

Table of Contents

Course: Programming with Sitellite	1
1. <u>Syllabus</u>	1
<u>Course instructor</u>	1
<u>Difficulty level and prerequisites</u>	1
<u>Course description</u>	1
<u>Course objective</u>	1
<u>Course outline</u>	1
2. <u>What is Sitellite?</u>	2
<u>Main components</u>	3
<u>App model</u>	3
3. <u>Hello World!: A first Sitellite application</u>	4
<u>What is a box?</u>	4
<u>Creating a box</u>	4
<u>Creating a template</u>	4
<u>Setting the access rights</u>	5
<u>Viewing the results</u>	5
<u>More advanced box features</u>	5
4. <u>Forms in Sitellite</u>	9
<u>MailForm package</u>	9
<u>Contact US: A first form in Sitellite</u>	9
5. <u>Object–Oriented Programming (OOP) in Sitellite</u>	12
<u>Loading SAF classes</u>	12
<u>Writing new classes</u>	12
<u>Loading your own classes</u>	13
<u>The Generic class</u>	13
6. <u>Sitellite collections</u>	15
<u>What is a collection?</u>	15
<u>Collections in Sitellite</u>	15
<u>Creating a collection</u>	15
<u>The collection API: Rex</u>	18
7. <u>SimpleTemplate in depth</u>	19
<u>The basics</u>	19
<u>Loops</u>	19
<u>Conditions</u>	19
<u>Filters</u>	20
<u>Additional tags</u>	20
8. <u>How to get help</u>	21
<u>Sitellite.org</u>	21
<u>Simian.ca</u>	21

Course: Programming with Sitellite

1. Syllabus

Course instructor

Name John Luxford

Email lux@simian.ca

Web Site <http://www.simian.ca/>

Difficulty level and prerequisites

This course assumes a comfortable level of familiarity with the PHP programming language or equivalent (Java, ASP, Perl, etc.), and is intended for web development professionals.

Course description

An introduction and overview to developing web-based applications using Sitellite platform.

Course objective

To familiarize the student with the various components and concepts of the Sitellite application framework and content server, as well as to provide the steps necessary to begin writing applications using them.

Course outline

1. Syllabus
 1. Course instructor
 2. Difficulty and prerequisites
 3. Course description
 4. Course objectives
 5. Course outline
2. What is Sitellite?
 1. Main components
 1. Sitellite application framework (SAF)
 2. Content server
 3. Content manager
 2. App model
 1. Model-View-Controller (MVC) design pattern
 2. App directory layout
3. Hello World!: A first Sitellite application
 1. What is a box?
 2. Creating a box
 3. Creating a template
 4. Setting the access rights
 5. Viewing the results
 6. More advanced box features

1. Parameter list
2. Accessing other global objects
3. Setting page properties
4. Forms in Sitellite
 1. MailForm package
 2. Contact Us: A first form in Sitellite
 1. The settings file
 2. The index file
 3. Setting the access rights
 4. Viewing the results
5. Object–Oriented Programming (OOP) in Sitellite
 1. Loading SAF classes
 2. Writing new classes
 3. Loading your own classes
 4. The Generic class
 1. Creating a database table
 2. Updating it with Generic
6. Sitelite collections
 1. What is a collection?
 2. Collections in Sitelite
 3. Creating a collection
 1. The collection definition file
 1. General settings
 2. Browse options
 3. Facets
 4. Type hints
 2. The versioning table schema
 4. The collection API: Rex
7. SimpleTemplate in depth
 1. The basics
 2. Loops
 3. Conditions
 4. Filters
 5. Additional tags
 1. {exec} and {php}
 2. {intl}
 3. {box}
 4. {alt}
8. How to get help
 1. Sitelite.org
 2. Simian.ca

2. What is Sitelite?

Sitelite is a web site content management system (CMS), which means that it provides the means for creating, publishing, and managing content on a web site. A CMS provides interfaces for the various roles in a web site project to come together, including the programmers, designers, administrators, and the content creators.

Main components

Sitellite application framework (SAF)

At the core of Sitellite is a collection of over 100 PHP classes that make up the core of the application itself. These classes provide reusable, generic components for building any type of web application.

