PRADO v3.1.4 Quickstart Tutorial [1]

Qiang Xue and Wei Zhuo

January 11, 2009

Contents

Preface

Prado quick start doc

License

PRADO is free software released under the terms of the following BSD 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 PRADO Group 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

¡div id=”intro” class=”block-content”¿

This Quickstart tutorial is provided to help you quickly start building your own Web applications based on PRADO version 3.x.

If you are an existing PRADO 3.x user and would like to learn what enhancements are available for each new version, please check out the new features page. Otherwise, the following sections are helpful for newbies.

¡!– ¡div class=”start-page”¿ ¡div class=”concepts start-block”¿

1.1.1 How PRADO Works

Concepts and fundamentals

1. Building web applications with PRADO
2. Web controls and events
3. Validating user input

4. Connecting to your database
5. Displaying data from database

¡/div¿ ¡div class=”examples start-block”¿

1.1.2 Examples and Demos

- Hello World
- Currency Converter
- Address Book
- Blog
- AJAX Chat
- Project Time Tracker

More examples in Standard Controls, Validation Controls and Data Controls. ¡/div¿

¡div class=”tutorials start-block”¿

1.1.3 Tutorials and Help

- Hello World in detail
- Currency Converter Tutorial
- Address Book Tutorial
- AJAX Chat Tutorial

¡/div¿ ¡/div¿ –¿

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

¡/div¿

- PRADO Classes

- 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.

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.

To achieve the above goal, PRADO stipulates a protocol of writing and using components to construct Web applications. A component is a software unit that is self-contained and can be reused with trivial customization. New components can be created by simple composition of existing components.

To facilitate interacting with components, 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.

In summary, 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.

1.2.1 Why PRADO?

PRADO is mostly quoted as a unique framework. In fact, it is so unique that it may turn your boring PHP programming into a fun task. The following list is a short summary of the main features of PRADO,

- Reusability - Code following the PRADO component protocol are highly reusable. This benefits development teams in the long run as they can reuse their previous work and integrate other parties' work easily.
- Event-driven programming - End-user activities, such as clicking on a submit button, are captured as server events so that developers have better focus on dealing with user interactions.
- Team integration - Presentation and logic are separately stored. PRADO applications are themable.
- Powerful Web controls - PRADO comes with a set of powerful components dealing with Web user interfaces. Highly interactive Web pages can be created with a few lines of code. For example, using the datagrid component, one can quickly create a page presenting a data table which allows paging, sorting, editing, and deleting rows of the data.
- Strong database support - Since version 3.1, PRADO has been equipped with complete database support which is natively written and thus fits seemlessly with the rest part of the PRADO framework. According to the complexity of the business objects, one can choose to use the simple PDO-based data access, or the widely known active record, or the complete business object mapping scheme SqlMap.
- Seamless AJAX support - Using AJAX in PRADO has never been easier with its innovative active controls introduced since version 3.1. You can easily write an AJAX-enabled application without writing a single line of javascript code. In fact, using active controls is not much different from using the regular non-AJAX enabled Web controls.
- I18N and L10N support - PRADO includes complete support for building applications with multiple languages and locales.
- XHTML compliance - Web pages generated by PRADO are XHTML-compliant.
- Accommodation of existing work - PRADO is a generic framework with focus on the presentational layer. It does not exclude developers from using most existing class libraries or toolkits. For example, one can AdoDB or Creole to deal with DB in his PRADO application.

- Other features - Powerful error/exception handling and message logging; generic caching and selective output caching; customizable and localizable error handling; extensible authentication and authorization; security measures such as cross-site script (XSS) prevention, cookie protection, etc.

1.2.2 What Is PRADO Best For?

PRADO is best suitable for creating Web applications that are highly user-interactive. It can be used to develop systems as simple as a blog system to those as complex as a content management system (CMS) or a complete e-commerce solution. Because PRADO promotes object-oriented programming through its component-based methodology, it fits extremely well for team work and enterprise development.

PRADO comes with a complete set of caching techniques which help accelerate PRADO Web applications to accommodate high traffic requirement. Its modular architecture allows developers to use or plug in different cache modules for different needs. The output caching enables one to selectively choose to cache part of a rendered Web page.

1.2.3 How Is PRADO Compared with Other Frameworks?

PRADO is often quoted as a unique framework. Its uniqueness mainly lies in the component-based and event-driven programming paradigm that it tries to promote. Although this programming paradigm is not new in desktop application programming and not new in a few Web programming languages, PRADO is perhaps the first PHP framework enabling it.

Most PHP frameworks mainly focuses on separating presentation and logic and promotes the MVC (model-view-controller) design pattern. PRADO achieves the same goal naturally by requiring logic be stored in classes and presentation in templates. PRADO does much more on aspects other than MVC. It fills lot of blank area in PHP Web programming with its component-based programming paradigm, its rich set of Web controls, its powerful database support, its flexible error handling and logging feature, and many others.

1.2.4 Is PRADO Stable Enough?

Yes. PRADO was initially released in August 2004. Many test suites have been written and conducted frequently to ensure its quality. It has been used by thousands of developers and many Web applications have been developed based on it. Bugs and feature requests are managed through TRAC system and we have a great user community and development team to ensure all questions are answered in a timely fashion.

1.2.5 History of PRADO

The very original inspiration of PRADO came from Apache Tapestry. During the design and implementation, I borrowed many ideas from Borland Delphi and Microsoft ASP.NET. The first version of PRADO came out in June 2004 and was written in PHP 4. Driven by the Zend PHP 5 coding contest, I rewrote PRADO in PHP 5, which proved to be a wise move, thanks to the new object model provided by PHP 5. PRADO won the grand prize in the Zend contest, earning the highest votes from both the public and the judges' panel.

In August 2004, PRADO started to be hosted on SourceForge as an open source project. Soon after, the project site xisc.com was announced to public. With the fantastic support of PRADO developer team and PRADO users, PRADO evolved to version 2.0 in mid 2005. In this version, Wei Zhuo contributed to PRADO with the excellent I18N and L10N support.

In May 2005, we decided to completely rewrite the PRADO framework to resolve a few fundamental issues found in version 2.0 and to catch up with some cool features available in Microsoft ASP.NET 2.0. After nearly a year's hard work with over 50,000 lines of new code, version 3.0 was finally made available in April 2006.

Starting from version 3.0, significant efforts are allocated to ensure the quality and stability of PRADO. If we say PRADO v2.x and v1.x are proof-of-concept work, we can say PRADO 3.x has grown up to a project that is suitable for serious business application development.

1.3 Installing PRADO

¡div id="install-info" class="block-content"¿

If you are viewing this page from your own Web server, you are already done with the installation.

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.

¡/div¿

¡div id="install-steps" class="block-content"¿

Installation of PRADO mainly involves downloading and unpacking.

1. Go to pradosoft.com to grab the latest version of PRADO.
2. Unpack the PRADO release file to a Web-accessible directory.

¡/div¿ ¡div id="install-after" class="block-content"¿

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/prado/demos/`. Here we assume PRADO is unpacked to the `prado` subdirectory under the `DocumentRoot` of the Web server.

If you encounter any problems with the demo applications, please use the PRADO requirement checker script, accessible via `http://web-server-address/prado/requirements/index.php`, to check first if your server configuration fulfills the conditions required by PRADO.

¡/div¿

1.4 New Features

This page summarizes the main new features that are introduced in each PRADO release.

1.4.1 Version 3.1.3

- Added Drag and drop controls
- Added TActiveDatePicker control

1.4.2 Version 3.1.2

- Added a new active control TActivePager that allows to paginate a databound control with an ajax callback.
- Added TFirebugLogRoute to send logs to the Firebug console

1.4.3 Version 3.1.1

- Added a new control TTabPanel that displays tabbed views.
- Added a new control TKeyboard that displays a virtual keyboard for text input.
- Added a new control TCaptcha that displays a CAPTCHA to keep spammers from signing up for certain accounts online. A related validator `TCaptchaValidator` is also implemented.
- Added a new control TSlider that displays a slider which can be used for numeric input.
- Added a new control TConditional that conditionally displays one of the two kinds of content.
- Added Oracle DB support to Active Record.
- Added support to TDataGrid to allow grouping consecutive cells with the same content.
- Added support to allow configuring page properties and authorization rules using relative page paths in application and page configurations. Added support to allow authorization based on remote host address.
- Added a new page state persister `TCachePageStatePersister`. It allows page state to be stored using a cache module (e.g. TMemCache, TDbCache, etc.)
- Added support to the auth framework to allow remembering login.
- Added support to display a prompt item in TDropDownList and TListBox (something like 'Please select:' as the first list item.)
- Added support to column mapping in Active Record.

1.4.4 Version 3.1.0

- Added seamless AJAX support. A whole array of AJAX-enabled controls, called active controls, are introduced. The usage of these active controls is very similar to their non-AJAX counterparts, i.e., plug in and use. For more details, see the tutorial about active controls.
- Added complete database support.
- Added new controls, modules and services, including TSoapService, TOutputCache, TSessionPageStatePersister, TFeedService, TJsonService, cache dependency classes, TXmlTransform.
- Enhanced some data controls with renderers. Renderer enables reusing item templates that are commonly found in `TRepeater`, `TDataList` and `TDataGrid`, and makes the configuration on these controls much cleaner. For more details about renders, see the updated tutorials on `TRepeater`, `TDataList` and `TDataGrid`.
- Added support to allow including external application configurations. Enhanced template syntax to facilitate subproperty configuration.
- Added TDbUserManager and TDbUser to simplify authentication and authorization with user accounts stored in a database.

1.5 Upgrading from v2.x and v1.x

¡div id=”from-2-or-1” class=”block-content”¿

PRADO v3.0 is NOT backward compatible with earlier versions of PRADO.

A good news is, properties and events of most controls remain intact, and the syntax of control templates remains largely unchanged. Therefore, developers' knowledge of earlier versions of PRADO are still applicable in v3.0.

We summarize in the following the most significant changes in v3.0 to help developers upgrade their v2.x and v1.x PRADO applications more easily, if needed.

¡/div¿

1.5.1 Component Definition

Version 3.0 has completely discarded the need of component specification files. It relies more on conventions for defining component properties and events. In particular, a property is defined by the existence of a getter method and/or a setter method, while an event is defined by the existence of an `on`-method. Property and event names in v3.0 are both case-insensitive. As a consequence, developers are now required to take care of type conversions when a component property is being set. For example, the following code is used to define the setter method for the `Enabled` property of `TControl`, which is of `boolean` type,

```
public function setEnabled($value)
{
     $value=TPropertyValue::ensureBoolean($value);
     $this->setViewState('Enabled',$value,true);
}
```

where `TPropertyValue::ensureBoolean()` is used to ensure that the input value be a boolean. This is because when the property is configured in template, a string value is passed to the setter. In previous versions, PRADO knows the property type based on the component specification files and does the type conversion for you.

1.5.2 Application Controller

Application controller now implements a modular architecture. Modules can be plugged in and configured in application specifications. Each module assumes a particular functionality, and they are coordinated together by the application lifecycle. The concept of v2.x modules is replaced in v3.0 by page directories. As a result, the format of v3.0 application specification is also different from earlier versions.

1.5.3 Pages

Pages in v3.0 are organized in directories which may be compared to the module concept in v2.x. Pages are requested using the path to them. For example, a URL `index.php?page=Controls.Samples.Sample1` would be used to request for a page named `Sample1` stored under the `[BasePath]/Controls/Samples` directory, where `[BasePath]` refers to the root page path. The file name of a page template must be

ended with `.page`, mainly to differentiate page templates from non-page control templates whose file names must be ended with `.tpl`.

1.5.4 Control Relationship

Version 3.0 redefines the relationships between controls. In particular, the parent-child relationship now refers to the enclosure relationship between controls' presentation. And a new naming-container relationship is introduced to help better manage control IDs. For more details, see the controls section.

1.5.5 Template Syntax

¡div id=”template-syntax” class=”block-content”¿

The syntax of control templates in v3.0 remains similar to those in earlier versions, with many enhancements. A major change is about the databinding expression. In v3.0, this is done by the following,

```
<com:MyComponent PropertyName=<%# PHP expression %> .../>
```

Expression and statement tags are also changed similarly. For more details, see the template definition section.

¡/div¿

1.5.6 Theme Syntax

Themes in v3.0 are defined like control templates with a few restrictions.

Chapter 2

Tutorials

2.1 My First PRADO Application

¡div id=”hello1” class=”block-content”¿

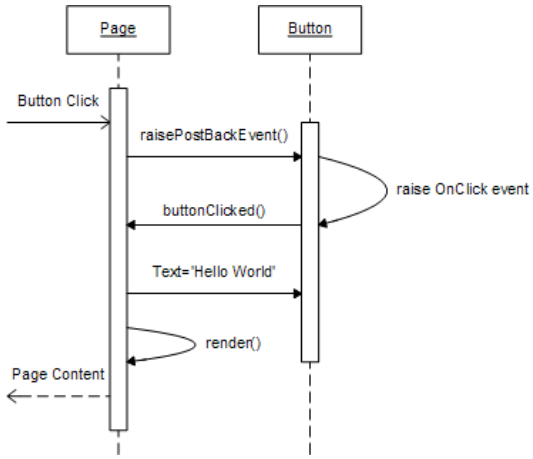
In this section, we guide you through creating your first PRADO application, the famous “Hello World” application.

“Hello World” perhaps is the simplest ¡i¿interactive¡/i¿ PRADO application that you can create. It displays to end-users a page with a submit button whose caption is `Click Me`. After the user clicks on the button, its caption is changed to `Hello World`.

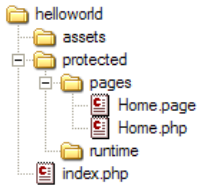
There are many approaches that can achieve the above goal. One can submit the page to the server, examine the POST variable, and generate a new page with the button caption updated. Or one can simply use JavaScript to update the button caption upon its `onclick` client event.

¡/div¿

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 `OnClick` event. To respond to the user clicking on the button, one simply needs to attach a function to the button’s `OnClick` event. Within the function, the button’s `Text` property is modified as “Hello World”. The following diagram shows the above sequence,



Our PRADO application consists of three files, `index.php`, `Home.page` and `Home.php`, which are organized as follows,



where each directory is explained as follows. Note, the above directory structure can be customized. For example, one can move the `protected` directory out of Web directories. You will know how to do this after you go through this tutorial.

- `assets` - directory storing published private files. See assets section for more details. This directory must be writable by the Web server process.
- `protected` - application base path storing application data and private script files. This directory should be configured as inaccessible to end-users.
- `runtime` - application runtime storage path storing application runtime information, such as application state, cached data, etc. This directory must be writable by the Web server process.
- `pages` - base path storing all PRADO pages.

> **Tip:**You may also use the `framework/prado-cli.php` command line script to create the Prado project directory structure. For example, type the command `php path/to/prado-cli.php -c helloworld` in the directory where you want to create the `helloworld` project.

The three files that we need are explained as follows.

- `index.php` - entry script of the PRADO application. This file is required by all PRADO applications and is the only script file that is directly accessible by end-users. Content in `index.php` mainly consists of the following three lines,

```
require_once('path/to/prado.php');  // include the prado script
$application=new TApplication;      // create a PRADO application instance
$application->run();                // run the application
```

- `Home.page` - template for the default page returned when users do not explicitly specify the page requested. A template specifies the presentational layout of components. In this example, we use two components, `TForm` and `TButton`, which correspond to the ¡form¿ and ¡input¿ HTML tags, respectively. The template contains the following content,

```
<html>
  <body>
    <com:TForm>
      <com:TButton Text="Click me" OnClick="buttonClicked" />
    </com:TForm>
  </body>
</html>
```

- `Home.php` - page class for the `Home` page. It mainly contains the method responding to the `OnClick` event of the button.

```
class Home extends TPage
{
    public function buttonClicked($sender,$param)
    {
        // $sender refers to the button component
        $sender->Text="Hello World!";
    }
}
```

¡div id=”hello-end” class=”block-content”¿

The application is now ready and can be accessed via: `http://Web-server-address/helloworld/index.php`, assuming `helloworld` is directly under the Web `DocumentRoot`. Try to change `TButton` in `Home.page` to `TLinkButton` and see what happens.

Complete source code of this demo can be found in the PRADO release. You can also try the online demo.

¡/div¿

2.2 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, ¡br /¿¡br /¿ To be continued...

Fundamentals.Samples.Hangman.Home Demo

2.3 Command Line Tool

The optional `prado-cli.php` PHP script file in the `framework` directory provides command line tools to perform various tedious takes in Prado. The `prado-cli.php` can be used to create Prado project skeletons, create initial test fixtures, and access to an interactive PHP shell.

2.3.1 Requirements

To use the command line tool, you need to use your command prompt, command console or terminal. In addition, PHP must be able to execute PHP scripts from the command line.

2.3.2 Usage

If you type `php path/to/framework/prado-cli.php`, you should see the following information. Alternatively, if you are not on Windows, you may try to change the `prado-cli.php` into an executable and execute it as a script

```
Command line tools for Prado 3.0.5.
usage: php prado-cli.php action <parameter> [optional]
example: php prado-cli.php -c mysite

actions:
  -c <directory>
    Creates a Prado project skeleton for the given <directory>.

  -t <directory>
    Create test fixtures in the given <directory>.

  shell [directory]
    Runs a PHP interactive interpreter. Initializes the Prado
    application in the given [directory].
```

The **¡parameter¿** are required parameters and [**optional**] are optional parameters.

2.3.3 Creating a new Prado project skeleton

To create a Prado project skeleton, do the following:

1. Change to the directory where you want to create the project skeleton.

2. Type, `php ../prado/framework/prado-cli.php -c helloworld`, where `helloworld` is the directory name that you want to create the project skeleton files.

3. Type, `php ../prado/framework/prado-cli.php -t helloworld` to create the test fixtures for the `helloworld` project.

2.3.4 Interactive Shell

The interactive shell allows you to evaluate PHP statements from the command line. The `prado-cli.php` script can be used to start the shell and load an existing Prado project. For example, let us load the blog demo project. Assume that your command line is in the `prado` distribution directory and you type.

```
$: php framework/prado-cli.php shell demos/blog
```

The output should be

```
Command line tools for Prado 3.0.5.
** Loaded Prado application in directory "demos\blog\protected".
PHP-Shell - Version 0.3.1
(c) 2006, Jan Kneschke <jan@kneschke.de>

>> use '?' to open the inline help

>>
```

Then we will get an instance of the Prado blog application, and from that instance we want an instance of the `'data'` module. Notice that a **semicolon** at the end of the line **suppresses the output**.

```
>> $app = Prado::getApplication();

>> $db = $app->getModule('data');
```

Lastly, we want to use the data module to query for a post with `ID=1`. Notice that we **leave out the semicolon** to show the results.

```
>> $db->queryPostByID(1)
```

There should not be any errors and you should see the following.

```
PostRecord#1
(
```

```
    [ID] => 1
    [AuthorID] => 1
    [AuthorName] => 'Prado User'
    [CreateTime] => 1148819691
    [ModifyTime] => 0
    [Title] => 'Welcome to Prado Weblog'
    [Content] => 'Congratulations! You have successfully installed Prado Blog --
 a PRADO-driven weblog system. A default administrator account has been created.
 Please login with \textbf{admin/prado} and update your password as soon as possible.'
    [Status] => 0
    [CommentCount] => 0
)
```

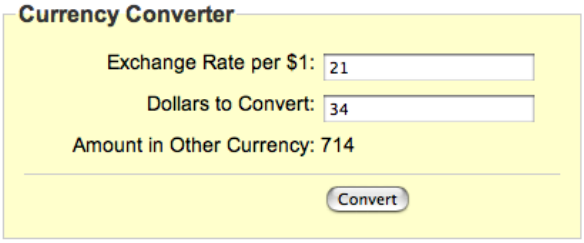
Chapter 3

Tutorial: Currency Converter

3.1 Building a Simple Currency Converter

This tutorial introduces the Prado web application framework and teaches you how to build a simple web application in a few simple steps. This tutorial assumes that you are familiar with PHP and you have access to a web server that is able to serve PHP5 scripts.

In this tutorial you will build a simple web application that converts a dollar amount to an other currency, given the rate of that currency relative to the dollar. The completed application is shown bellow.



You can try the application locally or at Pradosoft.com. Notice that the application still functions exactly the same if javascript is not available on the user's browser.

3.2 Downloading and Installing Prado

To install Prado, simply download the latest version of Prado from http://www.pradosoft.com and unzip the file to a directory **not** accessible by your web server (you may unzip it to a directory accessible by the web server if you wish to see the demos and test). For further detailed installation, see the Quickstart Installation guide.

3.3 Creating a new Prado web Application

The quickest and simplest way to create a new Prado web application is to use the command tool `prado-cli.php` found in the `framework` directory of the Prado distribution. We create a new application by running the following command in your command prompt or console. The command creates a new directory named `currency-converter` in your current working directory. You may need to change to the appropriate directory first. See the Command Line Tool for more details.

```
php prado/framework/prado-cli.php -c currency-converter
```

The above command creates the necessary directory structure and minimal files (including "index.php" and "Home.page") to run a Prado web application. Now you can point your browser's url to the web server to serve up the `index.php` script in the `currency-converter` directory. You should see the message "Welcome to Prado!"

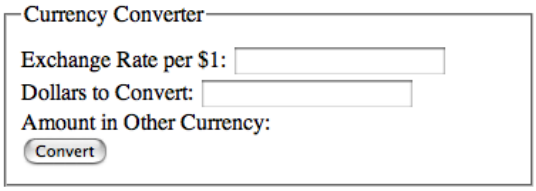
3.4 Creating the Currency Converter User Interface

We start by editing the `Home.page` file found in the `currency-converter/protected/pages/` directory. Files ending with ".page" are page templates that contains HTML and Prado controls. We simply add two textboxes, three labels and one button as follows.

```
<com:TForm>
    <fieldset>
        <legend>Currency Converter</legend>
        <div class="rate-field">
            <com:TLabel ForControl="currencyRate" Text="Exchange Rate per $1:" />
            <com:TTextBox ID="currencyRate" />
```

```
        </div>
        <div class="dollar-field">
            <com:TLabel ForControl="dollars" Text="Dollars to Convert:" />
            <com:TTextBox ID="dollars" />
        </div>
        <div class="total-field">
            <span class="total-label">Amount in Other Currency:</span>
            <com:TLabel ID="total" CssClass="result" />
        </div>
        <div class="convert-button">
            <com:TButton Text="Convert" />
        </div>
    </fieldset>
</com:TForm>
```

If you refresh the page, you should see something similar to the following figure. It may not look very pretty or orderly, but we shall change that later using CSS.



The first component we add is a TForm that basically corresponds to the HTML `<form>` element. In Prado, only **one** `TForm` element is allowed per page.

The next two pair of component we add is the TLabel and TTextBox that basically defines a label and a textbox for the user of the application to enter the currency exchange rate. The `ForControl` property value determines which component that the label is for. This allows the user of the application to click on the label to focus on the field (a good thing). You could have used a plain HTML `<label>` element to do the same thing, but you would have to find the correct `ID` of the textbox (or `<input>` in HTML) as Prado components may/will render the `ID` value differently in the HTML output.

The next pair of components are similar and defines the textbox to hold the dollar value to be converted. The `TLabel` with `ID` value "total" defines a simple label. Notice that the `ForControl` property is absent. This means that this label is simply a simple label which we are going to use to display the converted total amount.

The final component is a TButton that the user will click to calculate the results. The `Text` property sets the button label.

3.5 Implementing Currency Conversion

If you tried clicking on the "Convert" button then the page will refresh and does not do anything else. For the button to do some work, we need to add a "Home.php" to where "Home.page" is. The `Home` class should extends the TPage, the default base class for all Prado pages.

```
<?php
class Home extends TPage
{

}
?>
```

Prado uses PHP's `__autoload` method to load classes. The convention is to use the class name with ".php" extension as filename.

So far there is nothing interesting about Prado, we just declared some "web components" in some template file named Home.page and created a "Home.php" file with a `Home` class. The more interesting bits are in Prado's event-driven architecture as we shall see next.

We want that when the user click on the "Convert" button, we take the values in the textbox, do some calculation and present the user with the converted total. To handle the user clicking of the "Convert" button we simply add an `OnClick` property to the "Convert" button in the "Home.page" template and add a corresponding event handler method in the "Home.php".

```
<com:TButton Text="Convert" OnClick="convert_clicked" />
```

The value of the `OnClick`, "`convert_clicked`", will be the method name in the "Home.php" that will called when the user clicks on the "Convert" button.

```
class Home extends TPage
{
    public function convert_clicked($sender, $param)
```

```
    {
        $rate = floatval($this->currencyRate->Text);
        $dollars = floatval($this->dollars->Text);
        $this->total->Text = $rate * $dollars;
    }
}
```

¡div id=”3332” class=”block-content”¿

If you run the application in your web browser, enter some values and click the “Convert” button then you should see that calculated value displayed next to the “Amount in Other Currency” label.

In the “`convert_clicked`” method the first parameter, `$sender`, corresponds to the object that raised the event, in this case, the “Convert” button. The second parameter, `$param` contains any additional data that the `$sender` object may wish to have added.

We shall now examine, the three lines that implements the simply currency conversion in the “`convert_clicked`” method.

¡/div¿

```
$rate = floatval($this->currencyRate->Text);
```

The statement `$this->currencyRate` corresponds to the `TTextBox` component with `ID` value “currencyRate” in the “Home.page” template. The `Text` property of the `TTextBox` contains the value that the user entered. So, we obtain this value by `$this->currencyRate->Text` which we convert the value to a float value.

```
$dollars = floatval($this->dollars->Text);
```

¡div id=”5551” class=”block-content”¿

The next line does a similar things, it takes the user value from the `TTextBox` with `ID` value “dollars and converts it to a float value.

The third line calculates the new amount and set this value in the `Text` property of the `TLabel` with `ID="total"`. Thus, we display the new amount to the user in the label.

¡/div¿

```
$this->total->Text = $rate * $dollars;
```

3.6 Adding Validation

The way we convert the user entered value to float ensures that the total amount is always a number. So the user is free to enter what ever they like, they could even enter letters. The user's experience in using the application can be improved by adding validators to inform the user of the allowed values in the currency rate and the amount to be calcuated.

For the currency rate, we should ensure that

1. the user enters a value,
2. the currency rate is a valid number,
3. the currency rate is positive.

To ensure 1 we add one TRequiredFieldValidator. To ensure 2 and 3, we add one TCompareValidator. We may add these validators any where within the "Home.page" template. Further details regarding these validator and other validators can be found in the Validation Controls page.

```
<com:TRequiredFieldValidator
     ControlToValidate="currencyRate"
     ErrorMessage="Please enter a currency rate." />
<com:TCompareValidator
     ControlToValidate="currencyRate"
     DataType="Float"
     ValueToCompare="0"
     Operator="GreaterThan"
     ErrorMessage="Please enter a positive currency rate." />
```

For the amount to be calculated, we should ensure that

1. the user enters a value,
2. the value is a valid number (not including any currency or dollar signs).

To ensure 1 we just add another `TRequiredFieldValidator`, for 2 we could use a TDataTypeValidator. For simplicity we only allow the user to enter a number for the amount they wish to convert.

```
<com:TRequiredFieldValidator
     ControlToValidate="dollars"
     ErrorMessage="Please enter the amount you wish to calculate." />
<com:TDataTypeValidator
     ControlToValidate="dollars"
     DataType="Float"
     ErrorMessage="Please enter a number." />
```

Now if you try to enter some invalid data in the application or left out any of the fields the validators will be activated and present the user with error messages. Notice that the error messages are presented without reloading the page. Prado's validators by default validates using both javascript and server side. The server side validation is **always performed**. For the server side, we should skip the calculation if the validators are not satisfied. This can done as follows.

```
public function convert_clicked($sender, $param)
{
    if($this->Page->IsValid)
    {
        $rate = floatval($this->currencyRate->Text);
        $dollars = floatval($this->dollars->Text);
        $this->total->Text = $rate * $dollars;
    }
}
```

3.7 Improve User Experience With Active Controls

In this simple application we may further improve the user experience by increasing the responsiveness of the application. One way to achieve a faster response is calculate and present the results without reloading the whole page.

We can replace the `TButton` with the Active Control counter part, TActiveButton, that can trigger a server side click event without reloading the page. In addition, we can change the "totals" `TLabel` with the Active Control counter part, TActiveLabel, such that the server side can update the browser without reloading the page.

```
<div class="total-field">
    <span class="total-label">Amount in Other Currency:</span>
```

```
        <com:TActiveLabel ID="total" CssClass="result" />
    </div>
    <div class="convert-button">
        <com:TActiveButton Text="Convert" OnClick="convert_clicked" />
</div>
```

The server side logic remains the same, we just need to import the Active Controls name space as they are not included by default. We add the following line to the begin of "Home.php".

```
Prado::using('System.Web.UI.ActiveControls.*');
```

If you try the application now, you may notice that the page no longer needs to reload to calculate and display the converted total amount. However, since there is not page reload, there is no indication or not obvious that by clicking on the "Convert" button any has happened. We can further refine the user experience by change the text of "total" label to "calculating..." when the user clicks on the "Convert" button. The text of the "total" label will still be updated with the new calculate amount as before.

To indicate that the calculation is in progress, we can change the text of the "total" label as follows. We add a `ClientSide.OnLoading` property to the "Convert" button (since this button is responsible for requesting the calculation).

```
<com:TActiveButton Text="Convert" OnClick="convert_clicked" >
    <prop:ClientSide.OnLoading>
        $('<%= $this->total->ClientID %>').innerHTML = "calculating..."
    </prop:ClientSide.OnLoading>
</com:TActiveButton>
```

The `ClientSide.OnLoading` and various other properties accept a javascript block as their content or value. The javascript code `$('...')` is a javascript function that is equivalent to `document.getElementById('...')` that takes a string with the ID of an HTML element. Since Prado renders its components's IDs, we need to use the rendered ID of the "total" label, that is, `$this->total->ClientID`. We place this bit of code within a `<%= %>` to obtain the rendered HTML ID for the "total" label. The rest of the javascript code `innerHTML = ''calculating..."` simply changes the content of the "total" label.

3.8 Adding Final Touches

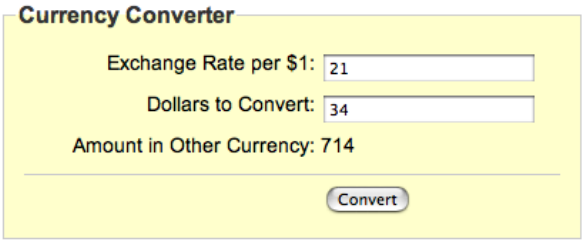
So far we have built a simple currency converter web application with little attention of the looks and feel. Now we can add a stylesheet to improve the overall appearance of the application. We can simply add the stylesheet inline with the template code or we may create a "theme".

To create and use a theme with Prado applications, we simply create a new directory "themes/Basic" in the `currency-converter` directory. You may need to create the `themes` directory first. Any directory within the `themes` are considered as a theme with the name of the theme being the directory name. See the Themes and Skins for further details.

We simply create a CSS file named "common.css" and save it in the `themes/Basic` directory. Then we add the following code to the beginning of "Home.page" (we add a little more HTML as well).

```
<%@ Theme="Basic" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" >
<com:THead Title="Currency Converter" />
<body>
```

The first line `<%@ Theme="Basic" %>` defines the theme to be used for this page. The THead corresponds to the HTML `<head>` element. In addition to display the `Title` property by the `THead`, all CSS files in the `themes/Basic` directory are also rendered/linked for the current page. Our final currency converter web application looks like the following.



