caused by the framework.

6.9.1 Caching

PRADO provides a generic caching technique used by in several core parts of the framework. For example, when caching is enabled, `TTemplateManager` will save parsed templates in cache and reuse them in the following requests, which saves time for parsing templates. The `TThemeManager` adopts the similar strategy to deal with theme parsing.

Enabling caching is very easy. Simply add the cache module in the application configuration, and PRADO takes care of the rest.

```
<modules>
    <module id="cache" class="System.Data.TSqliteCache" />
</modules>
```

Developers can also take advantage of the caching technique in their applications. The `Cache` property of `TApplication` returns the plugged-in cache module when it is available. To save and retrieve a data item in cache, use the following commands,

```
if($application->Cache) {
    // saves data item in cache
    $application->Cache->set($keyName,$dataItem);
    // retrieves data item from cache
    $dataItem=$application->Cache->get($keyName);
}
```

where `$keyName` should be a string that uniquely identifies the data item stored in cache.

6.9.2 Using `pradolite.php`

Including many PHP script files may impact application performance significantly. PRADO classes are stored in different files and when processing a page request, it may require including tens of class files.To alleviate this problem, in each PRADO release, a file named `pradolite.php` is also included. The file is a merge of all core PRADO class files with comments being stripped off and message logging removed.

To use `pradolite.php`, in your application entry script, replace the inclusion of `prado.php` with `pradolite.php`.

6.9.3 Changing Application Mode

Application mode also affects application performance. A PRADO application can be in one of the following modes: `Off`, `Debug`, `Normal` and `Performance`. The `Debug` mode should mainly be used during application development, while `Normal` mode is usually used in early stage after an application is deployed to ensure everything works correctly. After the application is proved to work stably for some period, the mode can be switched to `Performance` to further improve the performance.

The difference between `Debug`, `Normal` and `Performance` modes is that under `Debug` mode, application logs will contain debug information, and under `Performance` mode, timestamp checking is not performed for cached templates and published assets. Therefore, under `Performance` mode, application may not run properly if templates or assets are modified. Since `Performance` mode is mainly used when an application is stable, change of templates or assets are not likely.

To switch application mode, configure it in application configuration:

```
<application Mode="Performance" >
    ......
</application >
```

6.9.4 Reducing Page Size

By default, PRADO stores page state in hidden fields of the HTML output. The page state could be very large in size if complex controls, such as `TDataGrid`, is used. To reduce the size of the

network transmitted page size, two strategies can be used.

First, you may disable viewstate by setting `EnableViewState` to false for the page or some controls on the page if they do not need user interactions. Viewstate is mainly used to keep track of page state when a user interacts with that page.

Second, you may use a different page state storage. For example, page state may be stored in session, which essentially stores page state on the server side and thus saves the network transmission time. The module responsible for page state storage is `System.Web.UI.TPageStatePersister`, which uses hidden fields as persistent storage. To use your own storage, configure the module in application configuration as follows,

```
<service id="page" class="TPageService">
    <modules>
        <module id="state" class="MyPageStatePersister" />
    </modules>
</service>
```

6.9.5 Other Techniques

Server caching techniques are proven to be very effective in improving the performance of PRADO applications. For example, we have observed that by using Zend Optimizer, the RPS (request per second) of a PRADO application can be increased by more than ten times. Of course, this is at the cost of stale output, while PRADO's caching techniques always ensure the correctness of the output.