Content server

The content server is the publishing component of the CMS, which controls access to content such as web pages or news stories, as well as to applications, called apps, written on top of the Sitellite framework.

Content manager

The content manager is the user interface of Sitellite used by the editors and administrators to update the web site content and authentication rules, as well as to access additional apps written on top of the Sitellite framework.

App model

Sitellite is a server-side platform for deploying web-based applications. It can simultaneously deploy any number of applications, and also allows a high level of interaction and compatibility between applications.

This application server model makes Sitellite distinct as a CMS.

Model-View-Controller (MVC) design pattern

Sitellite employs the MVC design pattern as the basis for its app model. The MVC software design pattern aims to separate the various components of a given piece of software into distinct entities, which can be controlled and managed separately and predictably. To understand this better, it would be best to explain the MVC terminology.

Model	The model is the data source of the application. In Sitellite, usually this is the underlying MySQL database. The model also includes the abstraction layer used to access the data source, which is usually written as a set of PHP classes.
View	The view is the visual representation of the application, the graphical user interface (GUI). In Sitellite, this is stored in the form of templates.
Controller	The controller accepts visitor requests and determines which content from the model to join with which view in order to satisfy the visitor request. This is the content server within Sitellite, and is also extended in Sitellite via the box and form concepts, which I'll explain shortly.

App directory layout

Since each app in Sitellite is a distinct entity that is deployed alongside other apps, each app has its own standard directory layout where the various components of an app are placed.

Each app is a separate directory within the "inc/app" directory of Sitellite, and has the following sub-directories:

boxes	Contains all of the "boxes" of an app. Boxes are simply Sitellite–managed PHP scripts.
conf	Contains all of the configuration and settings for the app.
data	Contains any filesystem–bound data storage for the app.
docs	Contains the help files and the API documentation for the app.
forms	Contains all of the forms of an app. Forms are like boxes except that they render and process HTML forms.
html	Contains all of the templates for an app.
install	Contains any installation files, such as the custom database schema, for an app.
lib	Contains all of the PHP classes used by an app.
pix	Contains any images used by an app.

3. Hello World!: A first Sitellite application

What is a box?

For the most part, a box is simply a PHP script. However, boxes also contain a number of additional features by being part of the Sitellite system, including access control, interoperability with other parts of Sitellite, and documentation capabilities.

Creating a box

We'll start with the PHP code for our box:

```
<?php
$data = array ();
$data['message'] = 'Hello World!';
echo template_simple ('hello.spt', $data);
?>
```

For our testing, let's save this to a file named "inc/app/myapp/boxes/hello/index.php". You'll need to create the missing directories as well.

Exercise: Directory layout

Create the rest of the directories that make up the "myapp" Sitellite app.

Creating a template

Next, we need to create the "hello.spt" template which is referenced in our box. Save the following template into a file called "inc/app/myapp/html/hello.spt":

```
<p>{message}</p>
```

As you can see from the box code we just wrote, references to templates can omit the path, since Sitellite knows to look in the current app for the specified template.

Setting the access rights

We're going to set some access rights on our newly created box so that we can reference it from the URL. Save the following to a file named "inc/app/myapp/boxes/hello/access.php":

```
sitellite_access = public
sitellite_status = approved
sitellite_action = on
```

This tells Sitellite which access level and status the box is accessible for. It also tells Sitellite that the box should be allowed to be loaded from the "action" context, which means that we can call it directly via a special URL in our web browser.

For a complete list, please visit the documentation links at the bottom of this outline.

Viewing the results

Open a web browser and point it to the following URL:

```
http://www.example.com/index/myapp-hello-action
```

Be sure to replace "www.example.com" with the appropriate web site address that you have installed Sitelite to. You should now see the output of your first Sitelite box displayed in the default template of Sitelite.

More advanced box features

Parameter list

Many times you need your PHP scripts to input data to you for processing, or even to determine what item or group of items is being requested. For example, a news story script would need to know the ID of the news story you want to read.

In Sitelite, these are called parameters and they are passed automatically to every script as a key/value array called `$parameters`.