This completes introduction tutorial to the Prado web application framework.

Chapter 4

Tutorial: Building an AJAX Chat Application

4.1 Building an AJAX Chat Application

This tutorial introduces the Prado web application framework's ActiveRecord and Active Controls to build a Chat web application. It is assumed that you are familiar with PHP and you have access to a web server that is able to serve PHP5 scripts. This basic chat application will utilize the following ideas/components in Prado.

- Building a custom User Manager class.
- Authenticating and adding a new user to the database.
- Using ActiveRecord to interact with the database.
- Using Active Controls and callbacks to implement the user interface.
- Separating application logic and application flow.

In this tutorial you will build an AJAX Chat web application that allows multiple users to communicate through their web browser. The application consists of two pages: a login page that asks the user to enter their nickname and the main application chat page. You can try the application locally or at Pradosoft.com. The main application chat page is shown bellow.

4.2 Download, Install and Create a New Application

The download and installation steps are similar to those in the Currency converter tutorial. To create the application, we run from the command line the following. See the Command Line Tool for more details.

```
php prado/framework/prado-cli.php -c chat
```

The above command creates the necessary directory structure and minimal files (including "index.php" and "Home.page") to run a Prado web application. Now you can point your browser's URL to the web server to serve up the `index.php` script in the `chat` directory. You should see the message "Welcome to Prado!"

4.3 Authentication and Authorization

The first task for this application is to ensure that each user of the chat application is assigned with a unique (chosen by the user) username. To achieve this, we can secure the main chat application page to deny access to anonymous users. First, let us create the `Login` page with the following code. We save the `Login.php` and `Login.page` in the `chat/protected/pages/` directory (there should be a `Home.page` file created by the command line tool).

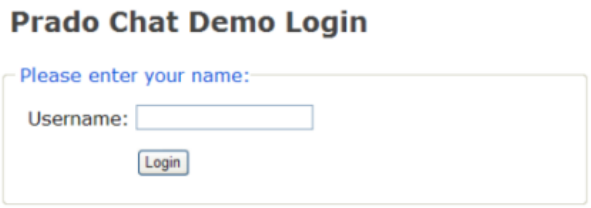
```
<?php
class Login extends TPage
```

```
{
}
?>
```

```
<!doctype html public "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <title>Prado Chat Demo Login</title>
</head>
<body>
<com:TForm>
    <h1 class="login">Prado Chat Demo Login</h1>
    <fieldset class="login">
        <legend>Please enter your name:</legend>
        <div class="username">
            <com:TLabel ForControl="username" Text="Username:" />
            <com:TTextBox ID="username" MaxLength="20" />
            <com:TRequiredFieldValidator
                ControlToValidate="username"
                Display="Dynamic"
                ErrorMessage="Please provide a username." />
        </div>
        <div class="login-button">
            <com:TButton Text="Login" />
        </div>
</com:TForm>
</body>
</html>
```

The login page contains a TForm, a TTextBox, a TRequiredFieldValidator and a TButton. The resulting page looks like the following (after applying some a style sheet).

If you click on the `Login` button without entering any text in the username textbox, an error message is displayed. This is due to the TRequiredFieldValidator requiring the user to enter some text in the textbox before proceeding.



4.3.1 Securing the `Home` page

Now we wish that if the user is trying to access the main application page, `Home.page`, before they have logged in, the user is presented with the `Login.page` first. We add a `chat/protected/application.xml` configuration file to import some classes that we shall use later.

```
<?xml version="1.0" encoding="utf-8"?>
<application id="Chat" Mode="Debug">
  <paths>
    <using namespace="System.Data.*" />
    <using namespace="System.Data.ActiveRecord.*" />
    <using namespace="System.Security.*" />
    <using namespace="System.Web.UI.ActiveControls.*" />
  </paths>
</application>
```

Next, we add a `chat/protected/pages/config.xml` configuration file to secure the `pages` directory.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <modules>
    <module id="users" class="TUserManager" />
    <module id="auth" class="TAuthManager" UserManager="users" LoginPage="Login" />
  </modules>
  <authorization>
    <allow pages="Login" users="?" />
    <allow roles="normal" />
    <deny users="*" />
  </authorization>
</configuration>
```

We setup the authentication using the default classes as explained in the authentication/authorization quickstart. In the authorization definition, we allow anonymous users to access the `Login` page (anonymous users is specified by the `?` question mark). We allow any users with role equal to "normal" (to be defined later) to access all the pages, that is, the `Login` and `Home` pages. Lastly, we deny all users without any roles to access any page. The authorization rules are executed on first match basis.

If you now try to access the `Home` page by pointing your browser to the `index.php` you will be redirected to the `Login` page.

4.4 Active Record for `chat_users` table

The TUserManager class only provides a read-only list of users. We need to be able to add or login new users dynamically. So we need to create our own user manager class. First, we shall setup a database with a `chat_users` table and create an ActiveRecord that can work with the `chat_users` table with ease. For the demo, we use `sqlite` as our database for ease of distributing the demo. The demo can be extended to use other databases such as MySQL or Postgres SQL easily. We define the `chat_users` table as follows.

```
CREATE TABLE chat_users
(
     username VARCHAR(20) NOT NULL PRIMARY KEY,
     last_activity INTEGER NOT NULL DEFAULT "0"
);
```

Next we define the corresponding `ChatUserRecord` class and save it as `chat/protected/App_Code/ChatUserRecord.ph` (you need to create the `App_Code` directory as well). We also save the sqlite database file as `App_Code/chat.db`.

```
class ChatUserRecord extends TActiveRecord
{
    const TABLE='chat_users';

    public $username;
    public $last_activity;

    public static function finder($className=__CLASS__)
    {
```

```
        return parent::finder($className);
    }
}
```

Before using the `ChatUserRecord` class we to configure a default database connection for ActiveRecord to function. In the `chat/protected/application.xml` we import classes from the `App_Code` directory and add an ActiveRecord configuration module.

```
<?xml version="1.0" encoding="utf-8"?>
<application id="Chat" Mode="Debug">
  <paths>
    <using namespace="Application.App_Code.*" />
    <using namespace="System.Data.*" />
    <using namespace="System.Data.ActiveRecord.*" />
    <using namespace="System.Security.*" />
    <using namespace="System.Web.UI.ActiveControls.*" />
  </paths>
  <modules>
    <module class="TActiveRecordConfig" EnableCache="true"
        Database.ConnectionString="sqlite:protected/App_Code/chat.db" />
  </modules>
</application>
```

4.4.1 Custom User Manager class

To implement a custom user manager module class we just need to extends the `TModule` class and implement the `IUserManager` interface. The `getGuestName()`, `getUser()` and `validateUser()` methods are required by the `IUserManager` interface. We save the custom user manager class as `App_Code/ChatUserManager.php`.

```
class ChatUserManager extends TModule implements IUserManager
{
    public function getGuestName()
    {
        return 'Guest';
    }

    public function getUser($username=null)
```

```
    {
        $user=new TUser($this);
        $user->setIsGuest(true);
        if($username !== null && $this->usernameExists($username))
        {
            $user->setIsGuest(false);
            $user->setName($username);
            $user->setRoles(array('normal'));
        }
        return $user;
    }

    public function addNewUser($username)
    {
        $user = new ChatUserRecord();
        $user->username = $username;
        $user->save();
    }

    public function usernameExists($username)
    {
        $finder = ChatUserRecord::finder();
        $record = $finder->findByUsername($username);
        return $record instanceof ChatUserRecord;
    }

    public function validateUser($username,$password)
    {
        return $this->usernameExists($username);
    }
}
```

The `getGuestName()` method simply returns the name for a guest user and is not used in our application. The `getUser()` method returns a `TUser` object if the username exists in the database, the `TUser` object is set with role of "normal" that corresponds to the `<authorization>` rules defined in our `config.xml` file.

The `addNewUser()` and `usernameExists()` method uses the ActiveRecord corresponding to the `chat_users` table to add a new user and to check if a username already exists, respectively.

The next thing to do is change the `config.xml` configuration to use our new custom user manager class. We simply change the `<module>` configuration with `id="users"`.

```
<module id="users" class="ChatUserManager" />
```

4.5 Authentication

To perform authentication, we just want the user to enter a unique username. We add a TCustomValidator for validate the uniqueness of the username and add an `OnClick` event handler for the login button.

```
<com:TCustomValidator
    ControlToValidate="username"
    Display="Dynamic"
    OnServerValidate="checkUsername"
    ErrorMessage="The username is already taken." />

...

<com:TButton Text="Login" OnClick="createNewUser" />
```

In the `Login.php` file, we add the following 2 methods.

```
function checkUsername($sender, $param)
{
    $manager = $this->Application->Modules['users'];
    if($manager->usernameExists($this->username->Text))
        $param->IsValid = false;
}

function createNewUser($sender, $param)
{
    if($this->Page->IsValid)
    {
        $manager = $this->Application->Modules['users'];
        $manager->addNewUser($this->username->Text);

        //do manual login
```

```
        $user = $manager->getUser($this->username->Text);
        $auth = $this->Application->Modules['auth'];
        $auth->updateSessionUser($user);
        $this->Application->User = $user;

        $url = $this->Service->constructUrl($this->Service->DefaultPage);
        $this->Response->redirect($url);
    }
}
```

The `checkUserName()` method uses the `ChatUserManager` class (recall that in the `config.xml` configuration we set the ID of the custom user manager class as "users") to validate the username is not taken.

In the `createNewUser` method, when the validation passes (that is, when the user name is not taken) we add a new user. Afterward we perform a manual login process:

- First we obtain a `TUser` instance from our custom user manager class using the `$manager->getUser(...)` method.
- Using the `TAuthManager` we set/update the user object in the current session data.
- Then we set/update the `Application`'s user instance with our new user object.

Finally, we redirect the client to the default `Home` page.

4.5.1 Default Values for ActiveRecord

If you try to perform a login now, you will receive an error message like "¡i¿Property '`ChatUserRecord::$last_activity` must not be null as defined by column '`last_activity`' in table '`chat_users`'.¡/i¿". This means that the `$last_activity` property value was null when we tried to insert a new record. We need to either define a default value in the corresponding column in the table and allow null values or set the default value in the `ChatUserRecord` class. We shall demonstrate the later by altering the `ChatUserRecord` with the addition of a set getter/setter methods for the `last_activity` property.

```
private $_last_activity;

public function getLast_Activity()
```

```
{
    if($this->_last_activity === null)
        $this->_last_activity = time();
    return $this->_last_activity;
}

public function setLast_Activity($value)
{
    $this->_last_activity = $value;
}
```

Notice that we renamed `$last_activity` to `$_last_activity` (note the underscore after the dollar sign).

4.6 Main Chat Application

Now we are ready to build the main chat application. We use a simple layout that consist of one panel holding the chat messages, one panel to hold the users list, a textarea for the user to enter the text message and a button to send the message.

```
<!doctype html public "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <title>Prado Chat Demo</title>
<style>
.messages
{
    width: 500px;
    height: 300px;
    float: left;
    border: 1px solid ButtonFace;
    overflow: auto;
}
.user-list
{
    margin-left: 2px;
    float: left;
```

```
    width: 180px;
    height: 300px;
    border: 1px solid ButtonFace;
    overflow: auto;
    font-size: 0.85em;
}
.message-input
{
    float: left;
}

.message-input textarea
{
    margin-top: 3px;
    padding: 0.4em 0.2em;
    width: 493px;
    font-family: Verdana, Geneva, Arial, Helvetica, sans-serif;
    font-size: 0.85em;
    height: 40px;
}
.send-button
{
    margin: 0.5em;
}
</style>
</head>
<body>
<com:TForm>
\section{Prado Chat Demo}
<div id="messages" class="messages">
    <com:TPlaceHolder ID="messageList" />
</div>
<div id="users" class="user-list">
    <com:TPlaceHolder ID="userList" />
</div>
<div class="message-input">
    <com:TActiveTextBox ID="userinput"
        Columns="40" Rows="2" TextMode="MultiLine" />
    <com:TActiveButton ID="sendButton" CssClass="send-button"
        Text="Send" />
```

```
</div>
</com:TForm>
<com:TJavascriptLogger />
</body>
</html>
```

We added two Active Control components: a TActiveTextBox and a TActiveButton. We also added a TJavascriptLogger that will be very useful for understanding how the Active Controls work.

4.6.1 Exploring the Active Controls

We should have some fun before we proceeding with setting up the chat buffering. We want to see how we can update the current page when we receive a message. First, we add an `OnClick` event handler for the `Send` button.

```
<com:TActiveButton ID="sendButton" CssClass="send-button"
     Text="Send" OnClick="processMessage"/>
```

And the corresponding event handler method in the `Home.php` class (we need to create this new file too).

```
class Home extends TPage
{
    function processMessage($sender, $param)
    {
        echo $this->userinput->Text;
    }
}
```

If you now type something in the main application textbox and click the send button you should see whatever you have typed echoed in the `TJavascriptLogger` console.

To append or add some content to the message list panel, we need to use some methods in the TCallbackClientScript class which is available through the `CallbackClient` property of the current `TPage` object. For example, we do can do

```
function processMessage($sender, $param)
```

```
{
    $this->CallbackClient->appendContent("messages", $this->userinput->Text);
}
```

This is one way to update some part of the existing page during a callback (AJAX style events) and will be the primary way we will use to implement the chat application.

4.7 Active Record for chat_buffer table

To send a message to all the connected users we need to buffer or store the message for each user. We can use the database to buffer the messages. The chat_buffer table is defined as follows.

```
CREATE TABLE chat_buffer
(
     id INTEGER PRIMARY KEY,
     for_user VARCHAR(20) NOT NULL,
     from_user VARCHAR(20) NOT NULL,
     message TEXT NOT NULL,
     created_on INTEGER NOT NULL DEFAULT "0"
);
```

The corresponding ChatBufferRecord class is saved as App_Code/ChatBufferRecord.php.

```
class ChatBufferRecord extends TActiveRecord
{
     const TABLE='chat_buffer';

    public $id;
    public $for_user;
    public $from_user;
    public $message;
    private $_created_on;

    public function getCreated_On()
    {
        if($this->_created_on === null)
            $this->_created_on = time();
```

```
        return $this->_created_on;
    }

    public function setCreated_On($value)
    {
        $this->_created_on = $value;
    }

    public static function finder($className=__CLASS__)
    {
        return parent::finder($className);
    }
}
```

4.8 Chat Application Logic

We finally arrive at the guts of the chat application logic. First, we need to save a received message into the chat buffer for **all** the current users. We add this logic in the `ChatBufferRecord` class.

```
public function saveMessage()
{
    foreach(ChatUserRecord::finder()->findAll() as $user)
    {
        $message = new self;
        $message->for_user = $user->username;
        $message->from_user = $this->from_user;
        $message->message = $this->message;
        $message->save();
        if($user->username == $this->from_user)
        {
            $user->last_activity = time(); //update the last activity;
            $user->save();
        }
    }
}
```

We first find all the current users using the `ChatUserRecord` finder methods. Then we duplicate the message and save it into the database. In addition, we update the message sender's last

activity timestamp. The above piece of code demonstrates the simplicity and succinctness of using ActiveRecords for simple database designs.

The next piece of the logic is to retrieve the users' messages from the buffer. We simply load all the messages for a particular username and format that message appropriately (remember to escape the output to prevent Cross-Site Scripting attacks). After we load the messages, we delete those loaded messages and any older messages that may have been left in the buffer.

```
public function getUserMessages($user)
{
    $content = '';
    foreach($this->findAll('for_user = ?', $user) as $message)
        $content .= $this->formatMessage($message);
    $this->deleteAll('for_user = ? OR created_on < ?',
                         $user, time() - 300); //5 min inactivity
    return $content;
}

protected function formatMessage($message)
{
    $user = htmlspecialchars($message->from_user);
    $content = htmlspecialchars($message->message);
    return "<div class=\"message\">\textbf{{$user}:}"
                  ." <span>{$content}</span></div>";
}
```

To retrieve a list of current users (formatted), we add this logic to the `ChatUserRecord` class. We delete any users that may have been inactive for awhile.

```
public function getUserList()
{
    $this->deleteAll('last_activity < ?', time()-300); //5 min inactivity
    $content = '\begin{itemize}';
    foreach($this->findAll() as $user)
        $content .= '\item '.htmlspecialchars($user->username).'';
    $content .= '\end{itemize}';
    return $content;
}
```

> **Note:** For simplicity we formatted the messages in these Active Record classes. For large applications, these message formatting tasks should be done using Prado components (e.g. using a TRepeater in the template or a custom component).

4.9 Putting It Together

Now comes to put the application flow together. In the `Home.php` we update the `Send` buttons `OnClick` event handler to use the application logic we just implemented.

```
function processMessage($sender, $param)
{
    if(strlen($this->userinput->Text) > 0)
    {
        $record = new ChatBufferRecord();
        $record->message = $this->userinput->Text;
        $record->from_user = $this->Application->User->Name;
        $record->saveMessage();

        $this->userinput->Text = '';
        $messages = $record->getUserMessages($this->Application->User->Name);
        $this->CallbackClient->appendContent("messages", $messages);
        $this->CallbackClient->focus($this->userinput);
    }
}
```

We simply save the message to the chat buffer and then ask for all the messages for the current user and update the client side message list using a callback response (AJAX style).

At this point the application is actually already functional, just not very user friendly. If you open two different browsers, you should be able to communicate between the two users whenever the `Send` button is clicked.

The next part is perhaps the more tricker and fiddly than the other tasks. We need to improve the user experience. First, we want a list of current users as well. So we add the following method to `Home.php`, we can call this method when ever some callback event is raised, e.g. when the `Send` button is clicked.

```
protected function refreshUserList()
```

```
{
    $lastUpdate = $this->getViewState('userList','');
    $users = ChatUserRecord::finder()->getUserList();
    if($lastUpdate != $users)
    {
        $this->CallbackClient->update('users', $users);
        $this->setViewstate('userList', $users);
    }
}
```

Actually, we want to periodically update the messages and user list as new users join in and new message may arrive from other users. So we need to refresh the message list as well.

```
function processMessage($sender, $param)
{
    ...
    $this->refreshUserList();
    $this->refreshMessageList();
    ...
}

protected function refreshMessageList()
{
    //refresh the message list
    $finder = ChatBufferRecord::finder();
    $content = $finder->getUserMessages($this->Application->User->Name);
    if(strlen($content) > 0)
    {
        $anchor = (string)time();
        $content .= "<a href=\"#\" id=\"{$anchor}\"> </a>";
        $this->CallbackClient->appendContent("messages", $content);
        $this->CallbackClient->focus($anchor);
    }
}
```

The anchor using `time()` as ID for a focus point is so that when the message list on the client side gets very long, the focus method will scroll the message list to the latest message (well, it works in most browsers).

Next, we need to redirect the user back to the login page if the user has been inactive for some time, say about 5 mins, we can add this check to any stage of the page life-cycle. Lets add it to the `onLoad()` stage.

```
public function onLoad($param)
{
    $username = $this->Application->User->Name;
    if(!$this->Application->Modules['users']->usernameExists($username))
    {
        $auth = $this->Application->Modules['auth'];
        $auth->logout();

        //redirect to login page.
        $this->Response->Redirect($this->Service->ConstructUrl($auth->LoginPage));
    }
}
```

4.10 Improving User Experience

The last few details are to periodically check for new messages and refresh the user list. We can accomplish this by polling the server using a TTimeTriggeredCallback control. We add a `TTimeTriggeredCallback` to the `Home.page` and call the `refresh` handler method defined in `Home.php`. We set the polling interval to 2 seconds.

```
<com:TTimeTriggeredCallback OnCallback="refresh"
     Interval="2" StartTimerOnLoad="true" />
```

```
function refresh($sender, $param)
{
    $this->refreshUserList();
    $this->refreshMessageList();
}
```

The final piece requires us to use some javascript. We want that when the user type some text in the textarea and press the `Enter` key, we want it to send the message without clicking on the `Send` button. We add to the `Home.page` some javascript.

```
<com:TClientScript>
Event.observe($("<%= $this->userinput->ClientID %>"), "keypress", function(ev)
{
    if(Event.keyCode(ev) == Event.KEY_RETURN)
    {
        if(Event.element(ev).value.length > 0)
            new Prado.Callback("<%= $this->sendButton->UniqueID %>");
        Event.stop(ev);
    }
});
</com:TClientScript>
```

Details regarding the javascript can be explored in the Introduction to Javascript section of the quickstart.

This completes the tutorial on making a basic chat web application using the Prado framework. Hope you have enjoyed it.

Chapter 5

Tutorial: Addressbook

5.1 A Simple Address Book

This tutorial introduces the basics of connecting to a database using ActiveRecord and using Active Record scaffolds to quickly build a simple address book.

Chapter 6

Fundamentals

6.1 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 some of the major classes provided by PRADO,

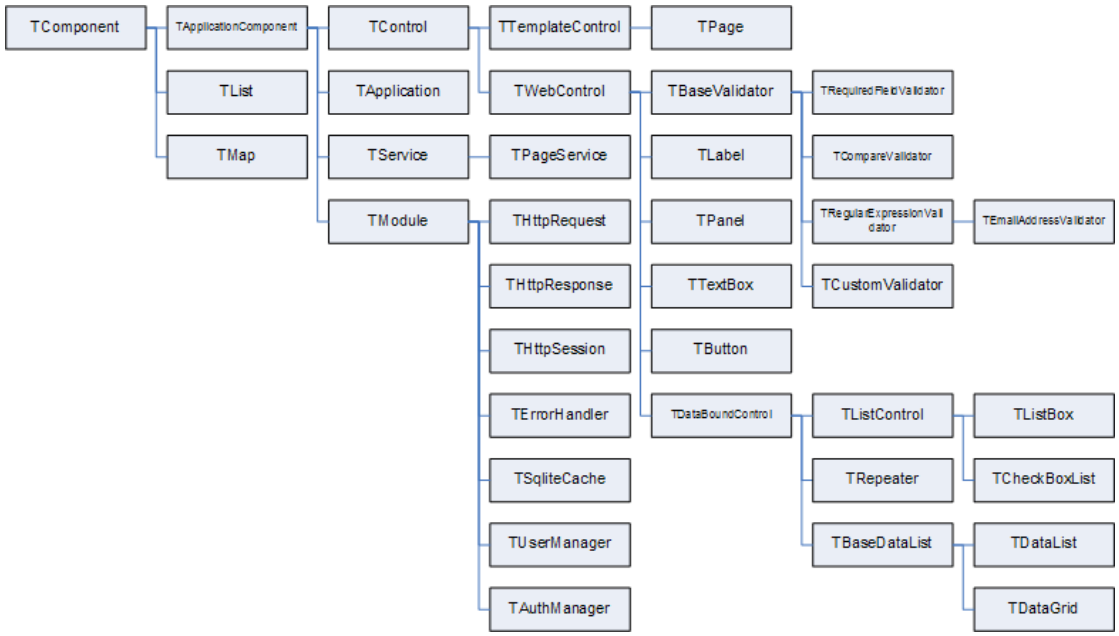
When a PRADO application is processing a page request, its static object diagram can be shown as follows,

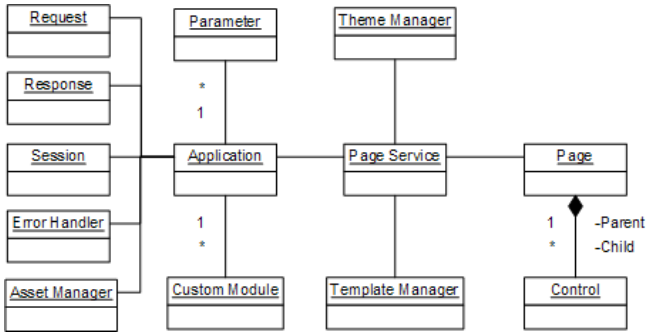
6.2 Components

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

6.2.1 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 );
```

6.2.2 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.)

6.2.3 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,

```
<div class="source">
```

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.

6.2.4 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" />
```

6.3 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.

6.3.1 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.

6.3.2 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`.

6.3.3 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.

6.3.4 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.

6.4 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`.

6.4.1 PostBack

A form submission is called ¡i¿postback¡/i¿ 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. `OnClick` and `OnCommand` events for `TButton`).

6.4.2 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,

6.5 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.

6.5.1 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.

6.5.2 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.

6.5.3 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.

6.5.4 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.

6.5.5 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`).

Custom modules and core modules are all configurable via configurations.

6.6 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.

6.6.1 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.

6.7 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.

6.7.1 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.

6.7.2 Application Deployment

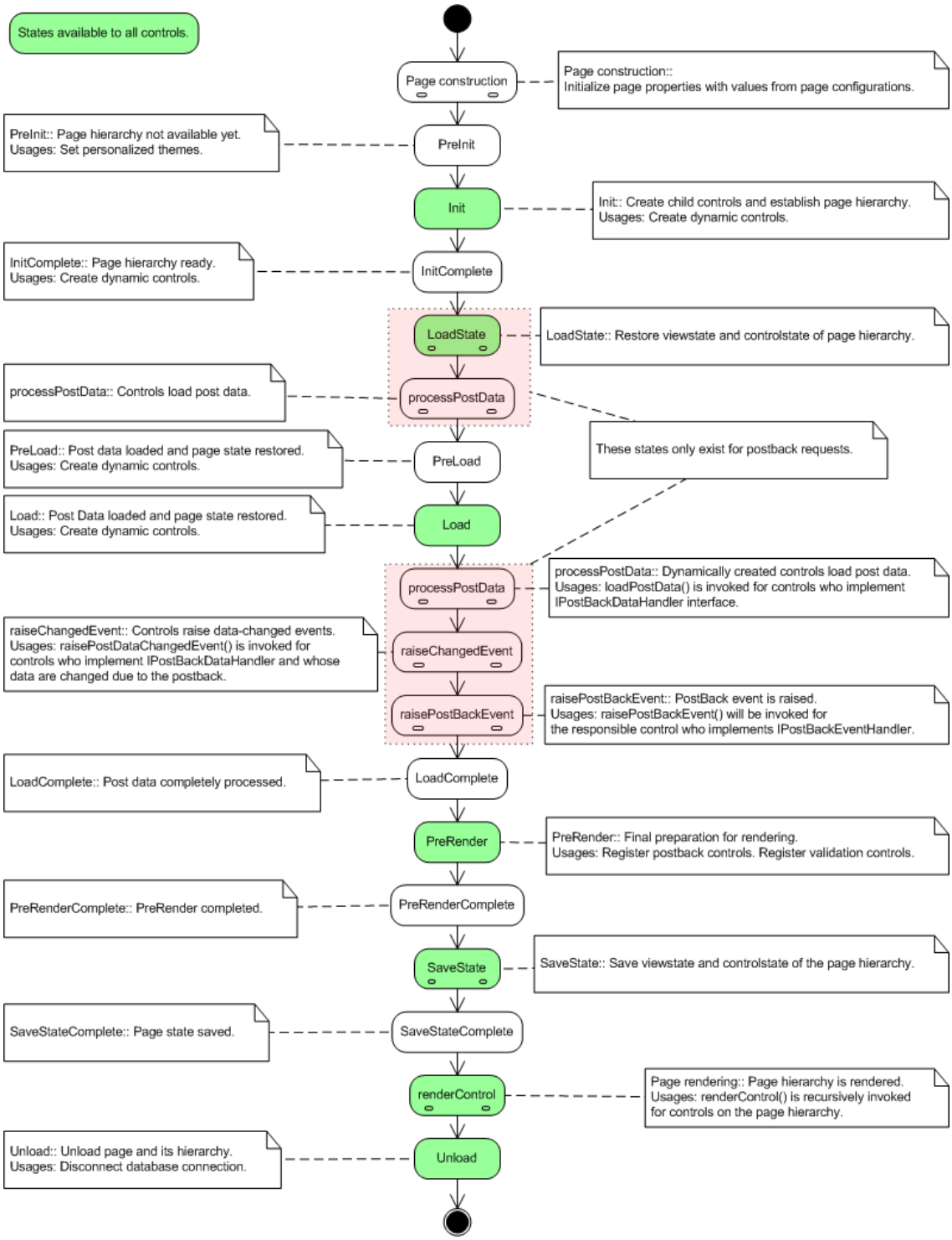
Deploying a PRADO application mainly involves copying directories. For example, to deploy the above minimal application to another server, follow the following steps,

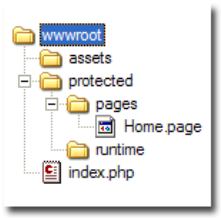
1. Copy the content under `wwwroot` to a Web-accessible directory on the new server.
2. Modify the entry script file `index.php` so that it includes correctly the `prado.php` file.
3. Remove all content under `assets` and `runtime` directories and make sure both directories are writable by the Web server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

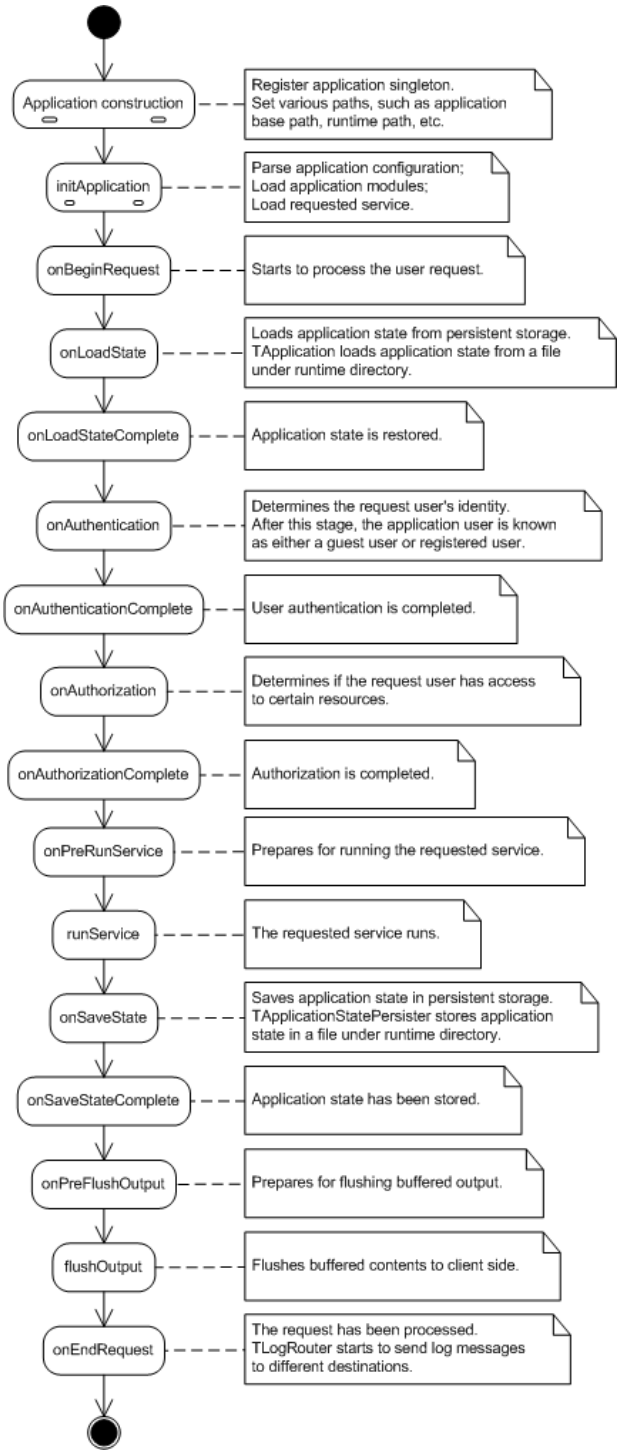
6.7.3 Application Lifecycles

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,


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
Register application singleton.
Set various paths, such as application base path, runtime path, etc.
initApplication
Parse application configuration;
Load application modules;
Load requested service.
onBeginRequest
Starts to process the user request.
onLoadState
Loads application state from persistent storage.
TApplication loads application state from a file under runtime directory.
onLoadStateComplete
Application state is restored.
onAuthentication
Determines the request user's identity.
After this stage, the application user is known as either a guest user or registered user.
onAuthenticationComplete
User authentication is completed.
onAuthorization
Determines if the request user has access to certain resources.
onAuthorizationComplete
Authorization is completed.
onPreRunService
Prepares for running the requested service.
runService
The requested service runs.
onSaveState
Saves application state in persistent storage.
TApplicationStatePersister stores application state in a file under runtime directory.
onSaveStateComplete
Application state has been stored.
onPreFlushOutput
Prepares for flushing buffered output.
flushOutput
Flushes buffered contents to client side.
onEndRequest
The request has been processed.
TLogRouter starts to send log messages to different destinations.

Chapter 7

Configurations

7.1 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.

7.2 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. .

7.2.1 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 facilitate initializing properties with big trunk of data, the following property initialization tag is introduced. It is equivalent to `...PropertyName="PropertyValue"...` in every aspect. Property initialization tags must be directly enclosed between the corresponding opening and closing component tag.