Exercise: Parameters

Change our box code to the following:

```
<?php
if (not empty ($parameters['message'])) {
    $data = $parameters;
} else {
    $data = array ();
}
```

```

        $data['message'] = 'Hello World!';
    }
    echo template_simple ('hello.spt', $data);
?>

```

Now you can try changing the message that is displayed by adding the following to the end of the URL that we used to view our box:

```
?message=Hello+Joe!
```

Refresh the browser window and you should see your "Hello World!" change to say "Hello Joe!". That's all there is to using parameters.

Accessing other global objects

Sitellite includes many built-in objects that can help save you time coding specific types of features into your app. These include:

- `$cgi` Contains the GET or POST variables sent to the current box or form. Also allows you to do input validation on any user-submitted data.
- `$cookie` Contains any browser cookies for the current user. Allows you to view and set cookies for the user.
- `$db` A database connection object. This allows you to send and receive data from the underlying MySQL database.
- `$intl` A language translation object. This allows you to create multi-lingual features in your app.
- `$loader` A class and box loading object. This allows you to import new classes from SAF or from any app, and also allows you to call boxes and forms from one another.
- `$menu` Contains a tree structure of the web pages of the site. Can be used to display dynamic menus.
- `$page` Contains the current web page data. Allows you to view and set new web page properties.
- `$session` A session authentication object. This contains the current user's profile and access rights. You can also use this object to view and set temporary data pertaining to the current user, such as a list of shopping cart items.
- `$site` Contains the directory and URL information for the web site and the current web page request.

Some of these objects provide functions to access them, such as `$db`, `$page`, and `$session`. Others have to be imported into the box's namespace before they can be used, since boxes are executed in their own namespace separate from the global one.

Exercise: Accessing global objects

Change our box code to the following:

```
<?php
```

```

if (not empty ($parameters['message'])) {
    $data = $parameters;
} else {
    $data = array ();
    $data['message'] = 'Hello World!';
}

$messages = session_get ('messages');
if (! is_array ($messages)) {
    $messages = array ();
}
$messages[] = $data['message'];
session_set ('messages', $messages);

$data['messages'] = $messages;

echo template_simple ('hello.spt', $data);

?>

```

We'll also change our hello.spt template to say the following now:

```

<p>{message}</p>

<p>Previous messages:</p>

<ul>
    {loop obj[messages]}
        <li>{loop/_value}</li>
    {end loop}
</ul>

```

If you refresh the browser window again, you should see the history of all of the messages set from this point on. Try changing the ?message= value again to see it working.

Setting page properties

Using the global \$page object, you can affect the other properties of the web page than just its body. This includes its title, metadata, and anything else about it.

Exercise: Setting the page title

Change our box code to the following:

```

<?php
if (not empty ($parameters['message'])) {
    $data = $parameters;
} else {
    $data = array ();

```

More advanced box features

```

        $data['message'] = 'Hello World!';
    }

    $messages = session_get ('messages');
    if (! is_array ($messages)) {
        $messages = array ();
    }
    $messages[] = $data['message'];
    session_set ('messages', $messages);

    $data['messages'] = $messages;

    page_title ('Hello');

    echo template_simple ('hello.spt', $data);

?>

```

If you refresh the browser window again, you should see the page title being set to "Hello" where before it was blank.

The other properties of a web page that can be altered include:

- `page_add_header()` Adds an HTTP header to send with the page.
- `page_add_link()` Adds an HTML link tag to the page.
- `page_add_meta()` Adds an HTML meta tag to the page.
- `page_add_style()` Adds a CSS stylesheet to the page.
- `page_add_script()` Adds a JavaScript script to the page.
- `page_below()` Sets the `below_page` property of the page. This sets the position of the page within the web site.
- `page_description()` Sets the description of the page.
- `page_id()` Sets the page ID. Handy for boxes with "fake" pages that alias them.
- `page_keywords()` Sets the keywords of the page.
- `page_onblur()` Adds an onblur handler to the page.
- `page_onclick()` Adds an onclick handler to the page.
- `page_onfocus()` Adds an onfocus handler to the page.
- `page_onload()` Adds an onload handler to the page.
- `page_onunload()` Adds an onunload handler to the page.
- `page_template()` Sets the template used to render the page.
- `page_template_set()` Sets the template set used to render the page.

page_title() Sets the title of the page.

4. Forms in Sitellite

MailForm package

Form capabilities in Sitellite are provided by the MailForm package in SAF. MailForm handles generating forms based on simple INI files and PHP scripts, validating form input using a simple but powerful validation rule syntax, and form processing using built-in handlers or custom ones.

Contact US: A first form in Sitellite

The settings file

The following is a simple form definition in INI format.

```
[Form]

message = Please use this form to contact us.

[name]

type = text
alt = Your Name

[email]

type = text
alt = Email Address
rule 1 = "not empty, You must enter your email address."
rule 2 = "contains `@`, Your email address does not appear to be valid."

[message]

type = textarea
alt = Your Message
labelPosition = left

[submit_button]

type = submit
setValues = Send
```

Save this to a file named "inc/app/myapp/forms/contact/settings.php". You'll also need to create any nonexistent directories in that path.

Some explanations of the different aspects of this file:

```
[Form]

message = Please use this form to contact us.
```

The file is organized into sections which begin with "[section_name]" and contain a series of "key = value" lines. Each form definition begins with a [Form] section for settings pertaining to the form itself, followed by

a series of sections which correspond to the fields (called "widgets" in Sitellite) shown in the form.

```
[name]

type = text
alt = Your Name
```

This tells Sitellite to create a text box named "name" with the label "Your Name" beside it, roughly like this:

```
Your Name <input type="text" name="name" />

rule 1 = "not empty, You must enter your email address."
rule 2 = "contains `@`, Your email address does not appear to be valid."
```

Rules are specified one per line and are numbered. You can have as many rules as you need for each form widget.

For complete information about form rules, widgets, and which properties they have, please see the references at the bottom of this outline.

The index file

Next we're going to plug our form definition into Sitellite via a PHP script that handles and controls the form.

```
<?php

class MyAppContactForm extends MailForm {
    function MyAppContactForm () {
        parent::MailForm ();

        $this->parseSettings (
            'inc/app/myapp/forms/contact/settings.php'
        );
    }

    function onSubmit ($vals) {
    }
}

?>
```

Save this file to "inc/app/myapp/forms/contact/index.php". I'll explain what each part does for you below:

```
class MyAppContactForm extends MailForm {
```

This begins our form object, which inherits all of its form capabilities from the MailForm package. Because Sitellite know this is a form, we don't have to explicitly load the MailForm package ourselves.

The name of the form object begins with the name of the app capitalized, followed by the name of each folder under the "forms" folder (in this case just "contact") also capitalized, followed by the word "Form" which distinguishes the forms purpose for Sitellite.

```
function MyAppContactForm () {
    parent::MailForm ();
```

This begins the object constructor method and calls the constructor method of the parent class as well. The constructor method of a class is called to initialize new objects created from that class.

```
$this->parseSettings (
    'inc/app/myapp/forms/contact/settings.php'
);
```

This line takes the settings.php file that we created and turns it into a series of "widget" objects, which will generate and handle the different fields of the form.

```
function onSubmit ($vals) {
}
```

This method is called when the form has been submitted and all of the validation rules have passed.

Exercise: Mailing the form results

Fill in the onSubmit() method with the following PHP code:

```
if (! @mail (
    'webmaster@example.com',
    'Contact Form',
    template_simple (
        'contact_email.spt',
        $vals
    ),
    'From: ' . $vals['name'] . ' <' . $vals['email'] . '>'
)) {
    page_title ('An Error Occurred');
    echo 'The message was unable to be sent. Please try again later.';
}

page_title ('Thank You');
echo 'Your message has been sent.';
```

This piece of code tells PHP to take the form results (the \$vals array) and send them to the specified email address. Be sure to replace "webmaster@example.com" with your own email address in order to receive the messages sent by the form.

For more information about the PHP mail() function, see <http://www.php.net/mail>

Setting the access rights

Copy the access.php file from the hello box we created and place it into the form folder with the index.php and settings.php files. The access information will be the same for our form as they were for the box.

Viewing the results

We're now ready to view the form we just created. Go to the following URL:

```
http://www.example.com/index/myapp-contact-form
```

You should see the form appear. Feel free to try it out and see the results. Try to play with the input validation rules we had set for the email address.