```
<prop:PropertyName>
```

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

Since version 3.1.0, the property initialization tag can also be used to initialize a set of sub-properties who share the same parent property. For example, the following is equivalent to `HeaderStyle.BackColor="black"` and `HeaderStyle.ForeColor="red"`.

```
<prop:HeaderStyle BackColor="black" ForeColor="red" />
```

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`.

7.2.2 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.

7.2.3 Comment Tags

Comment tags are used to put in a template developer comments that will not display to end-users. Contents enclosed within a comment tag will be treated as raw text strings and PRADO will not attempt to parse them. Comment tags cannot be used within property values. The format of comment tags is as follows,

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

> **Note:** The new comment tag `<!--- ... --->` has been introduced since PRADO version 3.1. Previously, it was `<!-- ... --!>` which was deprecated because some editors have problems in syntax-highlighting such tags.

7.2.4 Include Tags

Since version 3.0.5, PRADO starts to support external template inclusion. This is accomplished via include tags, where external template files are specified in namespace format and their file name must be terminated as `.tpl`.

```
<%include path.to.templateFile %>
```

External templates will be inserted at the places where the include tags occur in the base template.

Note, nested template inclusion is not supported, i.e., you cannot have include tags in an external template.

7.3 Templates: Part II

7.3.1 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 state-

ments. 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 in `PreRender` stage. The expression evaluation result is inserted at the place where the tag resides in the template. The context (namely `$this`) of the expression is the control owning the template.

The format of an expression tag is as follows,

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

For example, the following expression tag will display the current page title at the place,

```
<%= $this->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. The context (namely `$this`) of the statements is the control owning the template. 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. The context (namely `$this`) of a databind expression is the control owning the template. 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. Localization tags are in fact shortcuts to the function call `Prado::localize(string)`.

URL Tags

URL tags are used to insert the relative web url path to the Prado application in the template. You can use it in the following format:

```
<%/ image.jpg %>
```

If your Prado application is deployed on http://localhost/pradoapp/, the tag above will produce "/pradoapp/image.jpg". This tag will help you to use the correct file path even with UrlFormat set to Path, or if you are using url mappings.

7.4 Templates: Part III

7.4.1 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 control is in `PreRender` stage. 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 control owning the template. 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 %> />
```

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 owning the template. Databind tags do not apply to all components. They can only be used for controls.

The format of databind tags is as follows,

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

Since v3.0.2, expression tags and databind tags can be embedded within static strings. For example, you can write the following in a template,

```
<com:TLabel>
  <prop:Text>
  Today is <%= date('F d, Y',time()) %>.
```

```
  The page class is <%= get_class($this) %>.
  </prop:Text>
</com:TLabel>
```

Previously, you would have to use a single expression with string concatenations to achieve the same effect.

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 asset tags are evaluated when the template is instantiated.

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. The localization tags are evaluated when the template is instantiated. Localization tags are in fact shortcuts to the function call `Prado::localize(string)`.

7.5 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. Complete specification of application configurations can be found in the DTD and XSD files.

```
<application PropertyName="PropertyValue" ...>
  <paths>
    <alias id="AliasID" path="AliasPath" />
    <using namespace="Namespace" />
  </paths>
  <modules>
    <module id="ModuleID" class="ModuleClass"  PropertyName="PropertyValue" ... />
  </modules>
  <parameters>
    <parameter id="ParameterID" class="ParameterClass" PropertyName="PropertyValue" ... />
  </parameters>
  <include file="path.to.extconfig" when="PHP expression" />
  <services>
    <service id="ServiceID" class="ServiceClass" PropertyName="PropertyValue" ... />
  </services>
</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 `<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. In addition, the `System.Util.TParameterModule` module provides a way to load parameters from an external XML file. See more details in its API documentation.

- The `<include>` element allows one to include external configuration files. It has been introduced since v3.1.0. The `file` attribute specifies the external configuration file in namespace format. The extension name of the file must be `.xml`. The `when` attribute contains a PHP expression and is optional (defaults to true). Only when the expression evaluates true, will the external configuration file be included. The context of the expression is the application, i.e., `$this` in the expression would refer to the application object.

- The `<services>` element is similar to the `<modules>` element. It mainly specifies the services provided by the application. Within a `<service>` element, one can have any of the above elements. They will be effective only when the corresponding service is being requested.

An external configuration file has the same format as described above. Although the name of the root element does not matter, it is recommended to be `<configuration>`. External configurations will append to the main configuration. For example, if a path alias is specified in an external configuration, it will become available in addition to those aliases specified in the main configuration.

By default without explicit configuration, a PRADO application 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.

7.6 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>
  <parameters>
    <parameter id="ParameterID" class="ParameterClass" PropertyName="PropertyValue" ... />
```

```
  </parameters>
  <include file="path.to.extconfig" when="PHP expression" />
  <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>
</configuration>
```

The `<paths>`, `<modules>`, `<parameters>` and `<include>` are similar to those in an application configuration. The `<authorization>` element specifies the authorization rules that apply to the current page directory and all its subdirectories. For more details, see authentication and authorization section. 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.

Complete specification of page configurations can be found in the DTD and XSD files.

Since version 3.1.1, the `id` attribute in the ¡page¿ element can be a relative page path pointing to a page in the subdirectory of the directory containing the page configuration. For example, `id="admin.Home"` refers to the `Home` page under the `admin` directory. The `id` attribute can also contain wildcard '*' to match all pages under the specified directory. For example, `id="admin.*"` refers to all pages under the `admin` directory and its subdirectories. This enhancement allows developers to centralize their page configurations (e.g. put all page initializations in the aplication configuration or the root page configuration.)

7.7 URL Mapping (Friendly URLs)

System.Web.TUrlMapping API Reference

The TUrlMapping module allows PRADO to construct and recognize friendly URLs based on specific patterns.

TUrlMapping consists of a list of URL patterns which are used to match against the currently requested URL. The first matching pattern will then be used to decompose the URL into request

parameters (accessible via `$this->Request['paramname']`). The patterns can also be used to construct customized URLs. In this case, the parameters in an applied pattern will be replaced with the corresponding GET variable values.

To use `TUrlMapping`, one must set the `UrlManager` property of the `THttpRequest` module as the `TUrlMapping` module ID. See following for an example,

```
<modules>
    <module id="request" class="THttpRequest" UrlManager="friendly-url" />
    <module id="friendly-url" class="System.Web.TUrlMapping">
        <url ServiceParameter="Posts.ViewPost" pattern="post/{id}/" parameters.id="\d+" />
        <url ServiceParameter="Posts.ListPost" pattern="archive/{time}/" parameters.time="\d{6}" />
        <url ServiceParameter="Posts.ListPost" pattern="category/{cat}/" parameters.cat="\d+" />
    </module>
</modules>
```

The above example is part of the application configuration of the `blog` demo in the PRADO release. It enables recognition of the following URL formats:

- `/index.php/post/123` is recognized as `/index.php?page=Posts.ViewPost&id=123`
- `/index.php/archive/200605` is recognized as `/index.php?page=Posts.ListPost&time=200605`
- `/index.php/category/2` is recognized as `/index.php?page=Posts.ListPost&cat=2`

The `ServiceParameter` and `ServiceID` (the default ID is 'page') set the service parameter and service ID, respectively, of the Request module. The service parameter for the `TPageService` service is the Page class name, e.g., for an URL "index.php?page=Home", "page" is the service ID and the service parameter is "Home". Other services may use the service parameter and ID differently. See Services for further details.

> **Info:** The `TUrlMapping` must be configured before the request module resolves the request. This means delcaring the `TUrlMapping` outside of the `<services>` element in the application configuration. Specifying the mappings in the per directory `config.xml` is not supported.

7.7.1 Specifying URL Patterns

`TUrlMapping` enables recognition of customized URL formats based on a list prespecified of URL patterns. Each pattern is specified in a `<url>` tag.

The `Pattern` and `Parameters` attribute values are regular expression patterns that determine the mapping criteria. The `Pattern` property takes a regular expression with parameter names enclosed between a left brace '{' and a right brace '}'. The patterns for each parameter can be set using `Parameters` attribute collection. For example,

```
<url ServiceParameter="ArticleView" pattern="articles/{year}/{month}/{day}"
     parameters.year="\d{4}" parameters.month="\d{2}" parameters.day="\d+" />
```

The example is equivalent to the following regular expression (it uses the "named group" feature in regular expressions available in PHP):

```
<url ServiceParameter="ArticleView"
     RegularExpression="/articles\/(?P<year>\d{4})\/(?P<month>\d{2})\/(?P<day>\d+)/u" />
```

In the above example, the pattern contains 3 parameters named "`year`", "`month`" and "`day`". The pattern for these parameters are, respectively, "`{4}`" (4 digits), "`{2}`" (2 digits) and "`+`" (1 or more digits). Essentially, the `Parameters` attribute name and values are used as substrings in replacing the placeholders in the `Pattern` string to form a complete regular expression string.

> **Note:** If you intended to use the `RegularExpression` property you need to escape the slash in regular expressions.

Following from the above pattern example, an URL `http://example.com/index.php/articles/2006/07/21` will be matched and valid. However, `http://example.com/index.php/articles/2006/07/hello` is not valid since the `day` parameter pattern is not satisfied. In the default `TUrlMappingPattern` class, the pattern is matched against the $\mathbf{PATH}_{INFO part of the URL only. For example, only the}$`/articles/2006/07/2`

The mapped request URL is equivalent to `index.php?page=ArticleView&year=2006&month=07&day=21`. The request parameter values are available through the standard `Request` object. For example, `$this->Request['year']`.

The URL mapping are evaluated in order they are placed and only the first pattern that matches the URL will be used. Cascaded mapping can be achieved by placing the URL mappings in particular order. For example, placing the most specific mappings first.

Since version 3.1.4, Prado also provides wildcard patterns to use friendly URLs for a bunch of pages in a directory with a single rule. Therefore you can use the {*}

wildcard in your pattern to let Prado know, where to find the ServiceID in your request URL. You can also specify parameters with these patterns if a lot of pages share common parameters.

```
<url ServiceParameter="Posts.*" pattern="posts/{*}/{id}" parameters.id="\d+" />
<url ServiceParameter="Posts.*" pattern="posts/{*}" />
<url ServiceParameter="Static.Info.*" pattern="info/{*}" />
```

With these rules, any of the following URLs will be recognized:

- `/index.php/post/ViewPost/123` **is recognized as** `/index.php?page=Posts.ViewPost&id=123`
- `/index.php/post/ListPost/123` **is recognized as** `/index.php?page=Posts.ListPost&id=123`
- `/index.php/post/ListPost/123` **is recognized as** `/index.php?page=Posts.ListPost&id=123`
- `/index.php/post/MyPost` **is recognized as** `/index.php?page=Posts.MyPost`
- `/index.php/info/Conditions` **is recognized as** `/index.php?page=Static.Info.Conditions`
- `/index.php/info/About` **is recognized as** `/index.php?page=Static.Info.About`

As above, put more specific rules before more common rules as the first matching rule will be used.

To make configuration of friendly URLs for multiple pages even easier, you can also use `UrlFormat="Path"` **in combination with wildcard patterns. In fact, this feature only is available in combination with wildcard rules:¡/P¿**

```
<url ServiceParameter="user.admin.*" pattern="admin/{*}" UrlFormat="Path"/>
<url ServiceParameter="*" pattern="{*}" UrlFormat="Path" />
```

Parameters will get appended to the specified patterns as name/value pairs, separated by a "/". (You can change the separator character with `UrlParamSeparator`**.)**

- `/index.php/list/cat/15/month/12` **is recognized as** `/index.php?page=list&cat=15&month=12`
- `/index.php/edit/id/12` **is recognized as** `/index.php?page=list&id=12`
- `/index.php/show/name/foo` **is recognized as** `/index.php?page=show&name=foo`
- `/index.php/admin/edit/id/12` **is recognized as** `/index.php?page=user.admin.edit&id=12`

7.7.2 Constructing Customized URLs

Since version 3.1.1, `TUrlMapping` starts to support constructing customized URLs based on the provided patterns. To enable this feature, set `TUrlMapping.EnableCustomUrl` to true. When `THttpRequest.constrcutUrl()` is invoked, the actual URL construction work will be delegated to a matching `TUrlMappingPattern` instance. It replaces the parameters in the pattern with the corresponding GET variables passed to `constructUrl()`.

A matching pattern is one whose `ServiceID` and `ServiceParameter` properties are the same as those passed to `constructUrl()` and whose named parameters are found in the GET variables. For example, `constructUrl('Posts.ListPost',array('cat'=>2))` will use the third pattern in the above example.

By default, `TUrlMapping` will construct URLs prefixed with the currently requesting PHP script path, such as `/path/to/index.php/article/3`. Users may change this behavior by explicitly specifying the URL prefix through its `UrlPrefix` property. For example, if the Web server configuration treats `index.php` as the default script, we can set `UrlPrefix` as `/path/to` and the constructed URL will look like `/path/to/article/3`.

> **Note: If you use `constructUrl()` with string parameters that contain slashes ("/") they will get encoded to %2F. By default most Apache installations give a "404 Not found" if a URL contains a %2F. You can add `AllowEncodedSlashes On` to your `VirtualHost` configuration to resolve this. (Available since Apache 2.0.46).**

Chapter 8

Control Reference : Standard Controls

8.1 TButton

System.Web.UI.WebControls.TButton API Reference

`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, `OnClick` and `OnCommand`, when it is clicked on the client-side. The difference between `OnClick` and `OnCommand` events is that the latter event is bubbled up to the button's ancestor controls. An `OnCommand` 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`.

Controls.Samples.TButton.Home Demo

TODO: custom attributes

8.2 TCheckBox

System.Web.UI.WebControls.TCheckBox API Reference

`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`.

Controls.Samples.TCheckBox.Home Demo

8.3 TClientScript

System.Web.UI.WebControls.TClientScript API Reference

8.3.1 Including Bundled Javascript Libraries in Prado

`TClientScript` allows Javascript code to be insert or linked to the page template. PRADO is bundled with a large library of Javascript functionality including effects, AJAX, basic event handlers, and many others. The bundled Javascript libraries can be linked to the current page template using the `PradoScripts` property. Multiple bundled Javascript libraries can be specified using comma delimited string of the name of Javascript library to include on the page. For following example will include the "ajax" and "effects" library.

```
<com:TClientScript PradoScripts="ajax, effects" />
```

The available bundled libraries included in Prado are

- `prado` : basic prado javascript framework based on Prototype

- `effects` : visual effects from script.aculo.us
- `ajax` : ajax and callback related based on Prototype
- `validator` : validation
- `logger` : javascript logger and object browser
- `datepicker` : datepicker
- `colorpicker` : colorpicker

Many of the libraries, such as `validator` and `datepicker` will automatically when controls that uses these libraries are visible on the page. For example, all the validators if they have their `EnableClientScript` set to true will include both the `prado` and `validator` javascript libraries. The dependencies for each library are automatically resolved. That is, specifying, say the "ajax", will also include the "prado" library.

8.3.2 Including Custom Javascript Files

Custom Javascript files can be register using the `ScriptUrl` property. The following example includes the Javascript file "test.js" to the page. In this case, the file "test.js" is relative the current template you are using. Since the property value is dynamic asset tag, the file "test.js" will be published automatically, that is, the file will be copied to the assets directory if necessary.

```
<com:TClientScript ScriptUrl=<%~ test.js %> />
```

You can include Javascript files from other servers by specifying the full URL string in the `ScriptUrl` property.

8.3.3 Including Custom Javascript Code Blocks

Any content within the `TClientScript` control tag will be considered as Javascript code and will be rendered where it is declared.

8.4 TColorPicker

System.Web.UI.WebControls.TColorPicker API Reference

TBD

8.5 TDatePicker

System.Web.UI.WebControls.TDatePicker API Reference

`TDatePicker` **displays a text box for date input purpose. When the text box receives focus, a calendar will pop up and users can pick up from it a date that will be automatically entered into the text box. The format of the date string displayed in the text box is determined by the** `DateFormat` **property. Valid formats are the combination of the following tokens:**

```
Character      Format Pattern (en-US)
-----------------------------------------------------------------------
 d              day digit
 dd             padded day digit e.g. 01, 02
 M              month digit
 MM             padded month digit
 MMM            localized abbreviated month names, e.g. Mar, Apr
 MMMM           localized month name, e.g. March, April
 yy             2 digit year
 yyyy           4 digit year
-----------------------------------------------------------------------
```

The date of the date picker can be set using the `Date` **or** `Timestamp` **properties. The** `Date` **property value must be in the same format as the pattern specified in the** `DateFormat` **property. The** `Timestamp` **property only accepts integers such as the Unix timestamp.**

TDatePicker has three `Mode` **to show the date picker popup.**

- `Basic` **- Only shows a text input, focusing on the input shows the date picker.**
- `Button` **- Shows a button next to the text input, clicking on the button shows the date, button text can be by the** `ButtonText` **property.**

- `ImageButton` - Shows an image next to the text input, clicking on the image shows the date picker, image source can be change through the `ImageUrl` property.

The `CssClass` property can be used to override the CSS class name for the date picker panel. The `CalendarStyle` property changes the overall calendar style. The following `CalendarStyle` values are available:

- `default` - The default calendar style.

The `InputMode` property can be set to "TextBox" or "DropDownList" with default as "TextBox". In `DropDownList` mode, in addition to the popup date picker, three drop down list (day, month and year) are presented to select the date . When `InputMode` equals "DropDownList", the order and appearance of the date, month, and year will depend on the pattern specified in `DateFormat` property.

The popup date picker can be hidden by specifying `ShowCalendar` as false. Much of the text of the popup date picker can be changed to a different language using the `Culture` property.

The calendar picker year limit can be set using the `FromYear` and `UpToYear` properties where `FromYear` is the starting year and `UpToYear` is the last year selectable. The starting day of the week can be changed by the `FirstDayOfWeek` property, with 0 as Sunday, 1 as Monday, etc.

Note 1: If the `InputMode` is "TextBox", the `DateFormat` should only NOT contain `MMM` or `MMMM` patterns. The server side date parser will not be able to determine the correct date if `MMM` or `MMMM` are used. When `InputMode` equals "DropDownList", all patterns can be used.

Note 2: When the `TDatePicker` is used together with a validator, the `DateFormat` property of the validator must be equal to the `DateFormat` of the `TDatePicker` AND must set `DataType`="Date" on the validator to ensure correct validation. See TCompareValidator, TDataTypeValidator and TRangeValidator for details.

Controls.Samples.TDatePicker.Home Demo

8.6 TExpression

System.Web.UI.WebControls.TExpression API Reference

`TExpression` evaluates a PHP expression and displays the evaluation result. To specify the expression to be evaluated, set the `Expression` property. Note, `TExpression` evaluates the expression during the rendering control lifecycle.

The context of the expression in a `TExpression` control is the control itself. That is, `$this` represents the control object if it is present in the expression. For example, the following template tag will display the title of the page containing the `TExpression` control.

```
<com:TExpression Expression="$this->Page->Title" />
```

Be aware, since `TExpression` allows execution of arbitrary PHP code, in general you should not use it to evaluate expressions submitted by your application users.

Controls.Samples.TExpression.Home Demo

8.7 TFileUpload

System.Web.UI.WebControls.TFileUpload API Reference

`TFileUpload` displays a file upload field on a Web page. Upon postback, the text entered into the field will be treated as the (local) name of the file that is uploaded to the server.

`TFileUpload` raises an `OnFileUpload` event when it is post back. The property `HasFile` indicates whether the file upload is successful or not. If successful, the uploaded file may be saved on the server by calling `saveAs()` method.

The following properties give the information about the uploaded file:

- `FileName` - the original client-side file name without directory information.
- `FileType` - the MIME type of the uploaded file.

- `FileSize` - the file size in bytes.
- `LocalName` - the absolute file path of the uploaded file on the server. Note, this file will be deleted after the current page request is completed. Call `saveAs()` to save the uploaded file.

If the file upload is unsuccessful, the property `ErrorCode` gives the error code describing the cause of failure. See PHP documentation for a complete explanation of the possible error codes.

Controls.Samples.TFileUpload.Home Demo

8.8 THead

System.Web.UI.WebControls.THead API Reference

TBD

8.9 THiddenField

System.Web.UI.WebControls.THiddenField API Reference

`THiddenField` represents a hidden field on a Web page. The value of the hidden field can be accessed via its `Value` property.

`THiddenField` raises an `OnValueChanged` event if its value is changed during postback.

8.10 THtmlArea

System.Web.UI.WebControls.THtmlArea API Reference

`THtmlArea` displays a WYSIWYG text input field on a Web page to collect input in HTML format. The text displayed in the `THtmlArea` control is specified or determined by using the `Text` property. To adjust the size of the input region, set `Width` and `Height`

properties instead of Columns and Rows because the latter has no meaning under this situation. To disable the WYSIWYG feature, set EnableVisualEdit to false.

THtmlArea provides the WYSIWYG feature by wrapping the functionalities provided by the TinyMCE project.

The default editor gives only the basic tool bar. To change or add additional tool bars, use the Options property to add additional editor options with each options on a new line. See TinyMCE website for a complete list of options. The following example displays a toolbar specific for HTML table manipulation,

```
<com:THtmlArea>
  <prop:Options>
    plugins : "table"
    theme_advanced_buttons3 : "tablecontrols"
  </prop:Options>
</com:THtmlArea>
```

The client-side visual editing capability is supported by Internet Explorer 5.0+ for Windows and Gecko-based browser. If the browser does not support the visual editing, a traditional textarea will be displayed.

<pre>

Controls.Samples.THtmlArea.Home Demo

8.11 THyperLink

System.Web.UI.WebControls.THyperLink API Reference

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.

Controls.Samples.THyperLink.Home Demo

8.12 TImageButton

System.Web.UI.WebControls.TImageButton API Reference

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 OnClick event (not OnCommand).

Controls.Samples.TImageButton.Home Demo

8.13 TImageMap

System.Web.UI.WebControls.TImageMap API Reference

TImageMap represents an image on a Web page with predefined hotspot regions that can respond differently to users' clicks on them. Depending on the HotSpotMode of the hotspot region, clicking on the hotspot may trigger a postback or navigate to a specified URL.

Each hotspot is described using a THotSpot object and is maintained in the HotSpots collection in TImageMap. A hotspot can be a circle, rectangle, polygon, etc.

Hotspots can be added to TImageMap via its HotSpots property or in a template like the following,

```
<com:TImageMap ... >
  <com:TCircleHotSpot ... />
  <com:TRectangleHotSpot ... />
  <com:TPolygonHotSpot ... />
</com:TImageMap>
```

Controls.Samples.TImageMap.Home Demo

8.14 TImage

System.Web.UI.WebControls.TImage API Reference

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.

Controls.Samples.TImage.Home Demo

8.15 TInlineFrame

System.Web.UI.WebControls.TInlineFrame API Reference

TInlineFrame displays an inline frame (¡iframe¿) on a Web page. The location of the frame content is specified by the FrameUrl property.

The appearance of a TInlineFrame may be customized with the following properties, in addition to those inherited from TWebControl.

- **Align - the alignment of the frame.**
- **DescriptionUrl - the URI of a long description of the frame's contents.**
- **MarginWidth and MarginHeight - the number of pixels to use as the left/right margins and top/bottom margins, respectively.**
- **ScrollBars - whether scrollbars are provided for the inline frame. By default, it is Auto, meaning the scroll bars appear as needed. Setting it as None or Both to explicitly hide or show the scroll bars.**

The following samples show TInlineFrame with different property settings. The Google homepage is used as the frame content.

Controls.Samples.TInlineFrame.Home Demo

8.16 TJavascriptLogger

System.Web.UI.WebControls.TJavascriptLogger API Reference

`TJavascriptLogger` **provides logging for client-side javascript. It is mainly a wrapper of the Javascript developed at http://gleepglop.com/javascripts/logger/.**

To use `TJavascriptLogger`**, simply place the following component tag in a page template.**

```
<com:TJavascriptLogger />
```

Then, the client-side Javascript may contain the following statements. When they are executed, they will appear in the logger window.

```
Logger.info('something happend');
Logger.warn('A warning');
Logger.error('This is an error');
Logger.debug('debug information');
```

To toggle the visibility of the logger and console on the browser window, press ALT-D (or CTRL-D on OS X).

8.17 TLabel

System.Web.UI.WebControls.TLabel API Reference

`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.**

Controls.Samples.TLabel.Home Demo

8.18 TLinkButton

System.Web.UI.WebControls.TLinkButton API Reference

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.

Controls.Samples.TLinkButton.Home Demo

8.19 TLiteral

System.Web.UI.WebControls.TLiteral API Reference

TLiteral displays a static text on a Web page. TLiteral is similar to the TLabel control, except that the TLiteral * control has no style properties, such as BackColor, Font, etc.

The text displayed by TLiteral can be programmatically controlled by setting the Text property. The text displayed may be HTML-encoded if the Encode is true (the default value is false).

TLiteral will render the contents enclosed within its component tag if Text is empty.

Be aware, if Encode is false, make sure Text does not contain unwanted characters that may bring security vulnerabilities.

Controls.Samples.TLiteral.Home Demo

8.20 TMultiView

System.Web.UI.WebControls.TMultiView API Reference

TMultiView serves as a container for a group of TView controls, which can be retrieved by the Views property. Each view contains child controls. TMultiView determines which view and its child controls are visible. At any time, at most one view is visible

(called ¡i¿active¡/i¿). To make a view active, set `ActiveView` or `ActiveViewIndex`. Note, by default there is no active view.

To add a view to `TMultiView`, manipulate the `Views` collection or add it in template as follows,

```
<com:TMultiView>
   <com:TView>
      view 1 content
   </com:TView>
   <com:TView>
      view 2 content
   </com:TView>
</com:TMultiView>
```

`TMultiView` responds to the following command events to manage the visibility of its views.

- **`NextView` : switch to the next view (with respect to the currently active view).**
- **`PreviousView` : switch to the previous view (with respect to the currently active view).**
- **`SwitchViewID` : switch to a view by its ID path. The ID path is fetched from the command parameter.**
- **`SwitchViewIndex` : switch to a view by its zero-based index in the `Views` collection. The index is fetched from the command parameter.**

Upon postback, if the active view index is changed, `TMultiView` will raise an `OnActiveViewChanged` event.

The Hangman game is a typical use of `TMultiView`. The following example demonstrates another usage of `TMultiView`.

Controls.Samples.TMultiView.Home Demo

8.21 TOutputCache

System.Web.UI.WebControls.TOutputCache API Reference

TOutputCache enables caching a portion of a Web page, also known as partial caching. The content being cached are HTML page source coming from static texts on a PRADO template or rendered by one or several controls on the template. When the cached content is used, controls generating the content are no longer created for the page hierarchy and thus significant savings in page processing time can be achieved. The side-effect, as you might already find out, is that the content displayed may be stale if the cached version is shown to the users.

To use TOutputCache, simply enclose the content to be cached within the TOutputCache component tag on a template (either page or non-page control template), e.g.,

```
<com:TOutputCache>
   content to be cached
</com:TOutputCache>
```

where content to be cached can be static text and/or template tags. If the latter, the rendering results of the template tags will be cached. You can place one or several TOutputCache on a single template and they can be nested.

> **Note: TOutputCache stores cached content via PRADO cache modules (e.g. TSqliteCache) and thus requires at least one cache module loaded when the application runs.**

The validity of the cached content is determined based on two factors: the Duration and the cache dependency. The former specifies the number of seconds that the data can remain valid in cache (defaults to 60s), while the latter specifies conditions that the cached data depends on. If a dependency changes (e.g. relevant data in DB are updated), the cached data will be invalidated and discarded.

There are two ways to specify cache dependency. One may write event handlers to respond to the OnCheckDependency event and set the event parameter's IsValid property to indicate whether the cached data remains valid or not. One can also extend TOutputCache and override its getCacheDependency() method.

The content fetched from cache may be variated with respect to some parameters. `TOutputCache` supports variation with respect to request parameters, which is specified by `VaryByParam` property. If a specified request parameter is different, a different version of cached content is used. This is extremely useful if a page's content may be variated according to some GET parameters. The content being cached may also be variated with user sessions if `VaryBySession` is set true. To variate the cached content by other factors, override `calculateCacheKey()` method.

Output caches can be nested. An outer cache takes precedence over an inner cache in determining the validity of cached contents. This means, if the content cached by the inner cache expires or is invalidated, while that by the outer cache not, the outer cached content will be used.

By default, `TOutputCache` is effective only for non-postback page requests and when a cache module is enabled. Do not attempt to address child controls of `TOutputCache` when the cached content is currently being used. Use `ContentCached` property to determine whether the content is cached or not.

8.22 TPager

System.Web.UI.WebControls.TPager API Reference

`TPager` creates a pager that provides UI for end-users to interactively specify which page of data to be rendered in a `TDataBoundControl`-derived control, such as `TDataList`, `TRepeater`, `TCheckBoxList`, etc. The target data-bound control is specified by the `ControlToPaginate` property, which must be the ID path of the target control reaching from the pager's naming container.

Note, the target data-bound control must have its `AllowPaging` set to true. Otherwise the pager will be invisible. Also, in case when there is only one page of data available, the pager will also be invisible.

`TPager` can display one of the following three types of user interface, specified via its `Mode` property:

- `NextPrev` - a next page and a previous page button are rendered on each page.
- `Numeric` - a list of page index buttons are rendered.

- `DropDownList` - a dropdown list of page indices is rendered.

These user interfaces may be further customized by configuring the following properties

- `NextPageText` and `PrevPageText` - the label of the next/previous page button. These properties are used when the pager `Mode` is `NextPrev` or `Numeric`.
- `FirstPageText` and `LastPageText` - the label of the first/last page button. If empty, the corresponding button will not be displayed. These properties are used when the pager `Mode` is `NextPrev` or `Numeric`.
- `PageButtonCount` - the maximum number of page index buttons on a page. This property is used when the pager `Mode` is `Numeric`.
- `ButtonType` - type of page buttons, either `PushButton` meaning normal form submission buttons, or `LinkButton` meaning hyperlink buttons.

`TPager` raises an `OnPageIndexChanged` event when an end-user interacts with it and specifies a new page (e.g. by clicking on a next page button that would lead to the next page.) Developers may write handlers to respond to this event and obtain the desired new page index from the event parameter's property `NewPageIndex`. Using this new page index, one can feed a new page of data to the associated data-bound control.

Controls.Samples.TPager.Sample1 Demo

8.23 TPanel

System.Web.UI.WebControls.TPanel API Reference

`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 set `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.

Controls.Samples.TPanel.Home Demo

8.24 TPlaceHolder

System.Web.UI.WebControls.TPlaceHolder API Reference

`TPlaceHolder` reserves a place on a template, where static texts or controls may be dynamically inserted.

Controls.Samples.TPlaceHolder.Home Demo

8.25 TRadioButton

System.Web.UI.WebControls.TRadioButton API Reference

`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.

Controls.Samples.TRadioButton.Home Demo

8.26 TSafeHtml

System.Web.UI.WebControls.TSafeHtml API Reference

`TSafeHtml` is a control that strips down all potentially dangerous HTML content. It is mainly a wrapper of the SafeHTML project. According to the SafeHTML project, it tries to safeguard the following situations when the string is to be displayed to end-users:

- **Opening tag without its closing tag**

- **closing tag without its opening tag**
- **any of these tags: base, basefont, head, html, body, applet, object, iframe, frame, frameset, script, layer, ilayer, embed, bgsound, link, meta, style, title, blink, xml, etc.**
- **any of these attributes: on*, data*, dynsrc**
- **javascript:/vbscript:/about: etc. protocols**
- **expression/behavior etc. in styles**
- **any other active content.**

To use `TSafeHtml`, simply enclose the content to be secured within the `TSafeHtml` component tag in a template. The content may consist of both static text and PRADO controls. If the latter, the rendering result of the controls will be secured.

Controls.Samples.TSafeHtml.Home Demo

8.27 TStatements

System.Web.UI.WebControls.TStatements API Reference

`TStatements` evaluates a sequence of PHP statements and displays the content rendered by the statements. To specify the PHP statements to be evaluated, set the `Statements` property. For example, the following component tag displays the current time on the Web page,

```
<com:TStatements>
  <prop:Statements>
  setlocale(LC_ALL, 'nl_NL');
  echo strftime("%A %e %B %Y",time());
  </prop:Statements>
</com:TStatements>
```

Note, `TStatements` evaluates the PHP statements during the rendering control lifecycle. Unlike `TExpression`, `TStatements` only displays the content 'echoed' within the statements.

The context of the statements in a `TStatements` control is the control itself. That is, `$this` represents the control object if it is present in the statements. For example, the following statement tag will display the title of the page containing the `TStatements` control.