5. Object–Oriented Programming (OOP) in Sitellite

When writing applications in any language, it's easy to amass a large number of scripts very quickly that often have duplicate bits of code that would be better placed in a single, shared location. To solve this, we use PHP classes to abstract repetitious tasks so that we only write them once, then use the class in each place that the task is needed.

Loading SAF classes

Sitellite provides a convenient function for importing classes from the SAF library. Let's create a new box and play around with the "saf.HTML" package to illustrate this:

```
<?php

loader_import ('saf.HTML');

echo html::p (
    html::strong ('Class Test'),
    array ('style' => 'text-align: center')
);

?>
```

Save this to "inc/app/myapp/boxes/classtest/index.php". Also make sure to copy the access.php file into the new box folder from the "hello" box.

To load this box in your browser, go to:

```
http://www.example.com/index/myapp-classtest-action
```

Writing new classes

Writing your own classes in Sitellite is the same as in any PHP application:

```
<?php

class MyList {
    var $list = array ();

    function get ($name) {
        return $this->list[$name];
    }

    function set ($name, $value) {
        $this->list[$name] = $value;
    }
}

?>
```

In Sitellite, we save classes to the "lib" folder of our apps. Save this class to "inc/app/myapp/lib/MyList.php".

Loading your own classes

Now let's call our class instead of "saf.HTML". Replace the PHP code from our "classtest" box with the following:

```
<?php
loader_import ('myapp.MyList');

$list = new MyList ();

$list->set ('foo', 'bar');

echo $list->get ('foo');

?>
```

As you can see, the loader_import() function even knows about your app, without you having to tell it a thing. Similarly, you can call classes defined in any other app. That ease of programming is one of the benefits of the Sitellite platform.

The Generic class

The Generic class is part of the "saf.Database" package. It provides a set of generic accessor methods for reading and writing to database tables, which can drastically reduce the amount of effort required to write database abstraction objects, which is one of the most frequent uses of objects and classes.

Creating a database table

To illustrate the power of the Generic class, we need to start with a database table. Log into Sitellite and go to the "Control Panel" option in the top bar. Next, select the "DB Manager" under "Tools" and click on the "SQL Shell" link just under the screen title. This should pull up a text area into which you can enter the following SQL schema:

```
create table myapp_listing (
    id int not null auto_increment,
    name char(48) not null,
    description text,
    primary key (id),
    index (name)
);
```

Updating it with Generic

We're now ready to make use of the Generic class. Enter the following into a new text file and save it to "inc/app/myapp/lib/Listing.php":

```
<?php
loader_import ('saf.Database.Generic');

class MyappListing extends Generic {
    function MyappListing () {
        parent::Generic ('myapp_listing', 'id');
    }
}
```

```
}  
?>
```

Here we've imported the Generic class and then extended it in the same way we did with MailForm. Generic now knows all it needs to start managing our table for us. Now let's overwrite our "classtest" box code with the following:

```
<?php  
  
loader_import ('myapp.Listing');  
  
$listing = new MyAppListing ();  
  
// add a few listings  
  
if (! $listing->add (array (  
    'name' => 'Listing One',  
    'description' => 'This is a description of our listing.',  
))) {  
    die ($listing->error);  
}  
  
$listing->add (array (  
    'name' => 'Lstng Tw',  
    'description' => 'This is a description of our listing.',  
));  
  
$listing->add (array (  
    'name' => 'Listing Three',  
    'description' => 'This is a description of our listing.',  
));  
  
// update listing two (to correct our spelling)  
  
$listing->modify (2, array ('name' => 'Listing Two'));  
  
// display all of the listings  
  
echo template_simple (  
    'listings.spt',  
    $listing->find (array ())  
);  
  
// delete them all  
  
$listing->remove (array ('1=1'));  
?>
```

As you can see, Generic has done a lot of the work for us. Now all we have to do is pass our MyAppListing object the details and tell it *what* to do, instead of worrying about *how* to do it.

Before we preview this, let's create a new template called "listings.spt" that we referenced in the above code:

```
{loop obj}  
<p>  
    <strong>{loop/name}</strong><br />  
    {description}  
</p>
```

```
{end loop}
```

Now let's refresh our browser window to see it all come together.

6. Sitellite collections

What is a collection?