```
<com:TStatements>
  <prop:Statements>
  $page=$this->Page;
  echo $page->Title;
  </prop:Statements>
</com:TStatements>
```

Be aware, since `TStatements` allows execution of arbitrary PHP code, in general you should not use it to evaluate PHP code submitted by your application users.

Controls.Samples.TStatements.Home Demo

8.28 TTabPanel

System.Web.UI.WebControls.TTabPanel API Reference

`TTabPanel` displays a tabbed panel. Users can click on the tab bar to switching among different tab views. Each tab view is an independent panel that can contain arbitrary content.

A `TTabPanel` control consists of one or several `TTabView` controls representing the possible tab views. At any time, only one tab view is visible (active), which is specified by any of the following properties:

- **`ActiveViewIndex` - the zero-based integer index of the view in the view collection.**
- **`ActiveViewID` - the text ID of the visible view.**
- **`ActiveView` - the visible view instance.**

If both `ActiveViewIndex` and `ActiveViewID` are set, the latter takes precedence.

TTabPanel uses CSS to specify the appearance of the tab bar and panel. By default, an embedded CSS file will be published which contains the default CSS for TTabPanel. You may also use your own CSS file by specifying the CssUrl property. The following properties specify the CSS classes used for elements in a TTabPanel:

- **CssClass - the CSS class name for the outer-most div element (defaults to 'tab-panel');**
- **TabCssClass - the CSS class name for nonactive tab div elements (defaults to 'tab-normal');**
- **ActiveTabCssClass - the CSS class name for the active tab div element (defaults to 'tab-active');**
- **ViewCssClass - the CSS class for the div element enclosing view content (defaults to 'tab-view');**

To use TTabPanel, write a template like following:

```
<com:TTabPanel>
  <com:TTabView Caption="View 1">
    content for view 1
  </com:TTabView>
  <com:TTabView Caption="View 2">
    content for view 2
  </com:TTabView>
  <com:TTabView Caption="View 3">
    content for view 3
  </com:TTabView>
</com:TTabPanel>
```

Controls.Samples.TTabPanel.Home Demo

8.29 TTable

System.Web.UI.WebControls.TTable API Reference

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.

Controls.Samples.TTable.Home Demo

8.30 TTextBox

System.Web.UI.WebControls.TTextBox API Reference

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.

Controls.Samples.TTextBox.Home Demo

8.31 TTextHighlighter

System.Web.UI.WebControls.TTextHighlighter API Reference

TTextHighlighter does syntax highlighting for its body content, including both static text and the rendering results of its child controls. The text being highlighted follows the syntax of the specified Language, which can be 'php' (default), 'prado', 'css', 'html', etc. Here, 'prado' stands for the syntax of PRADO control templates.

If line numbers are desired in front of each line, set ShowLineNumbers to true.

To use TTextHighlighter, simply enclose the contents to be syntax highlighted within the body of a TTextHighlighter control. The following example highlights a piece of PHP code,

```
<com:TTextHighlighter ShowLineNumbers="true">
<?php
$str = 'one|two|three|four';
print_r(explode('|', $str, 2)); // will output an array
?>
</com:TTextHighlighter>
```

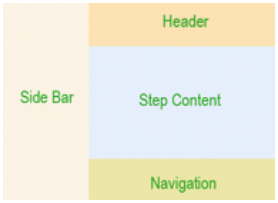
Controls.Samples.TTextHighlighter.Home Demo

8.32 TWizard

System.Web.UI.WebControls.TWizard API Reference

8.32.1 Overview

`TWizard` is analogous to the installation wizard commonly used to install software on Windows. It splits a large form and presents the user with a series of smaller forms, called wizard steps, to complete. The following figure shows how a wizard is composed of when presented to users, where `step content` is the main content of a wizard step for users to complete, `header` refers to header content common to all steps, `navigation` contains buttons that allow users to navigate step by step, and `side bar` contains a list of hyperlinks by which users can reach to any step with one click. The visibility of the side bar can be toggled by setting `ShowSideBar`.



By default, `TWizard` embeds the above components in an HTML table so that the side bar is displayed on the left while the rest on the right. If `UseDefaultLayout` is set to false, no HTML table will be used, and developers should use pure CSS techniques to position the wizard components. Note, each component is displayed as a ¡div¿ and the wizard itself is also a ¡div¿ that encloses its components' ¡div¿.

Wizard steps are represented by `TWizardStep` and are maintained in `TWizard` through its `WizardSteps` property. At any time, only one step is visible, which is determined by the `ActiveStep` property. The `ActiveStepIndex` property gives the index of the active step in the step collection. Clicking on navigation buttons can activate different wizard steps.

Wizard steps are typically added to a wizard through template as follows,

```
<com:TWizard>
     <com:TWizardStep Title="step 1" StepType="Start">
     content in step 1, may contain other controls
     </com:TWizardStep>

     <com:TWizardStep Title="step 2" StepType="Step">
     content in step 2, may contain other controls
     </com:TWizardStep>

     <com:TWizardStep Title="finish step" StepType="Finish">
     content in finish step, may contain other controls
     </com:TWizardStep>
</com:TWizard>
```

In the above, `StepType` refers to the type of a wizard step, which can affect how the navigation appearance and behavior of the step. A wizard step can be of one of the following types:

- **`Start` - the first step in the wizard.**
- **`Step` - the internal steps in the wizard.**
- **`Finish` - the last step that allows user interaction.**
- **`Complete` - the step that shows a summary to user. In this step, both side bar and navigation panel are invisible. Thus, this step usually does not allow user interaction.**
- **`Auto` - the step type is determined by wizard automatically.**

8.32.2 Using TWizard

A Single-Step Wizard Sample

In this sample, we use wizard to collect user's preference of color. In the first step, the user is presented with a dropdown list from which he can choose his favorite color. In the second step, the complete step, his choice in the previous step is displayed. In real application, at this step the choice may be stored in database in the backend.

Controls.Samples.TWizard.Sample1 Demo

Customizing Wizard Styles

`TWizard` **defines a whole set of properties for customization of appearance of its various components as shown in the above figure. In particular, the following properties are provided for style customization:**

- **Header -** `HeaderStyle`**.**
- **Step -** `StepStyle`**.**
- **Navigation -** `NavigationStyle`**,** `StartNextButtonStyle`**,** `StepNextButtonStyle`**,** `StepPreviousButtonStyle`**,** `FinishPreviousButtonStyle`**,** `FinishCompleteButtonStyle`**,** `CancelButtonStyle`**.**
- **Side bar -** `SideBarStyle`**,** `SideBarButtonStyle`**.**

Controls.Samples.TWizard.Sample2 Demo

Customizing Wizard Navigation

Given a set of wizard steps, `TWizard` **supports three different ways of navigation among them:**

- **Bidirectional - Users can navigate forward and backward along a sequence of wizard steps. User input data is usually collected at the last step. This is also known as commit-at-the-end model. It is the default navigation way that** `TWizard` **supports.**

- **Unidirectional - Users can navigate forward along a sequence of wizard steps. Backward move is not allowed. User input data is usually collected step by step. This is also known as command-as-you-go model. To disallow backward move to a specific step, set its `AllowReturn` property to false.**
- **Nonlinear - User input in a step determines which step to go next. To do so, set `ActiveStepIndex` to the step that you want the user to go to. In this case, when a user clicks on the previous button in the navigation panel, the previous step that they visited (not the previous step in the sequential order) will become visible.**

Controls.Samples.TWizard.Sample3 Demo

Using Templates in Wizard

`TWizard` supports more concrete control of its outlook through templates. In particular, it provides the following template properties that allow complete customization of the wizard's header, navigation and side bar.

- **Header - `HeaderTemplate`.**
- **Navigation - `StartNavigationTemplate`, `StepNavigationTemplate`, `FinishNavigationTemplate`.**
- **Side bar - `SideBarTemplate`.**

Controls.Samples.TWizard.Sample4 Demo

Using Templated Wizard Steps

Wizard steps can also be templated. By using `TTemplatedWizardStep`, one can customize step content and navigation through its `ContentTemplate` and `NavigationTemplate` properties, respectively. This is useful for control developers to build specialized wizards, such as user registration, shopping carts, etc.

Controls.Samples.TWizard.Sample5 Demo

Chapter 9

Control Reference : List Controls

9.1 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 post-back.**

- `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 the following two kinds of format:

- one-dimensional array or objects implementing `ITraversable` : 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();
```

9.1.1 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.

Controls.Samples.TListBox.Home Demo

9.1.2 TDropDownList

`TDropDownList` displays a dropdown list box that allows users to select a single option from a few prespecified ones.

Since v3.1.1, `TDropDownList` starts to support prompt text (something like 'Please select:' as the first list item). To use this feature, set either `PromptText` or `PromptValue`, or both. If the user chooses the prompt item, the dropdown list will be unselected.

Controls.Samples.TDropDownList.Home Demo

9.1.3 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).

Controls.Samples.TCheckBoxList.Home Demo

9.1.4 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.)

Controls.Samples.TRadioButtonList.Home Demo

9.1.5 TBulletedList

`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.**

Controls.Samples.TBulletedList.Home Demo

Chapter 10

Control Reference : Validation Controls

10.1 Validation Controls

Validation controls, called validators, perform validation on user-entered data values when they are post back to the server. The validation is triggered by a postback control, such as a `TButton`, a `TLinkButton` or a `TTextBox` (under `AutoPostBack` mode) whose `CausesValidation` property is true.

Validation is always performed on server side. If `EnableClientScript` is true and the client browser supports JavaScript, validators may also perform client-side validation. 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.

Validators share a common set of properties, which are defined in the base class `TBaseValidator` class and listed as follows,

- **`ControlToValidate` specifies the input control to be validated. This property must be set to the ID path of an input control. An ID path is the dot-connected IDs of the controls reaching from the validator's naming container to the target control.**

- `ErrorMessage` specifies the error message to be displayed in case the corresponding validator fails.
- `Text` is similar to `ErrorMessage`. If they are both present, `Text` takes precedence. This property is useful when used together with `TValidationSummary`.
- `ValidationGroup` specifies which group the validator is in. The validator will perform validation only if the current postback is triggered by a control which is in the same group.
- `EnableClientScript` specifies whether client-side validation should be performed. By default, it is enabled.
- `Display` specifies how error messages are displayed. It takes one of the following three values:
  - `None` - the error message will not be displayed even if the validator fails.
  - `Static` - the space for displaying the error message is reserved. Therefore, showing up the error message will not change your existing page layout.
  - `Dynamic` - the space for displaying the error message is NOT reserved. Therefore, showing up the error message will shift the layout of your page around (usually down).
- `ControlCssClass` - the CSS class that is applied to the control being validated in case the validation fails.
- `FocusOnError` - set focus at the validating place if the validation fails. Defaults to false.
- `FocusElementID` - the ID of the HTML element that will receive focus if validation fails and `FocusOnError` is true.

10.2 Prado Validation Controls

10.2.1 TRequiredFieldValidator

TRequiredFieldValidator ensures that the user enters some data in the specified input field. By default, TRequiredFieldValidator will check if the user input is empty or

not. The validation fails if the input is empty. By setting `InitialValue`, the validator can check if the user input is different from `InitialValue`. If not, the validation fails.

Controls.Samples.TRequiredFieldValidator.Home Demo

10.2.2 TRegularExpressionValidator

TRegularExpressionValidator verifies the user input against a regular pattern. The validation fails if the input does not match the pattern. The regular expression can be specified by the `RegularExpression` property. Some commonly used regular expressions include:

- At least 6 characters: `[\w]{6,}`
- 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/.

Note, TRegularExpressionValidator only checks for nonempty user input. Use a TRequiredFieldValidator to ensure the user input is not empty.

Controls.Samples.TRegularExpressionValidator.Home Demo

10.2.3 TEmailAddressValidator

TEmailAddressValidator verifies that the user input is a valid email address. The validator uses a regular expression to check if the input is in a valid email address format.

If `CheckMXRecord` is true, the validator will also check whether the MX record indicated by the email address is valid, provided `checkdnsrr()` is available in the installed PHP.

Note, if the input being validated is empty, TEmailAddressValidator will not do validation. Use a TRequiredFieldValidator to ensure the value is not empty.

Controls.Samples.TEmailAddressValidator.Home Demo

10.2.4 TCompareValidator

TCompareValidator compares the user input with a constant value specified by `ValueToCompare`, or another user input specified by `ControlToCompare`. The `Operator` property specifies how to compare the values, which includes `Equal`, `NotEqual`, `GreaterThan`, `GreaterThanEqual`, `LessThan` and `LessThanEqual`. Before comparison, the values being compared will be converted to the type specified by `DataType` listed as follows,

- `String` - A string data type.
- `Integer` - A 32-bit signed integer data type.
- `Float` - A double-precision floating point number data type.
- `Date` - A date data type. The date format can be specified by setting `DateFormat` property, which must be recognizable by `TSimpleDateFormatter`. If the property is not set, the GNU date syntax is assumed.

Note, if the input being validated is empty, TEmailAddressValidator will not do validation. Use a TRequiredFieldValidator to ensure the value is not empty.

N.B. If validating against a TDatePicker the `DataType` must be equal to "Date" and the `DateFormat` property of the validator must be equal to the `DateFormat` of the TDatePicker.

Controls.Samples.TCompareValidator.Home Demo

10.2.5 TDataTypeValidator

TDataTypeValidator verifies if the input data is of specific type indicated by `DataType`. The data types that can be checked against are the same as those in TCompareVal-

idator.

N.B. If validating against a TDatePicker the `DataType` must be equal to "Date" and the `DateFormat` property of the validator must be equal to the `DateFormat` of the TDatePicker.

Controls.Samples.TDataTypeValidator.Home Demo

10.2.6 TRangeValidator

TRangeValidator verifies whether an input value is within a specified range. TRangeValidator uses three key properties to perform its validation. The `MinValue` and `MaxValue` properties specify the minimum and maximum values of the valid range. The `DataType` property specifies the data type of the value being validated. The value will be first converted into the specified type and then compare with the valid range. The data types that can be checked against are the same as those in TCompareValidator.

If `StrictComparison` property is set to `true`, then the ranges are compared as strictly less than the `MaxValue` and/or strictly greater than the `MinValue`.

N.B. If validating against a TDatePicker the `DataType` must be equal to "Date" and the `DateFormat` property of the validator must be equal to the `DateFormat` of the TDatePicker.

Controls.Samples.TRangeValidator.Home Demo

10.2.7 TCustomValidator

TCustomValidator performs user-defined validation (either server-side or client-side or both) on an input control.

To create a server-side validation function, provide a handler for the `OnServerValidate` event that performs the validation. The data string of the input control to validate can be accessed by the event parameter's `Value` property. The result of the validation should be stored in the `IsValid` property of the parameter.

To create a client-side validation function, add the client-side validation javascript function to the page template and assign its name to the `ClientValidationFunction`

property. The function should have the following signature:

```
<script type="text/javascript">
function ValidationFunctionName(sender, parameter)
{
    // if(parameter == ...)
    //    return true;
    // else
    //    return false;
}
</script>
```

Controls.Samples.TCustomValidator.Home Demo

10.2.8 TValidationSummary

TValidationSummary displays a summary of validation errors inline on a Web page, in a message box, or both.

By default, a validation summary will collect `ErrorMessage` of all failed validators on the page. If `ValidationGroup` is not empty, only those validators who belong to the group will show their error messages in the summary.

The summary can be displayed as a list, a bulleted list, or a single paragraph based on the `DisplayMode` property. The messages shown can be prefixed with `HeaderText`. The summary can be displayed on the Web page or in a JavaScript message box, by setting the `ShowSummary` and `ShowMessageBox` properties, respectively. Note, the latter is only effective when `EnableClientScript` is true.

Controls.Samples.TValidationSummary.Home Demo

10.3 Interacting the Validators

10.3.1 Resetting or Clearing of Validators

Validators can be reset on the client-side using javascript by calling the `Prado.Validation.reset(groupID)` where `groupID` is the validator grouping name. If `groupID` is null, then validators without grouping are used.

```
<script type="text/javascript">
function reset_validator()
{
    Prado.Validation.reset("group1");
}
</script>
```

Controls.Samples.ResetValidation.Home Demo

10.3.2 Client and Server Side Conditional Validation

All validators contains the following events. The corresponding events for the client side is available as sub-properties of the `ClientSide` property of the validator.

- **The `OnValidate` event is raise before the validator validation functions are called.**
- **The `OnValidationSuccess` event is raised after the validator has successfully validate the control.**
- **The `OnValidationError` event is raised after the validator fails validation.**

> **Note: For Prado versions earlier than 3.1 the property names were `OnError` and `OnSuccess`. For Prado version 3.1 or later they are `OnValidationError` and `OnValidationSuccess`, respectively.**

The following example pop-up a message saying "hello" when the validator fails on the client-side.

```
<com:TRequiredFieldValidator ... >
     <prop:ClientSide.OnValidationError>
          alert("hello");
     </prop:ClientSide.OnValidationError>
</com:TRequiredFieldValidator>
```

The resulting client-side event callback function is of the following form.

```
function onErrorHandler(sender, parameter)
{
     alert("hello");
}
```

Where `sender` is the current client-side validator and `parameter` is the control that invoked the validator.

Conditional Validation Example

The following example show the use of client-side and server side validator events. The example demonstrates conditional validation. Notice that, we need to write code for both the server side and client-side. Moreover, on the server side, we need to re-enable the conditional validator so that its javascript code are produced no matter what (otherwise, the client-side validator is not available). Controls.Samples.TClientSideValidator.Home Demo

Chapter 11

Control Reference : Data Controls

11.1 Data Controls

- **TDataList is used to display or modify a list of data items.**
- **TDataGrid displays data in a tabular format with rows and columns.**
- **TRepeater displays its content defined in templates repeatedly based on the given data.**

11.2 TDataList

TDataList represents a data bound and updatable list control. Like `TRepeater`, TDataList displays its content repeatedly based on the data fetched from `DataSource`. The repeated contents in TDataList are called items, which are controls and can be accessed through `Items`. When `dataBind()` is invoked, TDataList creates an item for each row of data and binds the data row to the item. Optionally, a TDataList can have a header, a footer and/or separators between items.

TDataList differs from TRepeater in that it introduces the concept of item state and allows applying different styles to items in different states. In addition, TDataList supports tiling the repeated items in various ways.

The layout of the repeated contents in TDataList are specified by inline templates. TDataList items, header, footer, etc. are being instantiated with the corresponding templates when data is being bound to the repeater.

Since v3.1.0, the layout can also be by renderers. A renderer is a control class that can be instantiated as datalist items, header, etc. A renderer can thus be viewed as an external template (in fact, it can also be non-templated controls). For more details, see the explanation about renderer in the TRepeater tutorial.

The following properties are used to specify different types of template and renderer for a datalist. If a content type is defined with both a template and a renderer, the latter takes precedence.

- `ItemTemplate`, `ItemRenderer`: **for each repeated row of data**
- `AlternatingItemTemplate`, `AlternatingItemRenderer`: **for each alternating row of data. If not set,** `ItemTemplate` **or** `ItemRenderer` **will be used instead.**
- `HeaderTemplate`, `HeaderRenderer`: **for the datalist header.**
- `FooterTemplate`, `FooterRenderer`: **for the datalist footer.**
- `SeparatorTemplate`, `SeparatorRenderer`: **for content to be displayed between items.**
- `EmptyTemplate`, `EmptyRenderer`: **used when data bound to the datalist is empty.**
- `EditItemTemplate`, `EditItemRenderer`: **for the row being editted.**
- `SelectedItemTemplate`, `SelectedItemRenderer`: **for the row being selected.**

When `dataBind()` **is called, TDataList undergoes the following lifecycles for each row of data:**

1. **create item based on templates or renderers**
2. **set the row of data to the item**
3. **raise an** `OnItemCreated` **event**
4. **add the item as a child control**
5. **call** `dataBind()` **of the item**

6. **raise an `OnItemDataBound` event**

TDataList raises an `OnItemCommand` whenever a button control within some datalist item raises an `OnCommand` event. Therefore, you can handle all sorts of `OnCommand` event in a central place by writing an event handler for `OnItemCommand`. An additional event is raised if the `OnCommand` event has one of the following command names (case-insensitive):

- **`edit` - user wants to edit an item. `OnEditCommand` event will be raised.**
- **`update` - user wants to save the change to an item. `OnUpdateCommand` event will be raised.**
- **`select` - user selects an item. `OnSelectedIndexChanged` event will be raised.**
- **`delete` - user deletes an item. `OnDeleteCommand` event will be raised.**
- **`cancel` - user cancels previously editting action. `OnCancelCommand` event will be raised.**

TDataList provides a few properties to support tiling the items. The number of columns used to display the data items is specified via `RepeatColumns` property, while the `RepeatDirection` governs the order of the items being rendered. The layout of the data items in the list is specified via `RepeatLayout`, which takes one of the following values:

- **`Table` (default) - items are organized using HTML table and cells. When using this layout, one can set `CellPadding` and `CellSpacing` to adjust the cellpadding and cellspacing of the table, and `Caption` and `CaptionAlign` to add a table caption with the specified alignment.**
- **`Flow` - items are organized using HTML spans and breaks.**
- **`Raw` - TDataList does not generate any HTML tags to do the tiling.**

Items in TDataList can be in one of the three states: being browsed, being editted and being selected. To change the state of a particular item, set `SelectedItemIndex` or `EditItemIndex`. The former will change the indicated item to selected mode, which will cause the item to use `SelectedItemTemplate` or `SelectedItemRenderer` for presentation.

The latter will change the indicated item to edit mode and to use corresponding template or renderer. Note, if an item is in edit mode, then selecting this item will have no effect.

Different styles may be applied to items in different status. The style application is performed 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`
- `EditItemStyle`

Therefore, if background color is set as red in `ItemStyle`, `EditItemStyle` will also have red background color unless it is set to a different value explicitly.

When a page containing a datalist is post back, the datalist will restore automatically all its contents, including items, header, footer and separators. However, the data row associated with each item will not be recovered and become null. To access the data, use one of the following ways:

- **Use `DataKeys` to obtain the data key associated with the specified repeater item and use the key to fetch the corresponding data from some persistent storage such as DB.**
- **Save the whole dataset in viewstate, which will restore the dataset automatically upon postback. Be aware though, if the size of your dataset is big, your page size will become big. Some complex data may also have serializing problem if saved in viewstate.**

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

Controls.Samples.TDataList.Sample1 Demo

A common use of TDataList is for maintaining tabular data, including browsing, editing, 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.

Controls.Samples.TDataList.Sample2 Demo

11.3 TDataGrid

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 editing 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, editing 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.

11.3.1 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 editing 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.**
- `TLiteralColumn` **associates cells with a specific field of data and displays the cells with static texts.**
- `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.**
- `TDropDownListColumn` **associates cells with a specific field of data and displays the cells according to their modes. If in edit mode, a cell will be displayed with a** `TDropDownList`**.**
- `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.**

11.3.2 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.**

11.3.3 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`.

11.3.4 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 following 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();
    }
}
```

Controls.Samples.TDataGrid.Sample1 Demo

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 specifying cell contents.**

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

11.3.5 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 editing 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 editing 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`. In addition, `TDropDownListColumn` replaces the previous `TTemplateColumn` to allow users to select a rating from a dropdown list. Note, it is also possible to use `TTemplateColumn` to achieve the same task.

Controls.Samples.TDataGrid.Sample3 Demo

11.3.6 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.

Controls.Samples.TDataGrid.Sample4 Demo

11.3.7 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.

Controls.Samples.TDataGrid.Sample5 Demo

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.

Controls.Samples.TDataGrid.Sample6 Demo

11.3.8 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.

11.4 TRepeater

TRepeater displays its content repeatedly based on the data fetched from `DataSource`. The repeated contents in TRepeater are called ¡i¿items¡/i¿ which are controls accessible through `Items` property. When `dataBind()` is invoked, TRepeater creates an item for each row of data and binds the data row to the item. Optionally, a repeater can have a header, a footer and/or separators between items.

The layout of the repeated contents are specified by inline templates. In particular, repeater items, header, footer, etc. are being instantiated with the corresponding templates when data is being bound to the repeater.

Since v3.1.0, the layout can also be specified by ¡i¿renderers¡/i¿. A renderer is a control class that can be instantiated as repeater items, header, etc. A renderer can thus be viewed as an external template (in fact, it can also be non-templated controls). A renderer can be any control class. By using item renderers, one can avoid writing long and messy templates. Since a renderer is a class, it also helps reusing templates that previously might be embedded within other templates. If implemented with one of the following interfaces, a renderer will be initialized with additional properties relevant to the repeater items:

- **`IDataRenderer` - the `Data` property will be set as the row of the data bound to the repeater item. Many PRADO controls implement this interface, such as `TLabel`, `TTextBox`, etc.**
- **`IItemDataRenderer` - the `ItemIndex` property will be set as the zero-based index of the item in the repeater item collection, and the `ItemType` property as the item's type (such as `TListItemType::Item`). As a convenient base class, `TRepeaterItemRenderer` implements `IDataItemRenderer` and can have an associated template because it extends from `TTemplateControl`.**

The following properties are used to specify different types of template and renderer for a repeater. If an item type is defined with both a template and a renderer, the latter takes precedence.

- ItemTemplate, ItemRenderer - for each repeated row of data.
- AlternatingItemTemplate, AlternatingItemRenderer: for each alternating row of data. If not set, ItemTemplate or ItemRenderer will be used instead, respectively.
- HeaderTemplate, HeaderRenderer - for the repeater header.
- FooterTemplate, FooterRenderer - for the repeater footer.
- SeparatorTemplate, SeparatorRenderer - for content to be displayed between items.
- EmptyTemplate, EmptyRenderer - used when data bound to the repeater is empty.

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

When dataBind() is called, TRepeater undergoes the following lifecycles for each row of data:

1. create item based on templates or renderers
2. set the row of data to the item
3. raise an OnItemCreated event
4. add the item as a child control
5. call dataBind() of the item
6. raise an OnItemDataBound event

Normally, you only need to bind the data to repeater when the page containing the repeater is initially requested. When the page is post back, the repeater will restore automatically all its contents, including items, header, footer and separators. However, the data row associated with each item will not be recovered and thus become null.

To access the repeater item data in postbacks, use one of the following ways:

- **Use `DataKeys` to obtain the data key associated with the specified repeater item and use the key to fetch the corresponding data from some persistent storage such as DB.**
- **Save the whole dataset in viewstate, which will restore the dataset automatically upon postback. Be aware though, if the size of your dataset is big, your page size will become big. Some complex data may also have serializing problem if saved in viewstate.**

TRepeater raises an `OnItemCommand` event whenever a button control within some repeater item raises a `OnCommand` event. Therefore, you can handle all sorts of `OnCommand` event in a central place by writing an event handler for `OnItemCommand`.

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

Controls.Samples.TRepeater.Sample1 Demo

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.

Controls.Samples.TRepeater.Sample2 Demo

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.

Controls.Samples.TRepeater.Sample3 Demo

This sample shows how to use "drop-in" item renderers, available since v3.1.0. These renderers come in the PRADO release. They are essentially controls implementing the `IDataRenderer` interface. Common Web controls, such as `TTextBox`, `TLabel`, all implement this interface. When such controls are used item renderers, their `Data` property is assigned with the row of the data being bound to the repeater item.

Controls.Samples.TRepeater.Sample4 Demo

More often, one needs to customize the layout of repeater items. The sample above relies on `OnItemCreated` to adjust the appearance of the renderer. Templated item renderers are perferred in this situation, as they allow us to put in more complex layout and content in a repeater item. The following sample reimplements the nested repeater sample using a templated item renderer called `RegionDisplay`. As we can see, the new code is much easier to understand and maintain.

Controls.Samples.TRepeater.Sample5 Demo

Chapter 12

Control Reference : Active Controls (AJAX)

12.1 TActiveButton

System.Web.UI.ActiveControls.TActiveButton API Reference

`TActiveButton` is the active control counter part to TButton. When a `TActiveButton` is clicked, rather than a normal post back request a callback request is initiated. The `OnCallback` event is raised during a callback request and it is raise after the `OnClick` event.

When the `ActiveControl.EnableUpdate` property is true, changing the `Text` property during a callback request will update the button's caption on the client-side.

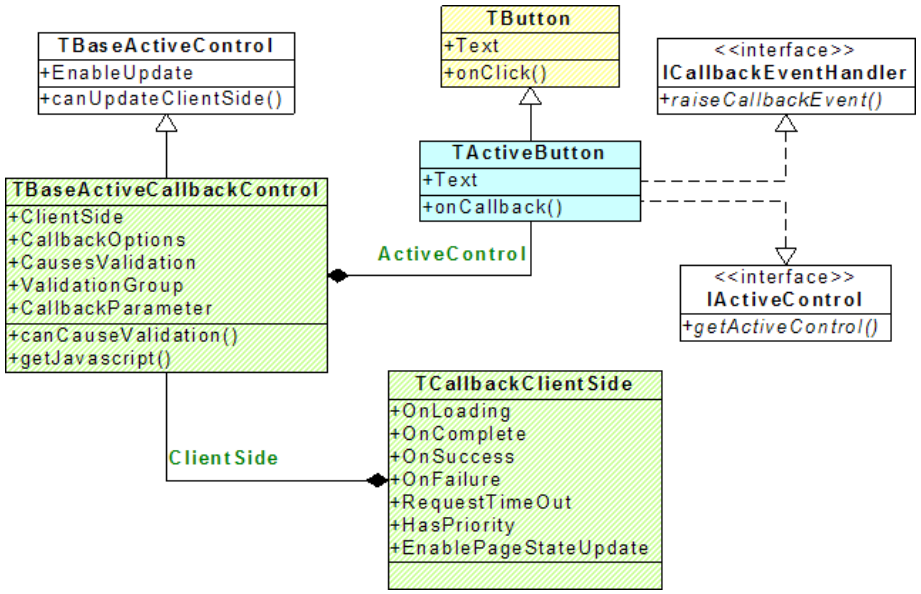
Since the `OnCallback` event is raised only during a callback request, the `OnCallback` event handler can be used to handle logic specifically related to callback requests. The `OnClick` event handler is raised when ever the button is clicked, even if javascript is disabled.

The following example the use of both the `OnClick` and `OnCallback` events of an `TActiveButton`.

ActiveControls.Samples.TActiveButton.Home Demo

12.1.1 TActiveButton Class Diagram

The class diagram for `TActiveButton` is illustrated in the figure below. Most active control that can perform callback request have a similar structure.



`TActiveButton` is an extension of TButton and implements two additional interfaces `ICallbackEventHandler` and `IActiveControl`. The `TActiveButton` contains an instance of TBaseActiveCallbackControl available through the `ActiveControl` property of `TActiveButton`. The following example set the callback parameter of the `TActiveButton` when a callback request is dispatched.