Content that is managed by Sitellite in the standard manner, including web pages, news stories, event listings, and others, is organized into what we call "collections". A collection is a group of items of the same type (ie. all of the news stories make up the "News Stories" collection).

Collections in Sitellite

When a database table or file structure becomes a collection, it adopts a number of benefits over the traditional ways of managing items in a database or filesystem. These include:

- Versioning – Keeps track of every change made to each item, and allows you to view and even revert to previous versions of an item — even after it's been deleted.
- Audit trail – Keeps track of *who* made what change and *when* it was made.
- Standardized editing – Collections become available through both the Sitellite Web View as well as the collection list in the Sitellite Control Panel, reducing the amount of learning involved in training existing editors for new content types.
- Standard API – Sitellite provides a built-in set of classes for accessing any collection programmatically, creating a standard way of programming for any type of collection.

Creating a collection

The collection definition file

A collection starts with two things: A data source (usually a database table) and a collection definition file. The collection definitions are stored as a series of INI files in the "inc/app/cms/conf/collections" directory.

Let's build a simple collection definition for the myapp_listing database table we created above. To start, create a blank file named "inc/app/cms/conf/collections/myapp_listing.php".

General settings

The first three blocks of the collection definition are [Collection], [Source], and [Store], which provide details about the collection itself, and which "source" and "store" drivers are being used.

Source means the data source being managed as a collection. In our example, this is the original myapp_listing database table.

Store means the location used to store the version changes to items in the collection. In our example, this will be a secondary database table that we will create shortly.

The reason for abstracting the two into separate components is so that, if needed, you can change the underlying data source or version storage layers (ie. changing the store from a database table to using the Concurrent Versioning System (CVS) application to manage versioning).

The following will be the general section of our collection definition file:

```
; <?php /*  
  
[Collection]  
  
name                = myapp_listing  
display             = Listings  
singular            = Listing  
key_field           = id  
title_field         = name  
title_field_name= Name  
summary_field      = description  
body_field          = description  
  
[Source]  
  
name = Database  
  
[Store]  
  
name = Database
```

Browse options

Browse options are used to determine which fields should be shown as table columns in the Sitellite Control Panel's collection browsing screen. For this, we'll show all three fields by adding the following to our collection definition file:

```
[browse:id]  
  
header = ID  
width = "10%"  
  
[browse:name]  
  
header = Name  
width = "25%"  
length = 30  
  
[browse:description]  
  
header = Description  
width = "55%"  
length = 80
```

This sets the column width for each field and the maximum text length for each text field.

Facets

Sitellite's browse screen has a powerful feature called "facets" which is shown as the "Search Parameters" box at the top of the screen. This allows users to perform compound queries on the collection data in just a few clicks. For example, an editor might want to search for all of the news stories under the "Sports" category that have a status of "Pending". These types of queries can be done with no technical knowledge at all in Sitellite. They can even be bookmarked for easy access from the main screen of the Control Panel, so a user only has to enter the query once.

For our three-field collection, we'll define two facets which will both be text boxes, one for entering the literal ID of the listing, and the other for searching both the name and description fields. To do this, add the following to your collection definition:

```
[facet:id]

display = ID
type = text
fields = id

[facet:name]

display = Text
type = text
fields = name, description
```

Type hints

Type hints tell the default Sitellite add and edit forms how to treat each field when shown as a MailForm widget. Using these you can change labels, widget types, and add validation rules to each field. For ours, we'll simply change the label position of the description field and add a rule that the name can't be empty.

```
[hint:name]

rule 1 = not empty, You must enter a name for your listing.

[hint:description]

labelPosition = left
```

The reason we don't have to specify everything explicitly is because Sitellite knows how to determine certain characteristics from the underlying source driver in order to properly display many common types of fields.

Now that we've finished our collection definition, let's add one more line to the bottom of the file:

```
; */ ?>
```

This and the first line help protect the file from potential hackers by fooling the web server into treating it as if it was an ordinary PHP script, which hides its contents from the visitor.