```
<com:TActiveButton
     Text="Click Me"
     OnCallback="button_callback"
     ActiveControl.CallbackParameter="value" />
```

In the `OnCallback` event handler method, the `CallbackParameter` is available in the `$param` object.

```
public function button_callback($sender, $param)
{
     echo $param->CallbackParameter; //outputs "value"
}
```

12.1.2 Adding Client Side Behaviour

With in the `ActiveControl` property is an instance of TCallbackClientSide available as a property `ClientSide` of `TActiveButton`. The `ClientSide` property contains sub-properties, such as `RequestTimeOut`, and client-side javascript event handler, such as `OnLoading`, that are used by the client-side when making a callback request. The following example demonstrates the toggling of a "loading" indicator when the client-side is making a callback request.

```
<com:TClientScript PradoScripts="effects" />
<span id="callback_status">Loading...</span>

<com:TActiveButton
     Text="Click Me"
     OnCallback="button_callback"
     ActiveControl.CallbackParameter="value" >
     <prop:ClientSide
          OnLoading="Element.show('callback_status')"
          OnComplete="Element.hide('callback_status')" />
</com:TActiveButton>
```

The example loads the "effects" javascript library using the TClientScript component. The `ClientSide.OnLoading` property value contains javascript statement that uses the "effects" library to show the "Loading..." span tag. Similarly, `ClientSide.OnComplete` property value contains the javascript statement that hides the "Loading..." span tag. See TCallbackClientSide for further details on client-side property details.

12.2 TActiveCheckBox

System.Web.UI.ActiveControls.TActiveCheckBox API Reference

`TActiveCheckBox` is the active control counter part to TCheckbox. The `AutoPostBack` property of `TActiveCheckBox` is set to true by default. Thus, when the checkbox is clicked the `OnCallback` event is raise after the `OnCheckedChanged` event.

The `Text` and `Checked` properties of `TActiveCheckBox` can be changed during a callback request. The `TextAlign` property of `TActiveCheckBox` can not be changed during a

callback request.

ActiveControls.Samples.TActiveCheckBox.Home Demo

12.3 TActiveCustomValidator

System.Web.UI.ActiveControls.TActiveCustomValidator API Reference

Performs custom validation using only server-side `OnServerValidate` validation event. The client-side uses callbacks to raise `onServerValidate` event. The `ClientValidationFunction` property is disabled and will throw an exception if trying to set this property.

Beware that the `onServerValidate` may be raised when the control to validate on the client side changes value, that is, the server validation may be called many times.

After the callback or postback, the @link onServerValidate onServerValidate is raised once more. The `IsCallback` property of the `TPage` class will be true when validation is made during a callback request.

ActiveControls.Samples.TActiveCustomValidator.Home Demo

Chapter 13

Active Control Overview

13.1 Active Controls (AJAX enabled Controls)

See the Introduction for a quick overview of the concept behind active controls (AJAX enabled controls). Most active controls have a property of ActiveControl and a sub-property ClientSide that provides many properties to customize the controls. The CallbackClient property of the `TPage` class provides many methods to update and alter the client-side content during a callback request. Active controls is reliant on a collection of javascript classes.

For a quick demo of active controls, try the TActiveButton control. See also the later part of the Currency Converter tutorial for a more in depth example.

* the tutorial for this control is not completed yet.

13.1.1 Standard Active Controls

- TActiveButton represents a click button on a Web page. It can be used to trigger a callback request.
- TActiveCheckBox represents a checkbox on a Web page. It can be used to collect two-state user input and can trigger a callback request.
- TActiveCustomValidator validates a particular control using a callback request.

- **TActiveHyperLink represents a hyperlink on a Web page.**
- *** TActiveImage represents an image on a Web page.**
- *** TActiveImageButton represents a click button that has an image as the background. It is can be used to trigger a callback request.**
- *** TActiveLabel represents a label on a Web page. The label can be customized via various CSS attributes.**
- *** TActiveLinkButton represents a hyperlink that can perform a callback request.**
- *** TActivePanel represents a container for other controls on a Web page. In HTML, it is displayed as a ¡div¿ element. The panel's contents can be replaced during a callback request.**
- **TActivePager generates UI that allows users to interactively specify which page of data to be displayed in a data-bound control.**
- *** TActiveRadioButton represents a radiobutton on a Web page. It is mainly used in a group from which users make a choice. It can be used to perform a callback request.**
- *** TActiveTextBox represents a text input field on a Web page. It can collect single-line, multi-line or password text input from users. It can be used to perform a callback request.**
- *** TCallbackOptions callback options such as `OnLoading` client-side event handlers.**

13.1.2 Active List Controls

- *** TActiveCheckBoxList displays a list of checkboxes on a Web page and each checkbox can trigger a callback request.**
- *** TActiveDropDownList displays a dropdown list box that allows users to select a single option from a few prespecified ones. It can be used to perform a callback request.**
- *** TActiveListBox displays a list box that allows single or multiple selection. It can be used to perform a callback request.**

- * TActiveRadioButtonList is similar to TActiveCheckBoxList in every aspect except that each TActiveRadioButtonList displays a group of radiobuttons. Each radio button can perform a callback request.

13.1.3 Extended Active Controls

- TAutoComplete extends TActiveTextBox to offer text completion suggestions.
- * TCallback a generic control that can perform callback requests.
- * TEventTriggeredCallback triggers a callback request based on HTML DOM events.
- * TInPlaceTextBox represents a label that can be edited by clicked.
- * TTimeTriggeredCallback triggers a callback request based on time elapsed.
- * TValueTriggeredCallback monitors (using a timer) an attribute of an HTML element and triggers a callback request when the attribute value changes.
- TDropContainer TDraggable represents drag and drop containers. The former will make its child controls "draggable" while the latter will raise a callback when a draggable control is dropped on it.

13.1.4 Active Control Abilities

The following table shows the Active Controls that can trigger a callback event and whether the control will raise a PostBack event if Javascript was disabled on the client's browser.

13.1.5 Active Control Infrastructure Classes

The following classes provide the basic infrastructure classes required to realize the active controls.

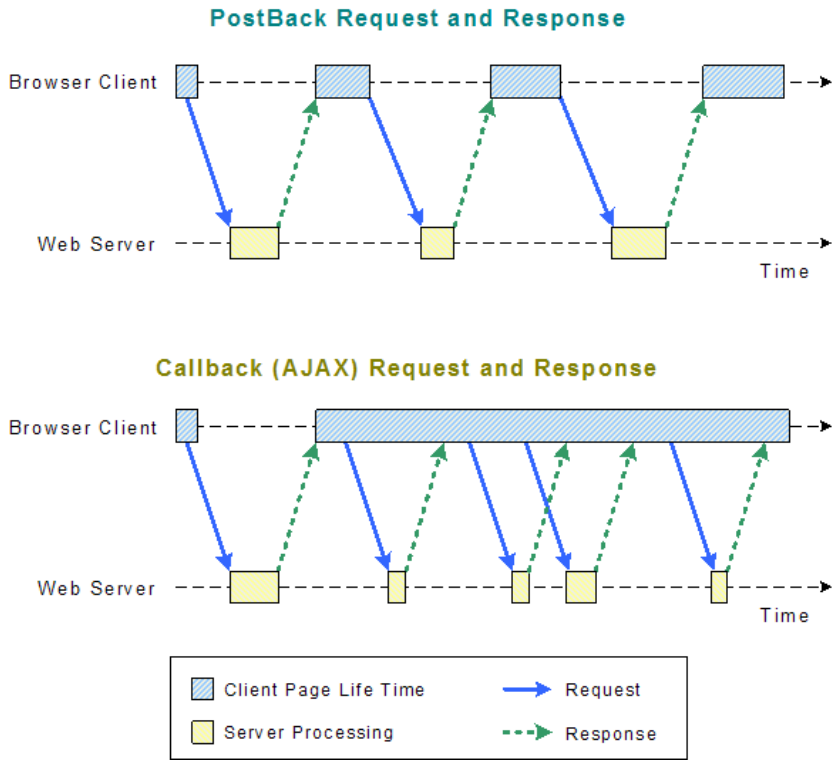
- * TActiveControlAdapter tracks the viewstate values of the control and update differences of the client-side HTML element attributes.

Control Name	Triggers Callback	Falls back to PostBack
TActiveButton	Yes	Yes
TActiveCheckBox	Yes	Yes
TActiveCustomValidator	Yes	Yes
TActiveHyperLink	No	Yes
TActiveImage	No	Yes
TActiveImageButton	Yes	Yes
TActiveLabel	No	Yes
TActiveLinkButton	Yes	No
TActivePanel	No	Yes
TActiveRadioButton	Yes	Yes
TActiveTextBox	Yes	Yes
TCallbackOptions	No	N/A
TActiveCheckBoxList	Yes	Yes
TActiveDropDownList	Yes	Yes
TActiveListBox	Yes	Yes
TActiveRadioButtonList	Yes	Yes
TAutoComplete	Yes	No
TCallback	Yes	No
TEventTriggeredCallback	Yes	No
TInPlaceTextBox	Yes	No
TTimeTriggeredCallback	Yes	No
TValueTriggeredCallback	Yes	No
TDropContainer	Yes	No
TDraggable	No	No

- * TActiveListControlAdapter allows the adapted list controls to change the selections on the client-side during a callback request.
- * TActivePageAdapter process the page life-cycle for callback requests.
- * TBaseActiveControl common active control methods and options.
- * TCallbackClientScript methods to manipulate the client-side HTML elements, also includes methods to invoke javascript Effects on HTML elements.
- * TCallbackClientSide common client-side callback request options, and client-side event handlers.
- * TCallbackResponseAdapter HTTP response for callback requests.

13.2 Overview of Active Controls

TODO:



Chapter 14

Write New Controls

14.1 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.

14.1.1 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 extend `TCompositeControl` and 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 components 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.

Controls.Samples.LabeledTextBox1.Home Demo

Composition by Overriding `createChildControls()`

For a composite control as simple as `LabeledTextBox`, it is better to create it by extending `TCompositeControl` and overriding the `createChildControls()` method, because

it does not use templates and thus saves template parsing time.

Complete code for `LabeledTextBox` is shown as follows,

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

Controls.Samples.LabeledTextBox2.Home Demo

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.

14.1.2 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 - Like pages, 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 overridden by derived classes as they usually have different attributes to be rendered.**
- **`renderBeginTag()` - renders the opening 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.

Chapter 15

Service References

15.1 SOAP Service

SOAP forms the foundation layer of the Web services stack. It provides a neat way for PHP applications to communicate with each other or with applications written in other languages. PRADO provides `TSoapService` that makes developing a SOAP server application an extremely easy task.

To use `TSoapService`, configure it in the application specification like following:

```
<services>
  <service id="soap" class="System.Web.Services.TSoapService">
    <soap id="stockquote" provider="path.to.StockQuote" />
    <!--
    <soap...other soap service... />
    -->
  </service>
</services>
```

The example specifies a SOAP service provider named `stockquote` which implements the `getPrice` SOAP method in the provider class `StockQuote`,

```
class StockQuote
```

```
{
    /**
     * @param string $symbol the symbol of the stock
     * @return float the stock price
     * @soapmethod
     */
    public function getPrice($symbol)
    {
        ....return stock price for $symbol
    }
}
```

> **Note:** `TSoapService` is based on **PHP SOAP extension** and thus requires the extension to be installed.

With the above simple code, we already finish a simple SOAP service that allows other applications to query the price of a specific stock. For example, a typical SOAP client may be written as follows to query the stock price of IBM,

```
$client=new SoapClient('http://path/to/index.php?soap=stockquote.wsdl');
echo $client->getPrice('IBM');
```

Notice the URL used to construct `SoapClient` (a class provided by PHP SOAP extension). This is the URL for the **WSDL** that describes the communication protocol for the SOAP service we just implemented. WSDL is often too complex to be manually written. Fortunately, `TSoapService` can generate this for us using a WSDL generator. In general, the URL for the automatically generated WSDL in PRADO has the following format:

```
http://path/to/index.php?SoapServiceID=SoapProviderID.wsdl
```

In order for the WSDL generator to generate WSDL for a SOAP service, the provider class needs to follow certain syntax. In particular, for methods to be exposed as SOAP methods, a keyword `@soapmethod` must appear in the phpdoc comment of the method with the following lines specifying method parameters and return value:

- **parameter:** `@param parameter-type $parameter-name description`

- **return value:** `@return value-type description`

Valid parameter and return types include: `string`, `int`, `boolean`, `float`, `array`, `mixed`, **etc. You may also specify a class name as the type, which translates into a complex SOAP type. For example, for a complex type** `Contact`

```
/**
 * Extends TComponent to provide property setter/getter methods
 */
class Contact {
    /**
     * @var string $name
     * @soapproperty
     */
    public $name;

    /**
     * @var Address $address
     * @soapproperty
     */
    private $_address;

    public function setAddress($value) {
        $this->_address=$value;
    }

    public function getAddress() {
        if($this->_address===null)
            $this->_address=new Address;
        return $this->_address;
    }
}

class Address{
    /**
     * @var string $city
     * @soapproperty
     */
    public $city;
```

```
}

class ContactManager {
    /**
     * @return Contact[] an array of contacts
     * @soapmethod
     */
    function getAllContacts() {
        return array(new Contact);
    }

    /**
     * @return Contact one contact
     * @soapmethod
     */
    function getContact($name) {
        return new Contact;
    }
}
```

For a complex soap object, the properties of the object are specified with `@soapproperty` keyword in the property phpdocs. Furthermore, the propert's type name must be specified as `@var type $name` where `type` is any valid type in mentioned earlier and `$name` will defined a property `name` (notice that if your class is a TComponent, you can provide property setter/getter methods).

An array of complex objects can also be returned by adding a pair of enclosing square brackets after the type name. For example, to return an array of `Contact` type, we define `@return Contact[] ...`.

> **Tip: A very useful tool to test out your web services is the free tool WebServiceStudio 2.0. It can invoke webmethods interactively. The user can provide a WSDL endpoint. On clicking button Get the tool fetches the WSDL, generates .NET proxy from the WSDL and displays the list of methods available. The user can choose any method and provide the required input parameters. The tool requires a MS .NET runtime to be installed.**
> **A similar tool is available for Mac OS X Tiger from http://www.ditchnet.org/soapclient/**

TSoapService may be configured and customized in several ways. In the example above, the <soap> element actually specifies a SOAP service using the default TSoapServer implementation. Attributes in <soap> are passed to TSoapServer as its initial property values. For example, the provider attribute initializes the Provider property of TSoapServer. By setting SessionPersistent to be true in <soap> element, the provider instance will persist within the user session. You may develop your own SOAP server class and use it by specifying the class attribute of <soap>.

By default, PHP's soap server will create objects of the type StdClass when objects are received from the client. The soap server can be configured to automatically create objects of certain type objects are received as method parameters. For example, if we have a Soap method that accepts a Contact object as parameter.

```
/**
 * @param Contact $contact
 * @return boolean true if saved, false otherwise
 * @soapmethod
 */
 function save(Contact $contact)
 {
     return true
 }
```

The do this, we need to set the ClassMaps property of the TSoapServer in the <soap> tags as a comma separated string of class names that we wish to be automatically converted.

```
<soap id="contact-manager" provider="path.to.ContactManager"
     ClassMaps="Contact, Address"/>
```

Chapter 16

Working with Databases

16.1 Data Access Objects (DAO)

Data Access Objects (DAO) separates a data resource's client interface from its data access mechanisms. It adapts a specific data resource's access API to a generic client interface. As a result, data access mechanisms can be changed independently of the code that uses the data.

Since version 3.1, PRADO starts to provide a DAO that is a thin wrap around PHP Data Objects (PDO). Although PDO has a nice feature set and good APIs, we choose to implement the PRADO DAO on top of PDO because the PRADO DAO classes are component classes and are thus configurable in a PRADO application. Users can use these DAO classes in a more PRADO-preferred way.

> **Note: Since the PRADO DAO is based on PDO, the PDO PHP extension needs to be installed. In addition, you need to install the corresponding PDO driver for the database to be used in your application. See more details in the PHP Manual.**

The PRADO DAO mainly consists of the following four classes (in contrast to PDO which uses only two classes, `PDO` and `PDOStatement`):

- `TDbConnection` **- represents a connection to a database.**

- `TDbCommand` - represents an SQL statement to execute against a database.
- `TDbDataReader` - represents a forward-only stream of rows from a query result set.
- `TDbTransaction` - represents a DB transaction.

In the following, we introduce the usage of PRADO DAO in different scenarios.

16.1.1 Establishing Database Connection

To establish a database connection, one creates a `TDbConnection` instance and activate it. A data source name (DSN) is needed to specify the information required to connect to the database. The database username and password may need to be supplied to establish the connection.

```
$connection=new TDbConnection($dsn,$username,$password);
// call setAttribute() to pass in additional connection parameters
// $connection->Persistent=true;  // use persistent connection
$connection->Active=true;  // connection is established
....
$connection->Active=false;  // connection is closed
```

Complete specification of DSN may be found in the PDO documentation. Below is a list of commonly used DSN formats:

- **MySQL** - `mysql:host=localhost;dbname=test`
- **SQLite** - `sqlite:/path/to/dbfile`
- **ODBC** - `odbc:SAMPLE`

In case any error occurs when establishing the connection (such as bad DSN or username/password), a `TDbException` will be raised.

16.1.2 Executing SQL Statements

Once a database connection is established, SQL statements can be executed through `TDbCommand`. One creates a `TDbCommand` by calling `TDbConnection.createCommand()` with

the specified SQL statement:

```
$command=$connection->createCommand($sqlStatement);
// if needed, the SQL statement may be updated as follows:
$command->Text=$newSqlStatement;
```

An SQL statement is executed via `TDbCommand` in one of the following two ways:

- `execute()` **- performs a non-query SQL statement, such as `INSERT`, `UPDATE` and `DELETE`. If successful, it returns the number of rows that are affected by the execution.**
- `query()` **- performs an SQL statement that returns rows of data, such as `SELECT`. If successful, it returns a `TDbDataReader` instance from which one can fetch the resulting rows of data.**

```
$affectedRowCount=$command->execute();  // execute the non-query SQL
$dataReader=$command->query();          // execute a query SQL
$row=$command->queryRow();              // execute a query SQL and return the first row of result
$value=$command->queryScalar();         // execute a query SQL and return the first column value
```

In case an error occurs during the execution of SQL statements, a `TDbException` will be raised.

16.1.3 Fetching Query Results

After `TDbCommand.query()` generates the `TDbDataReader` instance, one can retrieve rows of resulting data by calling `TDbDataReader.read()` repeatedly. One can also use `TDbDataReader` in PHP's `foreach` language construct to retrieve row by row.

```
// calling read() repeatedly until it returns false
while(($row=$dataReader->read())!==false) { ... }
// using foreach to traverse through every row of data
foreach($dataReader as $row) { ... }
// retrieving all rows at once in a single array
$rows=$dataReader->readAll();
```

16.1.4 Using Transactions

When an application executes a few queries, each reading and/or writing information in the database, it is important to be sure that the database is not left with only some of the queries carried out. A transaction, represented as a `TDbTransaction` instance in PRADO, may be initiated in this case:

- **Begin the transaction.**
- **Execute queries one by one. Any updates to the database are not visible to the outside world.**
- **Commit the transaction. Updates become visible if the transaction is successful.**
- **If one of the queries fails, the entire transaction is rolled back.**

```
$transaction=$connection->beginTransaction();
try
{
    $connection->createCommand($sql1)->execute();
    $connection->createCommand($sql2)->execute();
    //.... other SQL executions
    $transaction->commit();
}
catch(Exception $e) // an exception is raised if a query fails will be raised
{
    $transaction->rollBack();
}
```

16.1.5 Binding Parameters

To avoid SQL injection attacks and to improve performance of executing repeatedly used SQL statements, one can "prepare" an SQL statement with optional parameter placeholders that are to be replaced with the actual parameters during the parameter binding process.

The parameter placeholders can be either named (represented as unique tokens) or unnamed (represented as question marks). Call `TDbCommand.bindParameter()` or

`TDbCommand.bindValue()` to replace these placeholders with the actual parameters. The parameters do not need to be quoted: the underlying database driver does it for you. Parameter binding must be done before the SQL statement is executed.

```
// an SQL with two placeholders ":username" and ":email"
$sql="INSERT INTO users(username, email) VALUES(:username,:email)";
$command=$connection->createCommand($sql);
// replace the placeholder ":username" with the actual username value
$command->bindParameter(":username",$username,PDO::PARAM_STR);
// replace the placeholder ":email" with the actual email value
$command->bindParameter(":email",$email,PDO::PARAM_STR);
$command->execute();
// insert another row with a new set of parameters
$command->bindParameter(":username",$username2,PDO::PARAM_STR);
$command->bindParameter(":email",$email2,PDO::PARAM_STR);
$command->execute();
```

The methods `bindParameter()` and `bindValue()` are very similar. The only difference is that the former binds a parameter with a PHP variable reference while the latter with a value. For parameters that represent large block of data memory, the former is preferred for performance consideration.

For more details about binding parameters, see the relevant PHP documentation.

16.1.6 Binding Columns

When fetching query results, one can also bind columns with PHP variables so that they are automatically populated with the latest data each time a row is fetched.

```
$sql="SELECT username, email FROM users";
$dataReader=$connection->createCommand($sql)->query();
// bind the 1st column (username) with the $username variable
$dataReader->bindColumn(1,$username);
// bind the 2nd column (email) with the $email variable
$dataReader->bindColumn(2,$email);
while($dataReader->read()!==false)
{
    // $username and $email contain the username and email in the current row
```

```
}
```

16.2 Active Record

Active Records are objects that wrap a row in a database table or view, encapsulate the database access and add domain logic on that data. The basics of an Active Record are business classes, e.g., a `Products` class, that match very closely the record structure of an underlying database table. Each Active Record will be responsible for saving and loading data to and from the database.

> **Info: The data structure of an Active Record should match that of a table in the database. Each column of a table should have a corresponding member variable or property in the Active Record class the represents the table.**

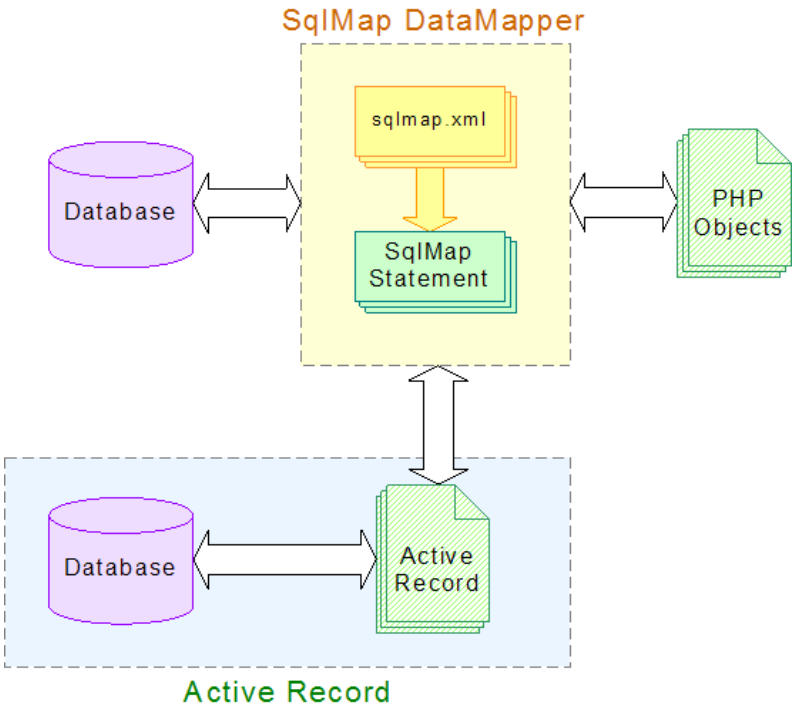
16.2.1 When to Use It

Active Record is a good choice for domain logic that isn't too complex, such as creates, reads, updates, and deletes. Derivations and validations based on a single record work well in this structure. Active Record has the primary advantage of simplicity. It's easy to build Active Records, and they are easy to understand.

However, as your business logic grows in complexity, you'll soon want to use your object's direct relationships, collections, inheritance, and so forth. These don't map easily onto Active Record, and adding them piecemeal gets very messy. Another argument against Active Record is the fact that it couples the object design to the database design. This makes it more difficult to refactor as a project goes forward.

The alternative is to use a Data Mapper that separates the roles of the business object and how these objects are stored. Prado provides a complimentary choice between Active Record and SqlMap Data Mapper. A SqlMap Data Mapper can be used to load Active Record objects, in turn; these Active Record objects can be used to update the database. The "relationship" between Active Records and SqlMap is illustrated in the following diagram. More details regarding the SqlMap Data Mapper can be found in the SqlMap Manual.

The Active Record class has functionality to perform the following tasks.



- **Create, Retrieve, Update and Delete records.**
- **Finder methods to wrap commonly used SQL queries and return Active Record objects.**
- **Fetch relationships (related foreign objects) such as "has many", "has one", "belongs to" and "many to many" via association table.**
- **Lazy loading of relationships.**

16.2.2 Design Implications

Prado's implementation of Active Record does not maintain referential identity. Each object obtained using Active Record is a copy of the data in the database. For example, If you ask for a particular customer and get back a `Customer` object, the next time you ask for that customer you get back another instance of a `Customer` object. This implies that a strict comparison (i.e., using ===) will return false, while loose comparison (i.e., using ==) will return true if the object values are equal by loose

comparison.

This is design implication related to the following question. ¡i¿"Do you think of the customer as an object, of which there's only one, or do you think of the objects you operate on as copies of the database?"¡/i¿ Other O/R mappings will imply that there is only one Customer object with custID 100, and it literally is that customer. If you get the customer and change a field on it, then you have now changed that customer. ¡i¿"That constrasts with: you have changed this copy of the customer, but not that copy. And if two people update the customer on two copies of the object, whoever updates first, or maybe last, wins."¡/i¿ [A. Hejlsberg 2003]

16.2.3 Database Supported

The Active Record implementation utilizes the Prado DAO classes for data access. The current Active Record implementation supports the following database.

- MySQL 4.1 or later
- Postgres SQL 7.3 or later
- SQLite 2 and 3
- MS SQL 2000 or later
- Oracle Database (alpha)

Support for other databases can be provided when there are sufficient demands.

16.3 Defining an Active Record

Let us consider the following "`users`" table that contains two columns named "`username`" and "`email`", where "`username`" is also the primary key.

```
CREATE TABLE users
(
    username VARCHAR( 20 ) NOT NULL ,
    email VARCHAR( 200 ) ,
```

```
    PRIMARY KEY ( username )
);
```

Next we define our Active Record class that corresponds to the "users" table.

```
class UserRecord extends TActiveRecord
{
    const TABLE='users'; //table name

    public $username; //the column named "username" in the "users" table
    public $email;

    /**
     * @return TActiveRecord active record finder instance
     */
    public static function finder($className=__CLASS__)
    {
        return parent::finder($className);
    }
}
```

Each column of the "users" table must have corresponding property of the same name as the column name in the UserRecord class. Of course, you also define additional member variables or properties that does not exist in the table structure. The class constant TABLE is optional when the class name is the same as the table name in the database, otherwise TABLE must specify the table name that corresponds to your Active Record class.

> **Tip: You may specify qualified table names. E.g. for MySQL,** TABLE = ```` ```database1`.`table1` ````".

> **Note: Since version 3.1.3 you can also use a method `table()` to define the table name. This allows you to dynamically specify which table should be used by the ActiveRecord.**
>
> ```
> class TeamRecord extends TActiveRecord
> {
>     public function table() {
>         return 'Teams';
>     }
> }
> ```

Since `TActiveRecord` extends `TComponent`, setter and getter methods can be defined to allow control over how variables are set and returned. For example, adding a `$level` property to the UserRecord class:

```
class UserRecord extends TActiveRecord {
    ... //existing definitions as above

    private $_level;
    public function setLevel($value) {
        $this->_level=TPropertyValue::ensureInteger($value,0);
    }
    public function getLevel($value){
        return $this->_level;
    }
}
```

More details regarding TComponent can be found in the Components documentation. Later we shall use the getter/setters to allow for lazy loading of relationship objects.

> **Info: `TActiveRecord` can also work with database views by specifying the constant `TABLE` corresponding to the view name. However, objects returned from views are read-only, calling the `save()` or `delete()` method will raise an exception.**

The static method `finder()` returns an `UserRecord` instance that can be used to load records from the database. The loading of records using the finder methods is discussed a little later. The `TActiveRecord::finder()` static method takes the name of an Active Record class as parameter.

16.3.1 Setting up a database connection

A default database connection for Active Record can be set as follows. See Establishing Database Connection for further details regarding creation of database connection in general.

```
//create a connection and give it to the Active Record manager.
$dsn = 'pgsql:host=localhost;dbname=test'; //Postgres SQL
$conn = new TDbConnection($dsn, 'dbuser','dbpass');
TActiveRecordManager::getInstance()->setDbConnection($conn);
```

Alternatively, you can create a base class and override the `getDbConnection()` method to return a database connection. This is a simple way to permit multiple connections and multiple databases. The following code demonstrates defining the database connection in a base class (not need to set the DB connection anywhere else).

```
class MyDb1Record extends TActiveRecord
{
    public function getDbConnection()
    {
        static $conn;
        if($conn===null)
            $conn = new TDbConnection('xxx','yyy','zzz');
        return $conn;
    }
}
class MyDb2Record extends TActiveRecord
{
    public function getDbConnection()
    {
        static $conn;
        if($conn===null)
            $conn = new TDbConnection('aaa','bbb','ccc');
        return $conn;
    }
}
```

¡h3 class="prado-specific"¿Using `application.xml` within the Prado Framework¡/h3¿ ¡div class="prado-specific"¿

The default database connection can also be configured using a `<module>` tag in the application.xml or config.xml as follows.

```
<modules>
  <module class="System.Data.ActiveRecord.TActiveRecordConfig" EnableCache="true">
    <database ConnectionString="pgsql:host=localhost;dbname=test"
        Username="dbuser" Password="dbpass" />
  </module>
</modules>
```

> **Tip: The `EnableCache` attribute when set to "true" will cache the table meta data, that is, the table columns names, indexes and constraints are saved in the cache and reused. You must clear or disable the cache if you wish to see changes made to your table definitions. A cache module must also be defined for the cache to function.**

A `ConnectionID` property can be specified with value corresponding to another `TDataSourceConfig` module configuration's ID value. This allows the same database connection to be used in other modules such as SqlMap.

```
<modules>
  <module class="System.Data.TDataSourceConfig" id="db1">
    <database ConnectionString="pgsql:host=localhost;dbname=test"
        Username="dbuser" Password="dbpass" />
  </module>

  <module class="System.Data.ActiveRecord.TActiveRecordConfig"
        ConnectionID="db1" EnableCache="true"  />

  <module class="System.Data.SqlMap.TSqlMapConfig"
        ConnectionID="db1"  ... />
</modules>
```

¡/div¿

16.3.2 Loading data from the database

The `TActiveRecord` class provides many convenient methods to find records from the database. The simplest is finding one record by matching a primary key or a composite key (primary keys that consists of multiple columns). See the System.Data.ActiveRecord.TActiveRecord API Reference for more details.

> **Info: All finder methods that may return 1 record only will return `null` if no matching data is found. All finder methods that return an array of records will return an empty array if no matching data is found.**

`findByPk()`

Finds one record using only a primary key or a composite key.

```
$finder = UserRecord::finder();
$user = $finder->findByPk($primaryKey);

//when the table uses a composite key
$record = $finder->findByPk($key1, $key2, ...);
$record = $finder->findByPk(array($key1, $key2,...));
```

`findAllByPks()`

Finds multiple records using a list of primary keys or composite keys. The following are equivalent for primary keys (primary key consisting of only one column/field).

```
$finder = UserRecord::finder();
$users = $finder->findAllByPks($key1, $key2, ...);
$users = $finder->findAllByPks(array($key1, $key2, ...));
```

The following are equivalent for composite keys.

```
//when the table uses composite keys
$record = $finder->findAllByPks(array($key1, $key2), array($key3, $key4), ...);
```

```
$keys = array(  array($key1, $key2), array($key3, $key4), ... );
$record = $finder->findAllByPks($keys);
```

`find()`

Finds one single record that matches the criteria. The criteria can be a partial SQL string or a `TActiveRecordCriteria` object.

```
$finder = UserRecord::finder();

//:name and :pass are place holders for specific values of $name and $pass
$finder->find('username = :name AND password = :pass',
                          array(':name'=>$name, ':pass'=>$pass));

//using position place holders
$finder->find('username = ? AND password = ?', array($name, $pass));
//same as above
$finder->find('username = ? AND password = ?', $name, $pass);

//$criteria is of TActiveRecordCriteria
$finder->find($criteria); //the 2nd parameter for find() is ignored.
```

The `TActiveRecordCriteria` class has the following properties:

- `Parameters` – **name value parameter pairs.**
- `OrdersBy` – **column name and ordering pairs.**
- `Condition` – **parts of the WHERE SQL conditions.**
- `Limit` – **maximum number of records to return.**
- `Offset` – **record offset in the table.**

```
$criteria = new TActiveRecordCriteria;
$criteria->Condition = 'username = :name AND password = :pass';
$criteria->Parameters[':name'] = 'admin';
$criteria->Parameters[':pass'] = 'prado';
$criteria->OrdersBy['level'] = 'desc';
```

```
$criteria->OrdersBy['name'] = 'asc';
$criteria->Limit = 10;
$criteria->Offset = 20;
```

> **Note: For MSSQL and when `Limit` and `Offset` are positive integer values. The actual query to be executed is modified by the TMssqlCommandBuilder class according to http://troels.arvin.dk/db/rdbms/ to emulate the `Limit` and `Offset` conditions.**

`findAll()`

Same as `find()` but returns an array of objects.

`findBy*()` and `findAllBy*()`

Dynamic find method using parts of the method name as search criteria. Method names starting with `findBy` return 1 record only and method names starting with `findAllBy` return an array of records. The condition is taken as part of the method name after `findBy` or `findAllBy`.

The following blocks of code are equivalent:

```
$finder->findByName($name)
$finder->find('Name = ?', $name);
```

```
$finder->findByUsernameAndPassword($name,$pass);
$finder->findBy_Username_And_Password($name,$pass);
$finder->find('Username = ? AND Password = ?', $name, $pass);
```

```
$finder->findAllByAge($age);
$finder->findAll('Age = ?', $age);
```

> **Tip: You may also use a combination of `AND` and `OR` as a condition in the dynamic methods.**

`findBySql()` and `findAllBySql()`

Finds records using full SQL where `findBySql()` return an Active Record and `findAllBySql()`returns an array of record objects. For each column returned, the corresponding Active Record class must define a member variable or property for each corresponding column name.

```
class UserRecord2 extends UserRecord
{
    public $another_value;
}
$sql = "SELECT users.*, 'hello' as another_value FROM users";
$users = TActiveRecord::finder('UserRecord2')->findAllBySql($sql);
```

`count()`

Find the number of matchings records, accepts same parameters as the `findAll()` method.

16.3.3 Inserting and updating records

Add a new record using TActiveRecord is very simple, just create a new Active Record object and call the `save()` method. E.g.

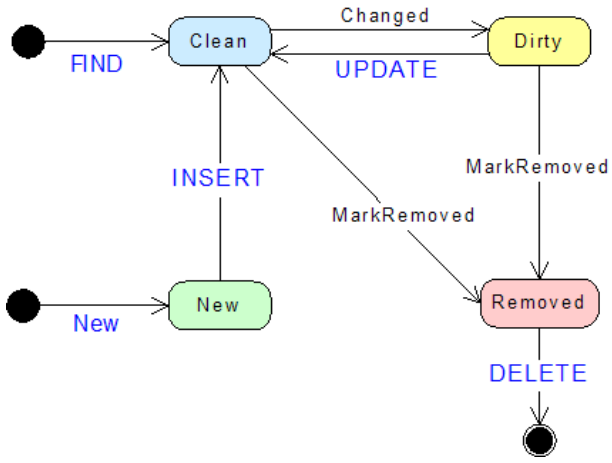
```
$user1 = new UserRecord();
$user1->username = "admin";
$user1->email = "admin@example.com";
$user1->save(); //insert a new record

$data = array('username'=>'admin', 'email'=>'admin@example.com');
$user2 = new UserRecord($data); //create by passing some existing data
$user2->save(); //insert a new record
```

> **Tip: If you insert a new record into a MySQL table that has columns defined with "autoincrement", the Active Record objects will be updated with the new incremented value.**

To update a record in the database, just change one or more properties of the Active Record object that has been loaded from the database and then call the `save()` method.

```
$user = UserRecord::finder()->findByName('admin');
$user->email="test@example.com"; //change property
$user->save(); //update it.
```

Active Record objects have a simple life-cycle illustrated in the following diagram.



We see that new TActiveRecord objects are created by either using one of the `find*()` methods or using creating a new instance by using PHP's `new` keyword. Objects created by a `find*()` method starts with `clean` state. New instance of TActiveRecord created other than by a `find*()` method starts with `new` state. Whenever you call the `save()` method on the TActiveRecord object, the object enters the `clean` state. Objects in the `clean` becomes `dirty` whenever one of more of its internal states are changed. Calling the `delete()` method on the object ends the object life-cycle, no further actions can be performed on the object.

16.3.4 Deleting existing records

To delete an existing record that is already loaded, just call the `delete()` method. You can also delete records in the database by primary keys without loading any records

using the `deleteByPk()` method (and equivalently the `deleteAllByPks()` method). For example, to delete one or several records with tables using one or more primary keys.

```
$finder->deleteByPk($primaryKey); //delete 1 record
$finder->deleteAllByPks($key1,$key2,...); //delete multiple records
$finder->deleteAllByPks(array($key1,$key2,...)); //delete multiple records
```

For composite keys (determined automatically from the table definitions):

```
$finder->deleteByPk(array($key1,$key2)); //delete 1 record

//delete multiple records
$finder->deleteAllByPks(array($key1,$key2), array($key3,$key4),...);

//delete multiple records
$finder->deleteAllByPks(array( array($key1,$key2), array($key3,$key4), .. ));
```

`deleteAll()` **and** `deleteBy*()`

To delete by a criteria, use `deleteAll($criteria)` and `deleteBy*()` with similar syntax to `findAll($criteria)` and `findAllBy*()` as described above.

```
//delete all records with matching Name
$finder->deleteAll('Name = ?', $name);
$finder->deleteByName($name);

//delete by username and password
$finder->deleteBy_Username_And_Password($name,$pass);
```

16.3.5 Transactions

All Active Record objects contain the property `DbConnection` that can be used to obtain a transaction object.

```
$finder = UserRecord::finder();
$finder->DbConnection->Active=true; //open if necessary
```

```
$transaction = $finder->DbConnection->beginTransaction();
try
{
    $user = $finder->findByPk('admin');
    $user->email = 'test@example.com'; //alter the $user object
    $user->save();
    $transaction->commit();
}
catch(Exception $e) // an exception is raised if a query fails
{
    $transaction->rollBack();
}
```

16.3.6 Events

The TActiveRecord offers two events, `OnCreateCommand` and `OnExecuteCommand`.

The `OnCreateCommand` event is raised when a command is prepared and parameter binding is completed. The parameter object is `TDataGatewayEventParameter` of which the `Command` property can be inspected to obtain the SQL query to be executed.

The `OnExecuteCommand` event is raised when a command is executed and the result from the database was returned. The parameter object is `TDataGatewayResultEventParameter` of which the `Result` property contains the data return from the database. The data returned can be changed by setting the `Result` property.

Logging Example

Using the `OnExecuteCommand` we can attach an event handler to log the entire SQL query executed for a given TActiveRecord class or instance. For example, we define a base class and override either the `getDbConnection()` or the constructor.

```
class MyDb1Record extends TActiveRecord
{
    public function getDbConnection()
    {
        static $conn;
```

```
        if($conn===null)
        {
            $conn = new TDbConnection('xxx','yyy','zzz');
            $this->OnExecuteCommand[] = array($this,'logger');
        }
        return $conn;
    }
    public function logger($sender,$param)
    {
        var_dump($param->Command->Text);
    }
}
//alternatively as per instance of per finder object
function logger($sender,$param)
{
    var_dump($param->Command->Text);
}
TActiveRecord::finder('MyRecord')->OnExecuteCommand[] = 'logger';
$obj->OnExecuteCommand[] = array($logger, 'log'); //any valid PHP callback.
```

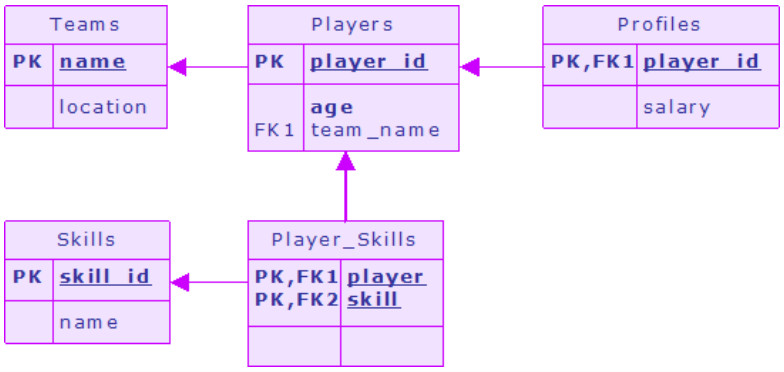
16.4 Active Record Relationships

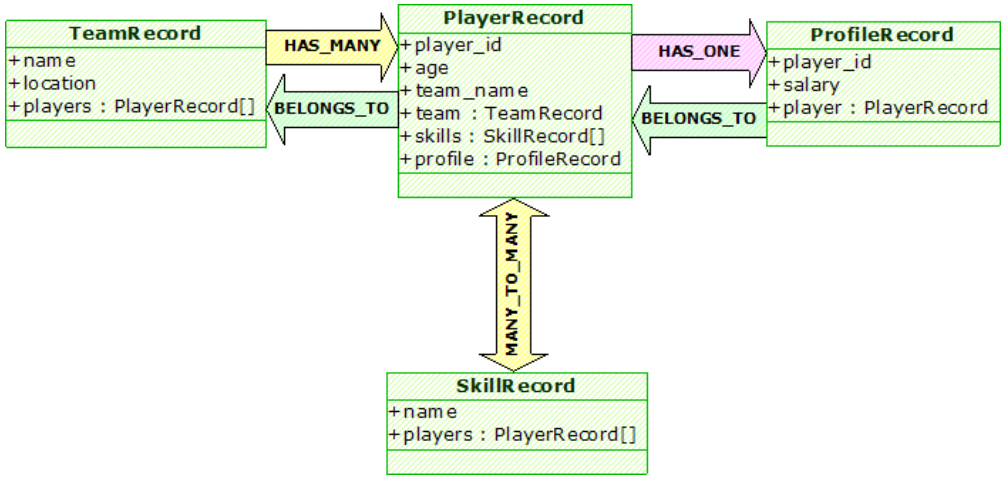
The Prado Active Record implementation supports the foreign key mappings for database that supports foreign key constraints. For Active Record relationships to function the underlying database must support foreign key constraints (e.g. MySQL using InnoDB).

In the following sections we will consider the following table relationships between `Teams`, `Players`, `Skills` **and** `Profiles`.

The goal is to obtain object models that represent to some degree the entity relationships in the above figure.

There is a mismatch between relationships with objects and table relationships. First there's a difference in representation. Objects handle links by storing references that are held by the runtime memory-managed environment. Relational databases handle links by forming a key into another table. Second, objects can easily use collections to handle multiple references from a single field, while normalization forces all entity





relation links to be single valued. This leads to reversals of the data structure between objects and tables. The approach taken in the Prado Active Record design is to use the table foreign key constraints to derive object relationships. This implies that the underlying database must support foreign key constraints.

Tip: For SQLite database, you may create tables that defines the foreign key constraints such as the example below. However, these constraints are NOT enforced by the SQLite database itself.

```
CREATE TABLE foo
(
    id INTEGER NOT NULL PRIMARY KEY,
    id2 CHAR(2)
);
CREATE TABLE bar
(
    id INTEGER NOT NULL PRIMARY KEY,
    foo_id INTEGER
        CONSTRAINT fk_foo_id REFERENCES foo(id) ON DELETE CASCADE
);
```

16.4.1 Foreign Key Mapping

The entity relationship between the `Teams` and `Players` table is what is known as an 1-M relationship. That is, one Team may contain 0 or more Players. In terms of object relationships, we say that a `TeamRecord` object has many `PlayerRecord` objects. (Notice the reversal of the direction of relationships between tables and objects.)

Has Many Relationship

We model the `Team` object as the following Active Record classes.

```
class TeamRecord extends TActiveRecord
{
    const TABLE='Teams';
    public $name;
    public $location;

    public $players=array();  // this declaration is no longer needed since v3.1.2

    //define the $player member having has many relationship with PlayerRecord
    public static $RELATIONS=array
    (
```

```
        'players' => array(self::HAS_MANY, 'PlayerRecord', 'team_name'),
    );

    public static function finder($className=__CLASS__)
    {
        return parent::finder($className);
    }
}
```

The static `$RELATIONS` property of `TeamRecord` defines that the property `$players` has many `PlayerRecords`. Multiple relationships is permitted by defining each relationship with an entry in the `$RELATIONS` array where array key for the entry corresponds to the property name. In `array(self::HAS_MANY, 'PlayerRecord')`, the first element defines the relationship type, the valid types are `self::HAS_MANY`, `self::HAS_ONE`, `self::BELONGS_TO` and `self::MANY_TO_MANY`. The second element is a string `'PlayerRecord'` that corresponds to the class name of the `PlayerRecord` class. And the third element 'team$_{name}$** refers to the foreign key column in the Players table that references to the Teams table.**

> **Note: As described in the code comment above, since version 3.1.2, related properties no longer need to be explicitly declared. By default, they will be implicitly declared according to keys of the `$RELATIONS` array. A major benefit of declared related properties implicitly is that related objects can be automatically loaded in a lazy way. For example, assume we have a `TeamRecord` instance `$team`. We can access the players via `$team->players`, even if we have never issued fetch command for players. If `$players` is explicitly declared, we will have to use the `with` approach described in the following to fetch the player records.**

The foreign key constraint of the `Players` table is used to determine the corresponding `Teams` table's corresponding key names. This is done automatically handled in Active Record by inspecting the `Players` and `Teams` table definitions.

Info: Since version 3.1.2, Active Record supports multiple foreign key references of the same table. Ambiguity between multiple foreign key references to the same table is resolved by providing the foreign key column name as the 3rd parameter in the relationship array. For example, both of the following foreign keys `owner_id` and `reporter_id` references to the same table defined in `UserRecord`.

```
class TicketRecord extends TActiveRecord
{
     public $owner_id;
      public $reporter_id;

     public $owner;     // this declaration is no longer needed since v3.1.2
      public $reporter;  // this declaration is no longer needed since v3.1.2

      public static $RELATION=array
      (
          'owner' => array(self::BELONGS_TO, 'UserRecord', 'owner_id'),
           'reporter' => array(self::BELONGS_TO, 'UserRecord', 'reporter_id'),
      );
}
```

This is applicable to relationships including `BELONGS_TO`, `HAS_ONE` and `HAS_MANY`. See section Self Referenced Association Tables for solving ambiguity of `MANY_TO_MANY` relationships.

The "has many" relationship is not fetched automatically when you use any of the Active Record finder methods. You will need to explicitly fetch the related objects as follows. In the code below, both lines are equivalent and the method names are case insensitive.

```
$team = TeamRecord::finder()->withPlayers()->findAll();
$team = TeamRecord::finder()->with_players()->findAll(); //equivalent
```

The method `with_xxx()` (where `xxx` is the relationship property name, in this case, `players`) fetches the corresponding PlayerRecords using a second query (not by using a join). The `with_xxx()` accepts the same arguments as other finder methods of TActiveRecord, e.g. `with_players('age = ?', 35)`.

> **Note: It is essential to understand that the related objects are fetched using additional queries. The first query fetches the source object, e.g. the `TeamRecord` in the above example code. A second query is used to fetch the corresponding related `PlayerRecord` objects. The usage of the two query is similar to a single query using Left-Outer join with the exception that null results on the right table are not returned. The consequence of using two or more queries is that the aggregates and other join conditions are not feasible using Active Records. For queries outside the scope of Active Record the SqlMap Data Mapper may be considered.**

> **Info: The above `with` approach also works with implicitly declared related properties (introduced in version 3.1.2). So what is the difference between the `with` approach and the lazy loading approach? Lazy loading means we issue an SQL query if a related object is initially accessed and not ready, while the `with` approach queries for the related objects once for all, no matter the related objects are accessed or not. The lazy loading approach is very convenient since we do not need to explictly load the related objects, while the `with` approach is more efficient if multiple records are returned, each with some related objects.**

Has One Relationship

The entity relationship between `Players` and `Profiles` is one to one. That is, each `PlayerRecord` object has one `ProfileRecord` object (may be none or null). A has one relationship is nearly identical to a has many relationship with the exception that the related object is only one object (not a collection of objects).

Belongs To Relationship

The "has many" relationship in the above section defines a collection of foreign objects. In particular, we have that a `TeamRecord` has many (zero or more) `PlayerRecord` objects. We can also add a back pointer by adding a property in the `PlayerRecord` class that links back to the `TeamRecord` object, effectively making the association bidirectional. We say that the `$team` property in `PlayerRecord` class `belongs to` a `TeamRecord` object. The following code defines the complete `PlayerRecord` class with 3 relationships.

```
class PlayerRecord extends TActiveRecord
{
    const TABLE='Players';
    public $player_id;
    public $age;
    public $team_name;

    public $team;              // this declaration is no longer needed since v3.1.2
    public $skills=array();    // this declaration is no longer needed since v3.1.2
    public $profile;           // this declaration is no longer needed since v3.1.2

    public static $RELATIONS=array
    (
        'team' => array(self::BELONGS_TO, 'TeamRecord', 'team_name'),
        'skills' => array(self::MANY_TO_MANY, 'SkillRecord', 'Player_Skills'),
        'profile' => array(self::HAS_ONE, 'ProfileRecord', 'player_id'),
    );

    public static function finder($className=__CLASS__)
    {
        return parent::finder($className);
    }
}
```

The static `$RELATIONS` property of `PlayerRecord` defines that the property `$team` belongs to a `TeamRecord`. The `$RELATIONS` array also defines two other relationships that we shall examine in later sections below. In `array(self::BELONGS_TO, 'TeamRecord', 'team_name')`, the first element defines the relationship type, in this case ¡strong¿`self::BELONGS_TO`¡/strong the second element is a string `'TeamRecord'` that corresponds to the class name of the `TeamRecord` class; and the third element 'team$_{n}ame'referstotheforeignkeyofPlayersreferencingTeams.Aplaye

```
$players = PlayerRecord::finder()->with_team()->findAll();
```

The method `with_xxx()` (where `xxx` is the relationship property name, in this case, `team`) fetches the corresponding `TeamRecords` using a second query (not by using a join). The `with_xxx()` accepts the same arguments as other finder methods of `TActiveRecord`, e.g. `with_team('location = ?', 'Madrid')`.

> **Tip: Additional relationships may be fetched by chaining the `with_xxx()` together as the following example demonstrates.**
> `$players = PlayerRecord::finder()->with_team()->with_skills()->findAll();`
> **Each `with_xxx()` method will execute an additional SQL query. Every `with_xxx()` accepts arguments similar to those in the `findAll()` method and is only applied to that particular relationship query.**

The "belongs to" relationship of `ProfileRecord` class is defined similarly.

```
class ProfileRecord extends TActiveRecord
{
    const TABLE='Profiles';
    public $player_id;
    public $salary;

    public $player;  // this declaration is no longer needed since v3.1.2

    public static $RELATIONS=array
    (
        'player' => array(self::BELONGS_TO, 'PlayerRecord'),
    );

    public static function finder($className=__CLASS__)
    {
        return parent::finder($className);
    }
}
```

In essence, there exists a "belongs to" relationship for objects corresponding to entities that has column which are foreign keys. In particular, we see that the `Profiles` table has a foreign key constraint on the column `player_id` that relates to the `Players` table's `player_id` column. Thus, the `ProfileRecord` object has a property (`$player`) that belongs to a `PlayerRecord` object. Similarly, the `Players` table has a foreign key constraint on the column `team_name` that relates to the `Teams` table's `name` column. Thus, the `PlayerRecord` object has a property (`$team`) that belongs to a `TeamRecord` object.

Parent Child Relationships

A parent child relationship can be defined using a combination of `has many` and `belongs to` relationship that refers to the same class. The following example shows a parent children relationship between "categories" and a "parent category".

```
class Category extends TActiveRecord
{
    public $cat_id;
    public $category_name;
    public $parent_cat_id;

    public $parent_category;            // this declaration is no longer needed since v3.1.2
    public $child_categories=array();   // this declaration is no longer needed since v3.1.2

    public static $RELATIONS=array
    (
        'parent_category' => array(self::BELONGS_TO, 'Category', 'parent_cat_id'),
        'child_categories' => array(self::HAS_MANY, 'Category', 'parent_cat_id'),
    );
}
```

Query Criteria for Related Objects

In the above, we show that an Active Record object can reference to its related objects by declaring a static class member $RELATIONS which specifies a list of relations. Each relation is specified as an array consisting of three elements: relation type, related AR class name, and the foreign key(s). For example, we use `array(self::HAS_MANY, 'PlayerRecord', 'team_name')` to specify the players in a team. There are two more optional elements that can be specified in this array: query condition (the fourth element) and parameters (the fifth element). They are used to control how to query for the related objects. For example, if we want to obtain the players ordered by their age, we can specify `array(self::HAS_MANY, 'PlayerRecord', 'team_name', 'ORDER BY age')`. If we want to obtain players whose age is smaller than 30, we could use ¡tt¿array(self::HAS**$_M ANY, 'PlayerRecord', 'team_name', 'age <: age', array(' : age' => 30)) $< /tt >$ .In general, these two additional elements are similar as the parameters passed to the `find()` method in AR.**

16.4.2 Association Table Mapping

Objects can handle multivalued fields quite easily by using collections as field values. Relational databases don't have this feature and are constrained to single-valued fields only. When you're mapping a one-to-many association you can handle this using has many relationships, essentially using a foreign key for the single-valued end of the association. But a many-to-many association can't do this because there is no single-valued end to hold the foreign key.

The answer is the classic resolution that's been used by relational data people for decades: create an extra table (an association table) to record the relationship. The basic idea is using an association table to store the association. This table has only the foreign key IDs for the two tables that are linked together, it has one row for each pair of associated objects.

The association table has no corresponding in-memory object and its primary key is the compound of the two primary keys of the tables that are associated. In simple terms, to load data from the association table you perform two queries (in general, it may also be achieved using one query consisting of joins). Consider loading the `SkillRecord` collection for a list `PlayerRecord` objects. In this case, you do queries in two stages. The first stage queries the `Players` table to find all the rows of the players you want. The second stage finds the `SkillRecord` object for the related player ID for each row in the `Player_Skills` association table using an inner join.

The Prado Active Record design implements the two stage approach. For the `Players-Skills` M-N (many-to-many) entity relationship, we define a many-to-many relationship in the `PlayerRecord` class and in addition we may define a many-to-many relationship in the `SkillRecord` class as well. The following sample code defines the complete `SkillRecord` class with a many-to-many relationship with the `PlayerRecord` class. (See the `PlayerRecord` class definition above to the corresponding many-to-many relationship with the `SkillRecord` class.)

```
class SkillRecord extends TActiveRecord
{
    const TABLE='Skills';
    public $skill_id;
    public $name;
```

```
    public $players=array();    // this declaration is no longer needed since v3.1.2

    public static $RELATIONS=array
    (
        'players' => array(self::MANY_TO_MANY, 'PlayerRecord', 'Player_Skills'),
    );

    public static function finder($className=__CLASS__)
    {
        return parent::finder($className);
    }
}
```

The static `$RELATIONS` property of SkillRecord defines that the property `$players` has many `PlayerRecords` via an association table '`Player_Skills`'. In `array(self::MANY_TO_MANY, 'PlayerRecord', 'Player_Skills')`, the first element defines the relationship type, in this case ¡strong¿`self::MANY_TO_MANY`¡/strong¿, the second element is a string `'PlayerRecord'` that corresponds to the class name of the `PlayerRecord` class, and the third element is the name of the association table name.

> **Note: Prior to version 3.1.2 (versions up to 3.1.1), the many-to-many relationship was defined using `self::HAS_MANY`. For version 3.1.2 onwards, this must be changed to `self::MANY_TO_MANY`. This can be done by searching for the `HAS_MANY` in your source code and carfully changing the appropriate definitions.**

A list of player objects with the corresponding collection of skill objects may be fetched as follows.

```
$players = PlayerRecord::finder()->withSkills()->findAll();
```

The method `with_xxx()` (where `xxx` is the relationship property name, in this case, `Skill`) fetches the corresponding `SkillRecords` using a second query (not by using a join). The `with_xxx()` accepts the same arguments as other finder methods of `TActiveRecord`.

Self Referenced Association Tables

For self referenced association tables, that is, the association points to the same table. For example, consider the `items` table with M-N related item via the `related_items` association table. The syntax in the following example is valid for a PostgreSQL database. For other database, consult their respective documentation for defining the foreign key constraints.

```
CREATE TABLE items
(
  "item_id" SERIAL,
  "name" VARCHAR(128) NOT NULL,
  PRIMARY KEY("item_id")
);
CREATE TABLE "related_items"
(
  "item_id" INTEGER NOT NULL,
  "related_item_id" INTEGER NOT NULL,
  CONSTRAINT "related_items_pkey" PRIMARY KEY("item_id", "related_item_id"),
  CONSTRAINT "related_items_item_id_fkey" FOREIGN KEY ("item_id")
    REFERENCES "items"("item_id")
    ON DELETE CASCADE
    ON UPDATE NO ACTION
    NOT DEFERRABLE,
  CONSTRAINT "related_items_related_item_id_fkey" FOREIGN KEY ("related_item_id")
    REFERENCES "items"("item_id")
    ON DELETE CASCADE
    ON UPDATE NO ACTION
    NOT DEFERRABLE
);
```

The association table name in third element of the relationship array may contain the foreign table column names. The columns defined in the association table must also be defined in the record class (e.g. the `$related_item_id` property corresponds to the `related_item_id` column in the `related_items` table).

```
class Item extends TActiveRecord
{
    const TABLE="items";
```

```
    public $item_id;
    public $details;

    //additional foreign item id defined in the association table
    public $related_item_id;
    public $related_items=array();    // this declaration is no longer needed since v3.1.2

    public static $RELATIONS=array
    (
        'related_items' => array(self::MANY_TO_MANY,
            'Item', 'related_items.related_item_id'),
    );
}
```

> **Tip: Compound keys in the foreign table can be specified as comma separated values between brackets. E.g.** `'related_items.(id1,id2)'`.

¡!—

16.4.3 Adding/Removing/Updating Related Objects

Related objects can be simply inserted/updated by first adding those related objects to the current source object (i.e. the object currently been worked on) and then call the `save()` method on the source object. The related object's references and the association reference (if required) will be added and/or updated. For example, to add two new players to the team (assuming that 'Team A' exists), we can simply do the following.

```
$team = TeamRecord::finder()->findByPk('Team A');
$team->players[] = new PlayerRecord(array('age'=>20));
$team->players[] = new PlayerRecord(array('age'=>25));
$team->save();
```

Since the `TeamRecord` class contains a has many relationship with the `PlayerRecord`, then saving a `TeamRecord` object will also update the corresponding foreign objects in `$players` array. That is, the objects in `$players` are inserted/updated in the database

and the `$team_name` property of those objects will contain the foreign key value that corresponds to the `$team` object's primary key value.

To delete a particular foreign object (or any Active Record object), simply call the object's `delete()` method. You may setup the database table's foreign key constraints such that when deleting a particular data in the database it will delete the referenced data as well (it may also be achieved using database triggers). E.g. such as having a "`ON DELETE CASCADE`" constraint. Deleting foreign objects by either setting the property value to null or removing the object from an array will NOT remove the corresponding data in the database.

To remove associations for the many-to-many relationships via an association table, an Active Record that corresponds to the association table can be used. Then the association can be removed by calling the `deleteByPk()` method, for example:

```
PlayerSkillAssocation::finder()->deleteByPk(array('fk1','fk2'));
//where 'fk1' is the primary key value of a player
// and 'fk2' is the primary key value of a skill
```

—¿

16.4.4 Lazy Loading Related Objects

> **Note: Implicitly declared related properties introduced in version 3.1.2 automatically have lazy loading feature. Therefore, the lazy loading technique described in the following is no longer needed in most of the cases, unless you want to manipulate the related objects through getter/setter.**

Using the `with_xxx()` methods will load the relationship record on demand. Retrieving the related record using lazy loading (that is, only when those related objects are accessed) can be achieved by using a feature of the `TComponent` that provides accessor methods. In particular, we define a pair of getter and setter methods where the getter method will retrieve the relationship conditionally. The following example illustrates that the `PlayerRecord` can retrieve its `$skills` foreign objects conditionally.

```
class PlayerRecord extends BaseFkRecord
```

```
{
    //... other properties and methods as before

    private $_skills; //change to private and default as null

    public function getSkills()
    {
        if($this->_skills===null && $this->player_id !==null)
        {
            //lazy load the skill records
            $this->setSkills($this->withSkills()->findByPk($this->player_id)->skills);
        }
        else if($this->_skills===null)
        {
            //create new TList;
            $this->setSkills(new TList());
        }

        return $this->_skills;
    }

    public function setSkills($value)
    {
        $this->_skills = $value instanceof TList ? $value : new TList($value);
    }
}
```

We first need to change the `$skills=array()` declaration to a private property `$_skills` (notice the underscore) and set it to null instead. This allows us to define the `skills` property using getter/setter methods (see Components for details). The `getSkills()` getter method for the `skills` property will lazy load the corresponding skills foreign record when it is used as follows. Notice that we only do a lazy load when its `$player_id` is not null (that is, when the record is already fetched from the database or player id was already set).

```
$player = PlayerRecord::finder()->findByPk(1);
var_dump($player->skills); //lazy load it on first access
var_dump($player->skills[0]); //already loaded skills property
$player->skills[] = new SkillRecord(); //add skill
```

The `setSkills()` ensures that the `skills` property will always be a TList. Using a TList allows us to set the elements of the `skills` property as if they were arrays. E.g. `$player->skills[] = new SkillRecord()`. If `array` was used, a PHP error will be thrown.

16.4.5 Column Mapping

Since v3.1.1, Active Record starts to support column mapping. Column mapping allows developers to address columns in Active Record using a more consistent naming convention. In particular, using column mapping, one can access a column using whatever name he likes, rather than limited by the name defined in the database schema.

To use column mapping, declare a static array named `COLUMN_MAPPING` in the Active Record class. The keys of the array are column names (called ¡i¿physical column names¡/i¿) as defined in the database schema, while the values are corresponding property names (called ¡i¿logical column names¡/i¿) defined in the Active Record class. The property names can be either public class member variable names or component property names defined via getters/setters. If a physical column name happens to be the same as the logical column name, they do not need to be listed in `COLUMN_MAPPING`.

```
class UserRecord extends TActiveRecord
{
     const TABLE='users';
     public static $COLUMN_MAPPING=array
     (
          'user_id'=>'id',
          'email_address'=>'email',
          'first_name'=>'firstName',
          'last_name'=>'lastName',
     );
     public $id;
     public $username; // the physical and logical column names are the same
     public $email;
     public $firstName;
     public $lastName;
     //....
```

```
}
```

With the above column mapping, we can address `first_name` using `$userRecord->firstName` instead of `$userRecord->first_name`. This helps separation of logic and model.

16.4.6 References

- **Fowler et. al. ¡i¿Patterns of Enterprise Application Architecture¡/i¿, Addison Wesley, 2002.**
- **B. Venners with B. Eckel. ¡i¿Inappropriate Abstractions - A Conversation with Anders Hejlsberg, Part VI.¡/i¿ Artima Developer, 2003.**

16.5 Active Record Scaffold Views

Active Record classes can be used together with TScaffoldListView and TScaffoldEditView (TScaffoldView links both `TScaffoldListView` and `TScaffoldEditView`) to create ¡i¿simple¡/i¿ Create/Read/Update/Delete (CRUD) web applications.

The scaffold views are intended to assist in prototyping web application, they are not designed to be as customiziable as more complex components such as TDataGrid. The scaffold views provide the following builtin functionality:

- **Listing of all active record items.**
- **Searching records.**
- **Paging and sorting.**
- **Deleting an item.**
- **Inserting a new item.**
- **Updating an existing item.**
- **Validates required fields and basic data types.**
- **Presents specialized controls such as date pickers.**

Scaffold views are dependent on Active Records and currently supports the following databases: Mysql, Sqlite and Postgres SQL. Support for other databases can be considered when there are sufficient demand.

16.5.1 Setting up a Scaffold View

To use the scaffold view, we first define an Active Record class that represents a table or view in the database. Consider the following Active Record class that corresponds to the `users` table as defined in the Active Record quickstart page.

```
class UserRecord extends TActiveRecord
{
    const TABLE='users';

    public $username;
    public $email;
}
```

The scaffold view classes are in the `System.Data.ActiveRecord.Scaffold.*` namespace. This namespace can be "imported" in the Application Configuration using the `application.xml` file or through the php code using the `Prado::using()` method. To start using the TScaffoldView simply set the `RecordClass` property value equal to an Active Record class name.

```
<com:TScaffoldView RecordClass="UserRecord" />
```

The above code will list the current records in the `users` table. Each record can be edited by clicking on the "edit" button and deleted by clicking on the "delete" button. A new record can be added by clicking on the "Add new record" button, enter some data (notice the automatic validation of required fields and data types), and click the "save" button. Specifying search terms in the search textbox to find particular records. Finally, the record list can be sorted for each column by changing the sorting column and order.

The `TScaffoldView` is a template control composed of other scaffold controls. The following properties gives access to these composite controls.

- ¡b¿ListView¡/b¿ – the TScaffoldListView displaying the record list.
- ¡b¿EditView¡/b¿ – the TScaffoldEditView that renders the inputs for editing and adding records.
- ¡b¿SearchControl¡/b¿ – the TScaffoldSearch responsible to the search user interface.

All these composite controls can be customized as we shall see below.

16.5.2 TScaffoldListView

A list of Active Records can be displayed using the TScaffoldListView with the following useful properties.

- ¡b¿Header¡/b¿ – a TRepeater displaying the Active Record property/field names.
- ¡b¿Sort¡/b¿ – a TDropDownList displaying the combination of properties and its possible ordering.
- ¡b¿Pager¡/b¿ – a TPager control displaying the links and/or buttons that navigate to different pages in the Active Record data.
- ¡b¿List¡/b¿ – a TRepeater that renders a row of Active Record data.

Custom rendering of the each Active Record can be achieved by specifying the ItemTemplate and/or AlternatingItemTemplate property of the List repeater. The TScaffoldListView will listen for two command events named "delete" and "edit". A "delete" command will delete a the record for the row where the "delete" command originates. An "edit" command will push the record data to be edited by a TScaffoldEditView with ID specified by the EditViewID property. The following example lists the usernames only with bold formatting.

```
<com:TScaffoldListView RecordClass="UserRecord" >
    <prop:List.ItemTemplate>
        <strong><%# $this->Data->username %></strong>
    </prop:List.ItemTemplate>
</com:TScaffoldListView>
```

> **Info: For the `TScaffoldView` the list view can be accessed throught the `ListView` property of a `TScaffoldView`. Thus, the subproperty `ListView.List.ItemTemplate` on `TScaffoldView` is equivalent to the `List.ItemTemplate` subproperty of `TScaffoldListView` in the above example.**

The `SearchCondition` property and `SearchParameters` property (takes array values) can be specified to customize the records to be shown. The `SearchCondition` will be used as the `Condition` property of `TActiveRecordCriteria` and the `SearchParameters` property corresponds to `Parameters` property of `TActiveRecordCriteria`.

16.5.3 TScaffoldEditView

```
<com:TScaffoldEditView RecordPk="user1" RecordClass="UserRecord" />
```

16.5.4 Combining list + edit views

```
<com:TScaffoldEditView ID="edit_view" RecordClass="UserRecord" />
<com:TScaffoldListView EditViewID="edit_view" RecordClass="UserRecord" />
```

16.5.5 Customizing the TScaffoldView

```
<com:TScaffoldView RecordClass="UserRecord" >
    <prop:ListView.List.ItemTemplate>
        <%# $this->DataItem->username %>
        <com:TLinkButton Text="Edit" CommandName="edit" />
    </prop:ListView.List.ItemTemplate>
</com:TScaffoldView/>
```

16.6 Data Mapper

Data Mappers moves data between objects and a database while keeping them independent of each other and the mapper itself. If you started with Active Records, you may eventually faced with more complex business objects as your project progresses.

When you build an object model with a lot of business logic it's valuable to use these mechanisms to better organize the data and the behavior that goes with it. Doing so leads to variant schemas; that is, the object schema and the relational schema don't match up.

The Data Mapper separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper the in-memory objects needn't know even that there's a database present; they need no SQL interface code, and certainly no knowledge of the database schema. (The database schema is always ignorant of the objects that use it.)

16.6.1 When to Use It

The primary occasion for using Data Mapper is when you want the database schema and the object model to evolve independently. Data Mapper's primary benefit is that when working on the business (or domain) objects you can ignore the database, both in design and in the build and testing process. The domain objects have no idea what the database structure is, because all the correspondence is done by the mappers.

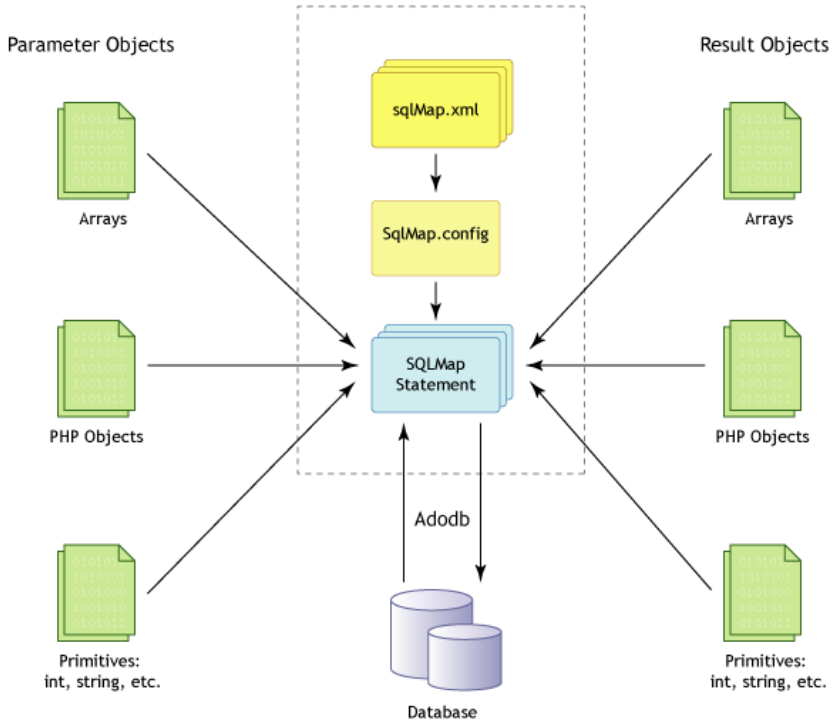
This helps you in the code because you can understand and work with the domain objects without having to understand how they're stored in the database. You can modify the business models or the database without having to alter either. With complicated mappings, particularly those involving existing databases, this is very valuable.

The price, of course, is the extra layer that you don't get with Active Record, so the test for using these patterns is the complexity of the business logic. If you have fairly simple business logic, an Active Record will probably work. For more complicated logic a Data Mapper may be more suitable.

16.6.2 SqlMap Data Mapper

The SqlMap DataMapper framework makes it easier to use a database with a PHP application. SqlMap DataMapper couples objects with stored procedures or SQL statements using a XML descriptor. Simplicity is the biggest advantage of the SqlMap DataMapper over object relational mapping tools. To use SqlMap DataMapper you

rely on your own objects, XML, and SQL. There is little to learn that you don't already know. With SqlMap DataMapper you have the full power of both SQL and stored procedures at your fingertip



Here's a high level description of the work flow illustrated in the figure above. Provide a parameter, either as an object or a primitive type. The parameter can be used to set runtime values in your SQL statement or stored procedure. If a runtime value is not needed, the parameter can be omitted.

Execute the mapping by passing the parameter and the name you gave the statement or procedure in your XML descriptor. This step is where the magic happens. The framework will prepare the SQL statement or stored procedure, set any runtime values using your parameter, execute the procedure or statement, and return the result.

In the case of an update, the number of rows affected is returned. In the case of a query, a single object, or a collection of objects is returned. Like the parameter, the result object, or collection of objects, can be a plain-old object or a primitive PHP type.

16.6.3 Setting up a database connection and initializing the SqlMap

A database connection for SqlMap can be set as follows. See Establishing Database Connection for futher details regarding creation of database connection in general.

```
//create a connection and give it to the SqlMap manager.
$dsn = 'pgsql:host=localhost;dbname=test'; //Postgres SQL
$conn = new TDbConnection($dsn, 'dbuser','dbpass');
$manager = new TSqlMapManager($conn);
$manager->configureXml('my-sqlmap.xml');
$sqlmap = $manager->getSqlMapGateway();
```

The `TSqlMapManager` is responsible for setting up the database connection and configuring the SqlMap with given XML file(s). The `configureXml()` method accepts a string that points to a SqlMap XML configuration file. Once configured, call the `getSqlMapGateway()` method to obtain an instance of the SqlMap gateway interface (use this object to insert/delete/find records).

SqlMap database connection can also be configured using a `<module>` tag in the application.xml or config.xml as follows.

```
<modules>
  <module id="my-sqlmap" class="System.Data.SqlMap.TSqlMapConfig"
        EnableCache="true" ConfigFile="my-sqlmap.xml" >
    <database ConnectionString="pgsql:host=localhost;dbname=test"
        Username="dbuser" Password="dbpass" />
  </module>
</modules>
```

The `ConfigFile` attribute should point to a SqlMap configuration file (to be detailed later) either using absolute path, relative path or the Prado's namespace dot notation path (must omit the ".xml" extension).

> **Tip: The `EnableCache` attribute when set to "true" will cache the parsed configuration. You must clear or disable the cache if you make changes to your configuration file. A cache module must also be defined for the cache to function.**

To obtain the SqlMap gateway interface from the ¡module¿ configuration, simply do, for example,

```
class MyPage extends TPage
{
    public function onLoad($param)
    {
        parent::onLoad($param);
        $sqlmap = $this->Application->Modules['my-sqlmap']->Client;
        $sqlmap->queryForObject(...); //query for some object
    }
}
```

16.6.4 A quick example

Let us consider the following "users" table that contains two columns named "username" and "email", where "username" is also the primary key.

```
CREATE TABLE users
(
    username VARCHAR( 20 ) NOT NULL ,
    email VARCHAR( 200 ) ,
    PRIMARY KEY ( username )
);
```

Next we define our plain `User` class as follows. Notice that the `User` is very simple.

```
class User
{
    public $username;
    public $email;
}
```

Next, we need to define a SqlMap XMl configuration file, lets name the file as `my-sqlmap.xml`

```
<?xml version="1.0" encoding="utf-8" ?>
<sqlMapConfig>
```

```
    <select id="SelectUsers" resultClass="User">
        SELECT username, email FROM users
    </select>
</sqlMapConfig>
```

The ¡select¿ tag returns defines an SQL statement. The `id` attribute will be used as the identifier for the query. The `resultClass` attribute value is the name of the class the the objects to be returned. We can now query the objects as follows:

```
//assume that $sqlmap is an TSqlMapGateway instance
$userList = $sqlmap->queryForList("SelectUsers");

//Or just one, if that's all you need:
$user = $sqlmap->queryForObject("SelectUsers");
```

The above example shows demonstrates only a fraction of the capabilities of the SqlMap Data Mapper. Further details can be found in the SqlMap Manual.

16.6.5 Combining SqlMap with Active Records

The above example may seem trival and it also seems that there is alot work just to retrieve some data. However, notice that the `User` class is totally unware of been stored in the database, and the database is unware of the `User` class.

One of advantages of SqlMap is the ability to map complex object relationship, collections from an existing database. On the other hand, Active Record provide a very simple way to interact with the underlying database but unable to do more complicated relationship or collections. A good compromise is to use SqlMap to retrieve complicated relationships and collections as Active Record objects and then using these Active Records to do the updates, inserts and deletes.

Continuing with the previous example, we change the definition of the `User` class to become an Active Record.

```
class UserRecord extends TActiveRecord
{
     const TABLE='users'; //table name
```

```
    public $username; //the column named "username" in the "users" table
    public $email;

    /**
     * @return TActiveRecord active record finder instance
     */
    public static function finder($className=__CLASS__)
    {
        return parent::finder($className);
    }
}
```

We also need to change the definition of the SqlMap XML configuration. We just need to change the value of `resultClass` attribute to `UserRecord`.

```
<?xml version="1.0" encoding="utf-8" ?>
<sqlMapConfig>
    <select id="SelectUsers" resultClass="UserRecord">
        SELECT username, email FROM users
    </select>
</sqlMapConfig>
```

The PHP code for retrieving the users remains the same, but SqlMap returns Active Records instead, and we can take advantage of the Active Record methods.

```
//assume that $sqlmap is an TSqlMapGateway instance
$user = $sqlmap->queryForObject("SelectUsers");

$user->email = 'test@example.com'; //change data
$user->save(); //save it using Active Record
```

16.6.6 References

- **Fowler et. al. ¡i¿Patterns of Enterprise Application Architecture¡/i¿, Addison Wesley, 2002.**
- **iBatis Team. ¡i¿iBatis Data Mapper¡/i¿, http://ibatis.apache.org.**

Chapter 17

Advanced Topics

17.1 Collections

Collection is a basic data structure in programming. In traditional PHP programming, array is used widely to represent collection data structure. A PHP array is a mix of cardinal-indexed array and hash table.

To enable object-oriented manipulation of collections, PRADO provides a set of powerful collection classes. Among them, the `TList` and `TMap` are the most fundamental and usually serve as the base classes for other collection classes. Since many PRADO components have properties that are of collection type, it is very important for developers to master the usage of PRADO collection classes.

17.1.1 Using `TList`

A `TList` object represents a cardinal-indexed array, i.e., an array (object) with the index 0, 1, 2, ...

`TList` may be used like a PHP array. For example,

```
$list=new TList; // create a list object
...
```

```
$item=$list[$index];  // read the item at the specified index
$list[]=$item;  // append the item at the end
$list[$index]=$item; // replace the item at the specified index
unset($list[$index]); // remove the item at $index
if(isset($list[$index])) // test if the list has an item at $index
foreach($list as $index=>$item) // traverse each item in the list
```

To obtain the number of items in the list, use the `Count` property. Note, do not use `count($list)`, as it always returns 1.

In addition, `TList` implements a few commonly used convenient methods for manipulating the data in a list. These include

- **`clear()`: removes all items in the list.**
- **`contains()`: tests if the list contains the specified item.**
- **`indexOf()`: obtains the zero-based index of the specified item in the list.**
- **`toArray()`: returns an array representation of the items in the list.**
- **`copyFrom()`: populates the list with data from an array or traversable object (including `TList`). Existing items will be removed first.**
- **`mergeWith()`: appends the list with data from an array or traversable object (including `TList`).**

Using `TList`-based component properties

As aforementioned, many PRADO component properties are based on `TList` or `TList`-derived collection classes. These properties all share the above usages.

For example, `TControl` (the base class for all PRADO controls) has a property called `Controls` which represents the collection of child controls. The type of `Controls` is `TControlCollection` which extends `TList`. Therefore, to append a new child control, we can use the following,

```
$control->Controls[]=$newControl;
```

To traverse through the child controls, we can use,

```
foreach($control->Controls as $childControl) ...
```

Another example is the `Items` property, available in list controls, `TRepeater`, `TDataList` and `TDataGrid`. In these controls, the ancestor class of `Items` is `TList`.

Extending `TList`

Often, we want to extend `TList` to perform additional operations for each addition or removal of an item. The only methods that the child class needs to override are `insertAt()` and `removeAt()`. For example, to ensure the list only contains items that are of `TControl` type, we can override `insertAt()` as follows,

```
public function insertAt($index,$item)
{
    if($item instanceof TControl)
        parent::insertAt($index,$item)
    else
        throw new Exception('TControl required.');
}
```

17.1.2 Using `TMap`

A `TMap` object represents a hash table (or we say string-indexed array).

Similar to `TList`, `TMap` may be used like an array,

```
$map=new TMap; // create a map object
...
$map[$key]=$value; // add a key-value pair
unset($map[$key]); // remove the value with the specified key
if(isset($map[$key])) // if the map contains the key
foreach($map as $key=>$value) // traverse the items in the map
```

The `Count` property gives the number of items in the map while the `Keys` property returns a list of keys used in the map.

The following methods are provided by `TMap` for convenience,

- `clear()`: **removes all items in the map.**
- `contains()`: **tests if the map contains the specified key.**
- `toArray()`: **returns an array representation of the items in the map.**
- `copyFrom()`: **populates the map with data from an array or traversable object (including `TMap`). Existing items will be removed first.**
- `mergeWith()`: **appends the map with data from an array or traversable object (including `TMap`).**

Using of `TAttributeCollection`

`TAttributeCollection` is a special class extending from `TMap`. It is mainly used by the `Attributes` property of `TControl`.

Besides the normal functionalities provided by `TMap`, `TAttributeCollection` allows you to get and set collection items like getting and setting properties. For example,

```
$collection->Label='value'; // equivalent to: $collection['Label']='value';
echo $collection->Label; // equivalent to: echo $collection['Label'];
```

Note, in the above `$collection` does NOT have a `Label` property.

Unlike `TMap`, keys in `TAttributeCollection` are case-insensitive. Therefore, `$collection->Label` is equivalent to `$collection->LABEL`.

Because of the above new features, when dealing with the `Attributes` property of controls, we may take advantage of the subproperty concept and configure control attribute values in a template as follows,

```
<com:TButton Attributes.onclick="if(!confirm('Are you sure?')) return false;" .../>
```

which adds an attribute named `onclick` to the `TButton` control.

17.2 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.

17.2.1 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.

17.2.2 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, not set or wildcard '*', 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. If empty, not set or wildcard '*', this rule will apply to all users including anonymous/guest user. A character ? refers to anonymous/guest user. And a character @ refers to authenticated users (available since v3.1).
- `roles` - list of comma-separated user roles that this rule applies to. If empty, not set or wildcard '*', this rule will apply to all user roles.
- `verb` - page access method that this rule applies to. It can be either `get` or `post`. If empty, not set or wildcard '*', the rule will apply to both methods.

When a page request is being processed, a list of authorization rules may be available. However, only the ¡i¿first effective¡/i¿ rule ¡i¿matching¡/i¿ 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 ¡i¿effective¡/i¿ 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).

Since version 3.1.1, the `pages` attribute in the authorization rules can take relative page paths with wildcard '*'. For example, `pages="admin.Home"` refers to the `Home` page under the `admin` directory, and `pages="admin.*"` would refer to all pages under the `admin` directory and subdirectories.

Also introduced in version 3.1.1 are IP rules. They are specified by a new attribute `ips` in authorization rules. The IP rules are used to determine if an authorization rule aplies to an end-user according to his IP address. One can list a few IPs together, separated by comma ','. Wildcard '*' can be used in the rules. For example,

`ips="192.168.0.2, 192.168.1.*"` **means the rule applies to users whose IP address is 192.168.0.2 or 192.168.1.*. The latter matches any host in the subnet 192.168.1. If the attribute 'ips' is empty, not set or wildcard '*', the corresponding rule will apply to requests coming from any host address.**

17.2.3 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.**

17.2.4 Using `TDbUserManager`

`TDbUserManager` **is introduced in v3.1.0. Its main purpose is to simplify the task of managing user accounts that are stored in a database. It requires developers to write a user class that represents the necessary information for a user account. The user class must extend from** `TDbUser`**.**

To use `TDbUserManager`**, configure it in the application configuration like following:**

```
<module id="db"
     class="System.Data.TDataSourceConfig" ..../>
<module id="users"
     class="System.Security.TDbUserManager"
     UserClass="Path.To.MyUserClass"
     ConnectionID="db" />
<module id="auth"
```

```
    class="System.Security.TAuthManager"
    UserManager="users" LoginPage="Path.To.LoginPage" />
```

In the above, `UserClass` specifies what class will be used to create user instance. The class must extend from `TDbUser`. `ConnectionID` refers to the ID of a `TDataSourceConfig` module which specifies how to establish database connection to retrieve user information.

The user class has to implement the two abstract methods in `TDbUser`: `validateUser()` and `createUser()`. Since user account information is stored in a database, the user class may make use of its `DbConnection` property to reach the database.

Since 3.1.1, `TAuthManager` provides support to allow remembering login by setting `AllowAutoLogin` to true. Accordingly, `TDbUser` adds two methods to facilitate the implementation of this feature. In particular, two new methods are introduced: `createUserFromCookie()` and `saveUserToCookie()`. Developers should implement these two methods if remembering login is needed. Below is a sample implementation:

```
public function createUserFromCookie($cookie)
{
    if(($data=$cookie->Value)!==’’)
    {
        $application=Prado::getApplication();
        if(($data=$application->SecurityManager->validateData($data))!==false)
        {
            $data=unserialize($data);
            if(is_array($data) && count($data)===3)
            {
                list($username,$address,$token)=$data;
                $sql=’SELECT passcode FROM user WHERE LOWER(username)=:username’;
                $command=$this->DbConnection->createCommand($sql);
                $command->bindValue(’:username’,strtolower($username));
                if($token===$command->queryScalar() && $token!==false && $address=$application->Request->
                    return $this->createUser($username);
            }
        }
    }
    return null;
}
```

```
public function saveUserToCookie($cookie)
{
     $application=Prado::getApplication();
     $username=strtolower($this->Name);
     $address=$application->Request->UserHostAddress;
     $sql='SELECT passcode FROM user WHERE LOWER(username)=:username';
     $command=$this->DbConnection->createCommand($sql);
     $command->bindValue(':username',strtolower($username));
     $token=$command->queryScalar();
     $data=array($username,$address,$token);
     $data=serialize($data);
     $data=$application->SecurityManager->hashData($data);
     $cookie->setValue($data);
}
```

17.3 Security

17.3.1 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 is modified, PRADO will stop restoring the viewstate and return 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 `TSecurityManager` in application configuration,

```
<modules>
    <module id="security"
        class="TSecurityManager"
        ValidationKey="my private key" />
</modules>
```

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. To enable viewstate encryption, set the `EnableStateEncryption` of pages to true. This can be done in page configurations or in page code. Note, encrypting viewstate may degrade the application performance. A better strategy is to store viewstate on the server side, rather than the default hidden field.

17.3.2 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 preferable 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.

17.3.3 Cookie Attack Prevention

Protecting cookies from being attacked is of extreme important, as session IDs are commonly stored in cookies. If one gets hold of a session ID, he essentially owns all relevant session information.

There are several countermeasures to prevent cookies from being attacked.

- **An application can use SSL to create a secure communication channel and only pass the authentication cookie over an HTTPS connection. Attackers are thus unable to decipher the contents in the transferred cookies.**
- **Expire sessions appropriately, including all cookies and session tokens, to reduce the likelihood of being attacked.**
- **Prevent cross-site scripting (XSS) which causes arbitrary code to run in a user's browser and expose his cookies.**
- **Validate cookie data and detect if they are altered.**

PRADO implements a cookie validation scheme that prevents cookies from being modified. In particular, it does HMAC check for the cookie values if cookie validation is enable.

Cookie validation is disabled by default. To enable it, configure the `THttpRequest` module as follows,

```
<modules>
  <module id="request" class="THttpRequest" EnableCookieValidation="true" />
</modules>
```

To make use of cookie validation scheme provided by PRADO, you also need to retrieve cookies through the `Cookies` collection of `THttpRequest` by using the following PHP statements,

```
foreach($this->Request->Cookies as $cookie)
    // $cookie is of type THttpCookie
```

To send cookie data encoded with validation information, create new `THttpCookie` objects and add them to the `Cookies` collection of `THttpResponse`,

```
$cookie=new THttpCookie($name,$value);
$this->Response->Cookies[]=$cookie;
```

17.4 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.

17.4.1 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 `$object->publishAsset($assetPath)` to publish an asset and obtain its URL. Here, `$object` refers to an instance of `TApplicationComponent`

or derived class, and `$assetPath` is a file or directory relative to the directory containing the 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.

17.4.2 Customization

Asset publishing is managed by the `System.Web.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 `TAssetManager` module in application configuration,

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

17.4.3 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.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 checking is 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 component 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.

17.4.4 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.

17.5 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:

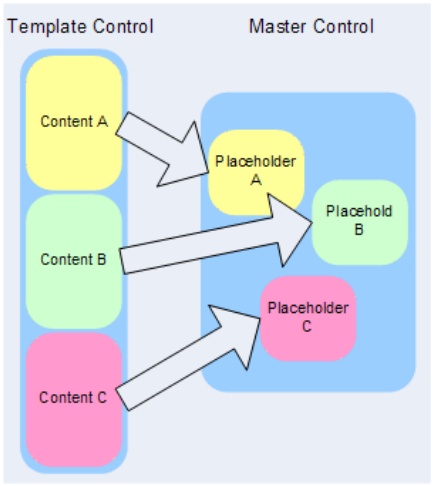
```
<%@ MasterClass="MasterControl" %>
<com:TContent ID="A" >
content A
</com:TContent >
<com:TContent ID="B" >
content B
</com:TContent >
<com:TContent ID="B" >
content B
</com:TContent >
```

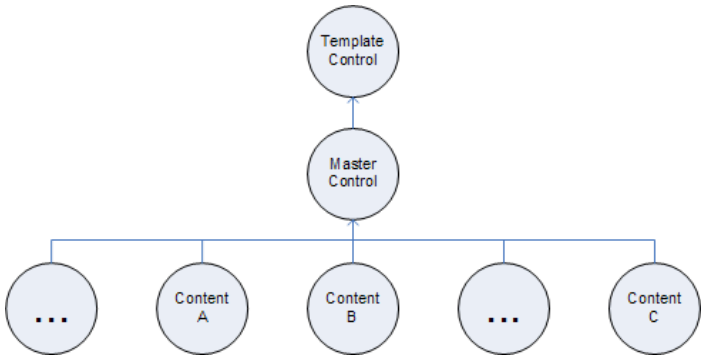
which uses `MasterControl` as its master control. The master control has the following template,

```
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.





17.5.1 Master vs. External Template

Master is very similar to external templates which are introduced since version 3.0.5. A special include tag is used to include an external template file into a base template.

Both master and external template can be used to share common contents among pages. A master is a template control whose template contains the common content and whose class file contains the logic associated with the master. An external template, on the other hand, is a pure template file without any class files.

Therefore, use master control if the common content has to be associated with some logic, such as a page header with search box or login box. A master control allows you to specify how the common content should interact with end users. If you use external templates, you will have to put the needed logic in the page or control class who owns the base template.

Performancewise, external template is lighter than master as the latter is a self-contained control participating the page lifecycles, while the former is used only when the template is being parsed.

17.6 Themes and Skins

17.6.1 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.

17.6.2 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 unique `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.

17.6.3 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.

It is possible to specify media types of CSS files contained in a theme. By default, a CSS file applies to all media types. If the CSS file is named like `mystyle.print.css`, it will be applied only to `print` media type. As another example, `mystyle.screen.css` applies to `screen` media only, and `mystyle.css` applies to all media types.

17.6.4 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>
```

17.6.5 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.

17.7 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.

17.7.1 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 round trips 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');
```

17.7.2 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 following commands,

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

17.7.3 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 following commands,

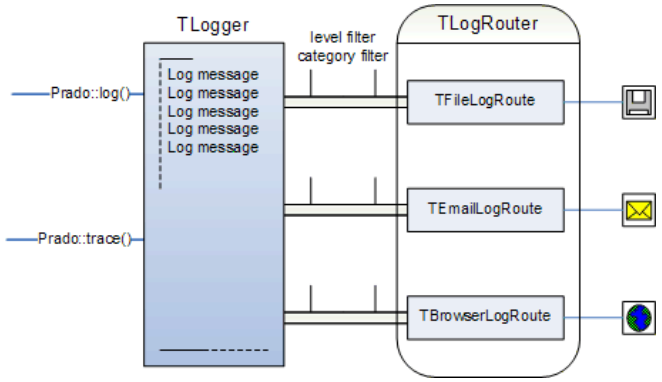
```
$application->setGlobalState('Caption',$caption);
$caption = $application->getGlobalState('Caption');
```

17.7.4 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.

17.8 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,



17.8.1 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.

17.8.2 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.Util.TLogRouter` module. When plugged into an application, it can route the messages to different destination in parallel. Currently, PRADO provides four 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.**
- **`TFirebugLogRoute` - filtered messages are sent to the Firebug console**

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

```
<module id="log" class="System.Util.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.

17.8.3 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.Util.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 child 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, messages logged in `TPage` will be of category `System.Web.UI.TPage`.

17.9 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 cul-

tures. 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.

17.9.1 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.**
- **Honorific 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).

17.9.2 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"
        marker="@@"
        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. The `marker` value is used to surround any untranslated text.

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.

17.9.3 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.

17.9.4 Using a Database for translation

Since version 3.1.3 the messages can also be stored in a database using the connection id from an existing `TDataSourceConfig`. You have to create two tables in your database: `catalogue` and `trans_unit`. The catalogue table needs an entry for each catalogue you want to use. Example schemas for different databases can be found in the framework's `I18N/schema` directory. To configure translation with a database use:¡/¿

```
<module id="db1" class="System.Data.TDataSourceConfig">
    <database ConnectionString="mysql:host=localhost;dbname=demodb" Username="demo" Password="demo" />
</module>

<module id="globalization" class="TGlobalization">
    <translation
        type="Database"
        autosave="true"
        cache="false"
```

```
        source="db1" />
</module>
```

The translation messages will be stored in the `trans_unit` table. Add your translation in the `target` field of that table. You should make sure that you are working on the right catalogue by comparing the message's `cat_id` with that from the catalogue table.

17.9.5 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" %>
```

17.9.6 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.

17.9.7 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=Prado::localize("Hello, world!");
}
```

17.9.8 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;
$message = Prado::localize("There are {num_users} users online.", array('num_users'=>$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**.

The following sample demonstrates the basics of localization in PRADO. **Advanced.Samples.I18N.Home Demo**

17.10 I18N Components

17.10.1 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. The `Parameters` property can be use to add name values pairs for substitution. Substrings in the translation enclosed with "{" and "}" are consider as the parameter names during substitution lookup. The following example will substitute the substring "{time}" with the value of the parameter attribute "`Parameters.time=<%= time() %>`".

```
<com:TTranslate Parameters.time=<%= time() %> >
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! ]%>" />
```

17.10.2 TDateFormat

Formatting localized date and time is straight forward.

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

The `Pattern` property accepts 4 predefined localized date patterns and 4 predefined localized time patterns.

- `fulldate`
- `longdate`
- `mediumdate`
- `shortdate`
- `fulltime`
- `longtime`
- `mediumtime`
- `shorttime`

The predefined can be used in any combination. If using a combined predefined pattern, the first pattern must be the date, followed by a space, and lastly the time pattern. For example, full date pattern with short time pattern. The actual ordering of the date-time and the actual pattern will be determine automatically from locale data specified by the `Culture` property.

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

You can also specify a custom pattern using the following sub-patterns. The date/time format is specified by means of a string time pattern. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following:

Symbol	Meaning	Presentation	Example
G	era designator	(Text)	AD

y	year	(Number)	1996
M	month in year	(Text & Number)	July & 07
d	day in month	(Number)	10
h	hour in am/pm (1~12)	(Number)	12
H	hour in day (0~23)	(Number)	0
m	minute in hour	(Number)	30
s	second in minute	(Number)	55
E	day of week	(Text)	Tuesday
D	day in year	(Number)	189
F	day of week in month	(Number)	2 (2nd Wed in July)
w	week in year	(Number)	27
W	week in month	(Number)	2
a	am/pm marker	(Text)	PM
k	hour in day (1~24)	(Number)	24
K	hour in am/pm (0~11)	(Number)	0
z	time zone	(Time)	Pacific Standard Time
'	escape for text	(Delimiter)	'Date='
''	single quote	(Literal)	'o''clock'

The count of pattern letters determine the format.

(Text): 4 letters uses full form, less than 4, use short or abbreviated form if it exists. (e.g., "EEEE" produces "Monday", "EEE" produces "Mon")

(Number): the minimum number of digits. Shorter numbers are zero-padded to this amount (e.g. if "m" produces "6", "mm" produces "06"). Year is handled specially; that is, if the count of 'y' is 2, the Year will be truncated to 2 digits. (e.g., if "yyyy" produces "1997", "yy" produces "97".) Unlike other fields, fractional seconds are padded on the right with zero.

(Text and Number): 3 or over, use text, otherwise use number. (e.g., "M" produces "1", "MM" produces "01", "MMM" produces "Jan", and "MMMM" produces "January".)

Any characters in the pattern that are not in the ranges of ['a'..'z'] and ['A'..'Z'] will be treated as quoted text. For instance, characters like ':', '.', ' ', and '@' will appear in the resulting time text even they are not embraced within single quotes.

Examples using the US locale:

```
Format Pattern                          Result
--------------                          -------
"yyyy.MM.dd G 'at' HH:mm:ss"      ->>  1996.07.10 AD at 15:08:56
"EEE, MMM d, ''yy"                ->>  Wed, Jul 10, '96
"h:mm a"                          ->>  12:08 PM
"hh 'o''clock' a, z"              ->>  12 o'clock PM, Pacific Daylight Time
"K:mm a"                          ->>  0:00 PM
"yyyy.MMMM.dd G hh:mm a"          ->>  1996.July.10 AD 12:08 PM
```

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

17.10.3 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.

Numbers can be formatted as currency, percentage, decimal or scientific numbers by specifying the `Type` attribute. The valid types are:

- `currency`
- `percentage`
- `decimal`
- `scientific`

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

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

If someone from US want to see sales figures from a store in Germany (say using the EURO currency), formatted using the german currency, you would need to use the attribute `Culture="de_DE"` to get the currency right, e.g. 100,00$. The decimal and grouping separator is then also from the `de_DE` locale. This may lead to some confusion because people from US uses the "," (comma) as thousand separator. Therefore a `Currency` attribute is available, so that the output from the following example results in $100.00

```
<com:TNumberFormat Type="currency"
          Culture="en_US" Currency="EUR" Value="100" />
```

The `Pattern` property determines the number of digits, thousand grouping positions, the number of decimal points and the decimal position. The actual characters that are used to represent the decimal points and thousand points are culture specific and will change automatically according to the `Culture` property. The valid `Pattern` characters are:

- **`#` (`hash`) - represents the optional digits**
- **`0` (`zero`) - represents the mandatory digits, zero left filled**
- **`.` (`full stop`) - the position of the decimal point (only 1 decimal point is allowed)**
- **`,` (`comma`) - thousand point separation (up to 2 commas are allowed)**

For example, consider the `Value="1234567.12345"` and with `Culture="en_US"` (which uses "," for thousand point separator and "." for decimal separators).

```
Pattern                    Output
-------                    ------
##,###.00             ->>  1,234,567.12
##,###.##             ->>  1,234,567.12345
##,##.0000            ->>  1,23,45,67.1235
##,###,##.0           ->>  12,345,67.1
000,000,000.0         ->>  001,234,567.1
```

17.10.4 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>
```

17.10.5 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.

¡pi class=”block-content”¿ Since version 3.1.1 the following set notation is also possible.

- **¡tt¿{n: n % 10 ¿ 1 n % 10 ¡ 5}¡/tt¿ – matches numbers like 2, 3, 4, 22, 23, 24**

Where set is defined by the expression after `n:`. In particular, the expression accepts the following mathematical/logical operators to form a set of logical conditions on the value given by `n`:

- **`<` – less than.**
- **`<=` – less than equals.**
- **`>` – greater than.**
- **`gt=` – greater than equals.**
- **`==` – of equal value.**
- **`%` – modulo, e.g., 1 % 10 equals 1, 11 % 10 equals 1.**
- **`-` – minus, negative.**
- **`+` – addition.**
- **`&` – conditional AND.**
- **`&&` – condition AND with short circuit.**
- **`|` – conditional OR.**
- **`||` – conditional OR with short circuit.**
- **`!` – negation.**

Additional round brackets can also be used to perform grouping. The following example represents ordinal values in English such as: “0th”, “1st”, “2nd”, “3rd”, “4th”, “11th”, “21st”, “22nd”, etc.

```
<com:TChoiceFormat Value="21">
 {n: n > 0 && n % 10 == 1 && n % 100 != 11} {Value}st
|{n: n > 0 && n % 10 == 2 && n % 100 != 12} {Value}nd
```

```
|{n: n > 0 && n % 10 == 3 && n % 100 != 13} {Value}rd
|{n: n > -1 } {Value}th
|(-Inf, 0) {Value}
</com:TChoiceFormat>
```

17.11 Error Handling and Reporting

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

17.11.1 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 caught 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` - **catchable 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`**.**

17.11.2 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.**

17.11.3 Error Capturing and Reporting

Exceptions raised during the runtime of PRADO applications are captured by `System.Exceptions.TErrorH` **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.**

17.11.4 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.**

17.12 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.

17.12.1 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.Caching.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.

Since v3.1.0, a new control called TOutputCache has been introduced. This control allows users to selectively cache parts of a page's output. When used appropriately, this technique can significant improve pages' performance because the underlying controls are not created at all if the cached versions are hit.

17.12.2 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`.

17.12.3 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 >
```

17.12.4 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/control.

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 `StatePersisterClass` property of the page determines which state persistence class to use. By default, it uses `System.Web.UI.TPageStatePersister` to store persistent state in hidden fields. You may modify this property to a persister class of your own, as long as the new persister class implements the `IPageStatePersister` interface. You may configure this property in several places, such as application configuration or page configuration using ¡pages¿ or ¡page¿ tags,

```
<pages StatePersisterClass="MyPersister1" ... >
    <page ID="SpecialPage" StatePersisterClass="MyPersister2" ... />
</pages>
```

Note, in the above the `SpecialPage` will use `MyPersister2` as its persister class, while the rest pages will use `MyPersister1`. Therefore, you can have different state persister strategies for different pages.

17.12.5 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.

Chapter 18

Client-side Scripting

18.1 Introduction to Javascript

This guide is based on the Quick guide to somewhat advanced JavaScript tour of some OO features by Sergio Pereira.

18.1.1 Hey, I didn't know you could do that

If you are a web developer and come from the same place I do, you have probably used quite a bit of Javascript in your web pages, mostly as UI glue.

Until recently, I knew that Javascript had more OO capabilities than I was employing, but I did not feel like I needed to use it. As the browsers started to support a more standardized featureset of Javascript and the DOM, it became viable to write more complex and functional code to run on the client. That helped giving birth to the AJAX phenomena.

As we all start to learn what it takes to write our cool, AJAX applications, we begin to notice that the Javascript we used to know was really just the tip of the iceberg. We now see Javascript being used beyond simple UI chores like input validation and frivolous tasks. The client code now is far more advanced and layered, much like a real desktop application or a client-server thick client. We see class libraries, object

models, hierarchies, patterns, and many other things we got used to seeing only in our server side code.

In many ways we can say that suddenly the bar was put much higher than before. It takes a heck lot more proficiency to write applications for the new Web and we need to improve our Javascript skills to get there. If you try to use many of the existing javascript libraries out there, like Prototype.js, Scriptaculous, moo.fx, Behaviour, YUI, etc you'll eventually find yourself reading the JS code. Maybe because you want to learn how they do it, or because you're curious, or more often because that's the only way to figure out how to use it, since documentation does not seem to be highly regarded with most of these libraries. Whatever the case may be, you'll face some kung-fu techniques that will be foreign and scary if you haven't seen anything like that before.

The purpose of this article is precisely explaining the types of constructs that many of us are not familiar with yet.

18.1.2 JSON (JavaScript Object Notation)

JavaScript Object Notation (JSON,) is one of the new buzzwords popping up around the AJAX theme. JSON, simply put, is a way of declaring an object in Javascript. Let's see an example right away and note how simple it is.

```
var myPet = { color: 'black', leg_count: 4, communicate: function(repeatCount){
for(i=0;i<repeatCount;i++) alert('Woof!');} };
```

Let's just add little bit of formatting so it looks more like how we usually find out there:

```
var myPet =
{
    color: 'black',
    legCount: 4,
    communicate: function(repeatCount)
    {
        for(i=0;i<repeatCount;i++)
            alert('Woof!');
```

```
    }
};
```

Here we created a reference to an object with two properties (`color` and `legCount`) and a method (`communicate`.) It's not hard to figure out that the object's properties and methods are defined as a comma delimited list. Each of the members is introduced by name, followed by a colon and then the definition. In the case of the properties it is easy, just the value of the property. The methods are created by assigning an anonymous function, which we will explain better down the line. After the object is created and assigned to the variable `myPet`, we can use it like this:

```
alert('my pet is ' + myPet.color);
alert('my pet has ' + myPet.legCount + ' legs');
//if you are a dog, bark three times:
myPet.communicate(3);
```

You'll see JSON used pretty much everywhere in JS these days, as arguments to functions, as return values, as server responses (in strings,) etc.

18.1.3 What do you mean? A function is an object too?

This might be unusual to developers that never thought about that, but in JS a function is also an object. You can pass a function around as an argument to another function just like you can pass a string, for example. This is extensively used and very handy.

Take a look at this example. We will pass functions to another function that will use them.

```
var myDog =
{
    bark: function()
    {
        alert('Woof!');
    }
};
```

```
var myCat =
{
    meow: function()
    {
        alert('I am a lazy cat. I will not meow for you.');
    }
};

function annoyThePet(petFunction)
{
    //let's see what the pet can do
    petFunction();
}

//annoy the dog:
annoyThePet(myDog.bark);
//annoy the cat:
annoyThePet(myCat.meow);
```

Note that we pass myDog.bark and myCat.meow without appending parenthesis `"()"` to them. If we did that we would not be passing the function, rather we would be calling the method and passing the return value, `undefined` in both cases here.

If you want to make my lazy cat start barking, you can easily do this:

```
myCat.meow = myDog.bark;
myCat.meow(); //alerts 'Woof!'
```

18.1.4 Arrays, items, and object members

The following two lines in JS do the same thing.

```
var a = new Array();
var b = [];
```

As I'm sure you already know, you can access individual items in an array by using the square brackets:

```
var a = ['first', 'second', 'third'];
var v1 = a[0];
var v2 = a[1];
var v3 = a[2];
```

But you are not limited to numeric indices. You can access any member of a JS object by using its name, in a string. The following example creates an empty object, and adds some members by name.

```
var obj = {}; //new, empty object
obj['member_1'] = 'this is the member value';
obj['flag_2'] = false;
obj['some_function'] = function(){ /* do something */};
```

The above code has identical effect as the following:

```
var obj =
{
    member_1:'this is the member value',
    flag_2: false,
    some_function: function(){ /* do something */}
};
```

In many ways, the idea of objects and associative arrays (hashes) in JS are not distiguishable. The following two lines do the same thing too.

```
obj.some_function();
obj['some_function']();
```

18.1.5 Enough about objects, may I have a class now?

The great power of object oriented programming languages derive from the use of classes. I don't think I would have guessed how classes are defined in JS using only my previous experience with other languages. Judge for yourself.

```
//defining a new class called Pet
```

```
var Pet = function(petName, age)
{
    this.name = petName;
    this.age = age;
};

//let's create an object of the Pet class
var famousDog = new Pet('Santa\'s Little Helper', 15);
alert('This pet is called ' + famousDog.name);
```

Let's see how we add a method to our `Pet` class. We will be using the `prototype` property that all classes have. The `prototype` property is an object that contains all the members that any object of the class will have. Even the default JS classes, like `String`, `Number`, and `Date` have a `prototype` object that we can add methods and properties to and make any object of that class automatically gain this new member.

```
Pet.prototype.communicate = function()
{
    alert('I do not know what I should say, but my name is ' + this.name);
};
```

That's when a library like prototype.js comes in handy. If we are using prototype.js, we can make our code look cleaner (at least in my opinion.)

```
var Pet = Class.create();
Pet.prototype =
{
    //our 'constructor'
    initialize: function(petName, age)
    {
        this.name = petName;
        this.age = age;
    },

    communicate: function()
    {
        alert('I do not know what I should say, but my name is ' + this.name);
    }
};
```

18.1.6 Functions as arguments, an interesting pattern

If you have never worked with languages that support closures you may find the following idiom too funky.

```
var myArray = ['first', 'second', 'third'];
myArray.each( function(item, index)
{
    alert('The item in the position #' + index + ' is:' + item);
});
```

Whoa! Let's explain what is going on here before you decide I've gone too far and navigate to a better article than this one.

First of all, in the above example we are using the prototype.js library, which adds the each function to the Array class. The each function accepts one argument that is a function object. This function, in turn, will be called once for each item in the array, passing two arguments when called, the item and the index for the current item. Let's call this function our iterator function. We could have also written the code like this.

```
function myIterator(item, index)
{
    alert('The item in the position #' + index + ' is:' + item);
}

var myArray = ['first', 'second', 'third'];
myArray.each( myIterator );
```

But then we would not be doing like all the cool kids in school, right? More seriously, though, this last format is simpler to understand but causes us to jump around in the code looking for the myIterator function. It's nice to have the logic of the iterator function right there in the same place it's called. Also, in this case, we will not need the iterator function anywhere else in our code, so we can transform it into an anonymous function without penalty.

18.1.7 This is `this` but sometimes `this` is also that

One of the most common troubles we have with JS when we start writing our code it the use of the `this` keyword. It could be a real tripwire.

As we mentioned before, a function is also an object in JS, and sometimes we do not notice that we are passing a function around.

Take this code snippet as an example.

```
function buttonClicked()
{
    alert('button ' + this.id + ' was clicked');
}

var myButton = document.getElementById('someButtonID');
var myButton2 = document.getElementById('someOtherButtonID');
myButton.onclick = buttonClicked;
myButton2.onclick = buttonClicked;
```

Because the buttonClicked function is defined outside any object we may tend to think the `this` keyword will contain a reference to the `window` or `document` object (assuming this code is in the middle of an HTML page viewed in a browser.)

But when we run this code we see that it works as intended and displays the `id` of the clicked button. What happened here is that we made the onclick method of each button contain the `buttonClicked` object reference, replacing whatever was there before. Now whenever the button is clicked, the browser will execute something similar to the following line.

```
myButton.onclick();
```

That isn't so confusing afterall, is it? But see what happens you start having other objects to deal with and you want to act on these object upon events like the button's click.

```
var myHelper =
{
```

```
    formFields: [ ],
    emptyAllFields: function()
    {
        for(i=0; i < this.formFields.length; i++)
        {
            var elementID = this.formFields[i];
            var field = document.getElementById(elementID);
            field.value = '';
        }
    }
};

//tell which form fields we want to work with
myHelper.formFields.push('txtName');
myHelper.formFields.push('txtEmail');
myHelper.formFields.push('txtAddress');

//clearing the text boxes:
myHelper.emptyAllFields();

var clearButton = document.getElementById('btnClear');
clearButton.onclick = myHelper.emptyAllFields;
```

So you think, nice, now I can click the Clear button on my page and those three text boxes will be emptied. Then you try clicking the button only to get a runtime error. The error will be related to (guess what?) the `this` keyword. The problem is that `this.formFields` is not defined if `this` contains a referece to the button, which is precisely what's happening. One quick solution would be to rewrite our last line of code.

```
clearButton.onclick = function()
{
    myHelper.emptyAllFields();
};
```

That way we create a brand new function that calls our helper method within the helper object's context.

18.2 Developer Notes for prototype.js

This guide is based on the Developer Notes for prototype.js by Sergio Pereira.

18.2.1 What is that?

In case you haven't already used it, prototype.js is a JavaScript library written by Sam Stephenson. This amazingly well thought and well written piece of standards-compliant code takes a lot of the burden associated with creating rich, highly interactive web pages that characterize the Web 2.0 off your back.

If you tried to use this library recently, you probably noticed that documentation is not one of its strongest points. As many other developers before me, I got my head around prototype.js by reading the source code and experimenting with it. I thought it would be nice to take notes while I learned and share with everybody else.

As you read the examples and the reference, developers familiar with the Ruby programming language will notice an intentional similarity between Ruby's built-in classes and many of the extensions implemented by this library.

18.2.2 Using the `$()` function

The `$()` function is a handy shortcut to the all-too-frequent `document.getElementById()` function of the DOM. Like the DOM function, this one returns the element that has the id passed as an argument.

Unlike the DOM function, though, this one goes further. You can pass more than one id and `$()` will return an `Array` object with all the requested elements. The example below should illustrate this.

```
<com:TClientScript UsingClientScripts="prado" />
<div id="myDiv">
    This is a paragraph
</div>

<div id="myOtherDiv">
```

```
    This is another paragraph
</div>

<input type="button" value=Test1 onclick="test1();" />
<input type="button" value=Test2 onclick="test2();" />

<script type="text/javascript">
/*<![CDATA[*/
function test1()
{
    var d = $('myDiv');
    alert(d.innerHTML);
}

function test2()
{
    var divs = $('myDiv','myOtherDiv');
    for(i=0; i<divs.length; i++)
    {
        alert(divs[i].innerHTML);
    }
}
/*]]>*/
</script>
```

Another nice thing about this function is that you can pass either the `id` string or the element object itself, which makes this function very useful when creating other functions that can also take either form of argument.

18.2.3 Using the `$F()` function

The `$F()` function is a another welcome shortcut. It returns the value of any field input control, like text boxes or drop-down lists. The function can take as argument either the element `id` or the element object itself.

```
<input type="text" id="userName" value="Joe Doe" />
<input type="button" value=Test3 onclick="test3();" />
```

```
<script type="text/javascript">
/*<![CDATA[*/
function test3()
{
    alert($F('userName'));
}
/*]]>*/
</script>
```

18.3 DOM Events and Javascript

18.3.1 Basic event handling

The syntax for working with events looks like the code below.

```
Event.observe(element, name, observer, [useCapture]);
```

Assuming for a moment that we want to observe when a link was clicked, we could do the following:

```
// <a id="clicker" href="http://foo.com">Click me</a>
Event.observe('clicker', 'click', function(event)
{
    alert('clicked!');
});
```

If we wanted to get the element that fired the event, we'd do this:

```
Event.observe('clicker', 'click', function(event)
{
    alert(Event.element(event));
});
```

18.3.2 Observing keystrokes

If we wanted to observe keystrokes for the entire document, we could do the following:

```
Event.observe(document, 'keypress', function(event)
{
    if(Event.keyCode(event) == Event.KEY_TAB)
        alert('Tab Pressed');
});
```

And lets say we wanted to keep track of what has been typed :

```
Event.observe('search', 'keypress', function(event)
{
    Element.update('search-results', $F(Event.element(event)));
});
```

Prototype defines properties inside the event object for some of the more common keys, so feel free to dig around in Prototype to see which ones those are.

A final note on keypress events; If you'd like to detect a left click you can use `Event.isLeftClick(event)`**.**

18.3.3 Getting the coordinates of the mouse pointer

Drag and drop, dynamic element resizing, games, and much more all require the ability to track the X and Y location of the mouse. Prototype makes this fairly simple. The code below tracks the X and Y position of the mouse and spits out those values into an input box named mouse.

```
Event.observe(document, 'mousemove', function(event)
{
    $('mouse').value = "X: " + Event.pointerX(event) +
                       "px Y: " + Event.pointerY(event) + "px";
});
```

If we wanted to observe the mouse location when it was hovering over a certain element, we'd just change the document argument to the id or element that was relevant.

18.3.4 Stopping Propagation

`Event.stop(event)` **will stop the propagation of an event .**

18.3.5 Events, Binding, and Objects

Everything has been fairly straight forward so far, but things start getting a little trickier when you need to work with events in and object-oriented environment. You have to deal with binding and funky looking syntax that might take a moment to get your head around.

Lets look at some code so you can get a better understanding of what I'm talking about.

```
EventDispenser = Class.create();
EventDispenser.prototype =
{
  initialize: function(list)
  {
    this.list = list;

    // Observe clicks on our list items
    $$(this.list + " li").each(function(item)
    {
        Event.observe(item, 'click', this.showTagName.bindEvent(this));
    }.bind(this));

    // Observe when a key on the keyboard is pressed.
    // In the observer, we check for
    // the tab key and alert a message if it is pressed.
    Event.observe(document, 'keypress', this.onKeyPress.bindEvent(this));

    // Observe our fake live search box.  When a user types
    // something into the box, the observer will take that
    // value(-1) and update our search-results div with it.
    Event.observe('search', 'keypress', this.onSearch.bindEvent(this));

    Event.observe(document, 'mousemove', this.onMouseMove.bindEvent(this));
  },
```

```
  // Arbitrary functions to respond to events
  showTagName: function(event)
  {
    alert(Event.element(event).tagName);
  },

  onKeyPress: function(event)
  {
    var code = event.keyCode;
    if(code == Event.KEY_TAB)
        alert('Tab key was pressed');
  },

  onSearch: function(event)
  {
    Element.update('search-results', $F(Event.element(event)));
  },

  onMouseMove: function(event)
  {
    $('mouse').value = "X: " + Event.pointerX(event) +
                "px Y: " + Event.pointerY(event) + "px";
  }
}
```

Whoa! What's going on here? Well, we've defined our a custom class `EventDispenser`. We're going to be using this class to setup events for our document. Most of this code is a rewrite of the code we looked at earlier except this time, we are working from inside an object.

Looking at the `initialize` method, we can really see how things are different now. Take a look at the code below:

```
// Observe clicks on our list items
$$(this.list + " li").each(function(item)
{
    Event.observe(item, 'click', this.showTagName.bindEvent(this));
}.bind(this));
```

We've got iterators, binding and all sorts of stuff going on. Lets break down what this chunk of code is doing.

First we are hunting for a collection of elements based on it's CSS selector. This uses the Prototype selector function `$$()`. After we've found the list items we are dealing with we send those into an each iteration where we will add our observers.

```
Event.observe(item, 'click', this.showTagName.bindEvent(this));
```

Now looking at the code above, you'll notice the `bindEvent` function. This takes the method before it `showTagName` and treats it as the method that will be triggered when, in this case, someone clicks one of our list items.

You'll also notice we pass this as an argument to the `bindEvent` function. This simply allows us to reference the object in context `EventDispenser` inside our function `showTagName(event)`. If the `showTagName` function requires additional parameters, you can attach them to the later parameters of `bindEvent`. For example

```
this.showTagName.bindEvent(this, param1, param2);

//where the showTagName function is defined as
showTime : function (event, param1, param2) { ... }
```

Moving on, you'll see `bind(this)` attached to our iterator function. This really has nothing to do with events, it is only here to allow me to use `this` inside the iterator. If we did not use `bind(this)`, I could not reference the method `showTagName` inside the iterator.

Ok, so we'll move on to looking at our methods that actually get called when an event occurs. Since we've been dealing with `showTagName`, lets look at it.

```
showTagName: function(event)
{
    alert(Event.element(event).tagName);
}
```

As you can see, this function accepts one argument–the event. In order for us to get the element which fired the event we need to pass that argument to `Event.element`. Now we can manipulate it at will.

This covers the most confusing parts of our code. The text above is also relevant to the remaining parts of our code. If there is anything about this you don't understand, feel free to ask questions in the forum.

18.3.6 Removing Event Listeners

This one threw me for a loop the first time I tried to use it. I tried something similar to what I did in the `Event.observe` call with the exception of using `stopObserving`, but nothing seemed to change. In other words, the code below does NOT work.

```
$$(this.list + " li").each(function(item)
{
    Event.stopObserving(item, 'click', this.showTagName);
}.bind(this));
```

What's the deal here? The reason this does not work is because there is no pointer to the observer. This means that when we passed `this.showTagName` in the `Event.observe` method before hand, we passed it as an anonymous function. We can't reference an anonymous function because it simply does not have a pointer.

So how do we get the job done? All we need to do is give the observing function a pointer, or the jargon free version: Set a variable that points to `this.showTagName`. Ok, lets change our code a bit.

```
this.showTagObserver = this.showTagName.bindEvent(this);

// Observe clicks on our list items
$$(this.list + " li").each(function(item)
{
    Event.observe(item, 'click', this.showTagObserver);
}.bind(this));
```

Now we can remove the event listeners from our list like this:

```
$$(this.list + " li").each(function(item)
{
    Event.stopObserving(item, 'click', this.showTagObserver);
}.bind(this));
```

18.4 Javascript in PRADO, Questions and Answers

18.4.1 How do I include the Javascript libraries distributed with Prado?

The javascript libraries distributed with Prado can be found in the `framework/Web/Javascripts/source` directory. The `packages.php` file in that directory defines a list of available package names available to be loaded. They can be loaded as follows.

- **Adding libraries in the template**

```
<com:TClientScript PradoScripts="effects" />
```

- **Adding libraries in PHP code**

```
$this->getPage()->getClientScript()->registerPradoScript("effects");
```

The available packaged libraries included in Prado are

- `prado` **: basic PRADO javascript framework based on Prototype**
- `effects` **: visual effects from script.aculo.us**
- `ajax` **: ajax and callback related based on Prototype**
- `validator` **: validation**
- `logger` **: javascript logger and object browser**
- `datepicker` **: datepicker**
- `colorpicker` **: colorpicker**

The dependencies for each library are automatically resolved. Components that require a particular library will also automatically load the necessary libraries. For example, if you add a `TDatePicker` component on the page, the `datepicker` and its dependencies will be automatically included on the page.

See TClientScript for options of adding your custom Javascript code to the page.

18.4.2 Publishing Javascript Libraries as Assets

Use TClientScriptLoader to publish and combine multiple existing javascript files (e.g. javascript libraries distributed with Prado or otherwise) as packages. For greater control on what and when to publish, use the `registerJavascriptPackages($base, $packages, $debug=null, $gzip=true)` method in the `TClientScriptManager` class, which an instance can be obtained using `$this->getPage()->getClientScript()` or its equivalents. For example, if multiple controls will use the same set of javascript libraries, write a class to handle the registration of packages required by those controls.

```
class MyJavascriptLib extends TComponent
{
    private $_packages=array(); //keep track of all registrations

    private $_manager;

    protected function __construct(TPage $owner)
    {
        $this->_manager = $owner->getClientScript();
        $owner->onPreRenderComplete = array($this, 'registerScriptLoader');
    }

    public static function registerPackage(TControl $control, $name)
    {
        static $instance;
        if($instance===null)
            $instance=new self($control->getPage());
        $instance->_packages[$name]=true;
    }

    protected function registerScriptLoader()
    {
        $dir = dirname(__FILE__).'/myscripts'; //contains my javascript files
        $scripts = array_keys($this->_packages);
        $url = $this->_manager->registerJavascriptPackages($dir, $scripts);
        $this->_manager->registerScriptFile($url,$url);
    }
}
```

```
// example control class using the javascript packages
class TestComp extends TControl
{
    public function onPreRender($param)
    {
        parent::onPreRender($param);
        MyJavascriptLib::registerPackage($this,'package1');
    }
}
```