The versioning table schema

The last step before we can view our collection in action is to define the database schema for the versioning table. In the SQL Shell of the DB Manager tool, enter the following:

```
create table myapp_listing_sv (
  sv_autoid int(11) not null auto_increment,
  sv_author varchar(48) not null default '',
  sv_action enum(
    'created','modified','republished','replaced','restored','deleted'
  ) not null default 'created',
  sv_revision timestamp(14) not null,
  sv_changelog text not null,
  sv_deleted enum('yes','no') not null default 'no',
  sv_current enum('yes','no') not null default 'yes',
  id int not null,
  name char(48) not null,
```

```

description text not null,
primary key (sv_autoid),
index (sv_author,sv_action,sv_revision,sv_deleted,sv_current),
index (id)
);

```

This table is based on our original table with several changes. These are:

- The table name has an "_sv" at the end, which stands for Sitellite Versioning.
- There are 7 new fields added to the start of the table, each beginning with the "sv_" prefix. These are:
 - ◆ sv_autoid – The unique version number
 - ◆ sv_author – The author of the change
 - ◆ sv_action – The type of change made
 - ◆ sv_revision – The date and time of the change
 - ◆ sv_changelog – A summary of the changes made, provided by the author of the change
 - ◆ sv_deleted – Whether this item has been deleted or not
 - ◆ sv_current – Whether this version is the most current for this item
- The primary key is now the "sv_autoid" field, and the "id" field is only indexed like any other.
- The other "sv_" fields are also indexed, since they are made frequent use of.

Now we're ready to view our collection in action. Go to the Sitellite Control Panel and look for the "Listings" collection under the top-left pane. This will take you to the browse view for your new collection. Feel free to add, edit, search, and delete items from the collection.

The collection API: Rex

In the same way that we used Generic to access simple database tables, we can use the Rex package to access collections. Create another box named "rextest" and copy the access.php file from one of our other boxes into it. Then create an index.php file for the new box with the following contents:

```

<

loader_import ('cms.Versioning.Rex');

$rex = new Rex ('myapp_listing');

$id = $rex->create (array ('name' => 'Test', 'description' => 'Test'));

echo template_simple (
    'listings.spt',
    $rex->getList ()
);

$rex->delete ($id);

?>

```

If you run this box via /index/myapp-rextest-action, you should see the test listing displayed using the listings.spt template from our previous example. As you can see, Rex makes it as easy as Generic did to access Sitellite-managed collections.

7. SimpleTemplate in depth

SimpleTemplate is one of two template engines in Sitellite. The other, XT, is an XML-based template language used to render the design templates for the web site. SimpleTemplate, on the other hand, is used for smaller output, such as the formatting of search results or news stories. The reason for the two, instead of the usual one-size-fits-all template language which is what most frameworks offer, is that we were able to better tailor each for its intended use.

SimpleTemplate allows you to create small templates fast. Its syntax is not overly complex, and it has none of the XML verbosity that XT does. This makes it ideal for programmers to use to format the output of database queries and other box tasks.

The basics

As we've seen from the above examples, SimpleTemplate templates are saved to ".spt" files in the "html" folder of each app.

The basic tags that can be used in the template correspond to the key names or property names of the associative array or object passed to the `template_simple()` function. So if the array contains the keys "id", "name", and "description", then the main tags are `{id}`, `{name}`, and `{description}`.

Loops

To loop in SimpleTemplate, we use the `{loop expression}...{end loop}` syntax. An example of this is:

```
<ul>
{loop obj[items]}
  <li>{loop/_value}</li>
{end loop}
</ul>
```

The "obj[items]" means "loop through the array stored in the 'items' key of the array passed to the template". "obj" represents the array or object passed to the template. Similarly, if we passed an object to the template, "obj[items]" would become "obj.items". These are both a shorthand way of saying "\$obj['items']" and "\$obj->items", respectively, which makes it easier to read within a template.

We refer to the current loop item via `{loop/...}` where "..." is replaced with some property of the current loop item. "_value" is one of several properties available in any SimpleTemplate loop, because they are created by SimpleTemplate itself. These also include `_key`, `_index`, `_total`, and `_properties`.

Conditions

Conditional execution can be done in SimpleTemplate via the `{if expression}...{end if}` syntax. This syntax is illustrated in the following two examples:

```
{if obj[some_value]}
  {some_value}
{end if}
```

This condition checks for the "some_value" property before attempting to output it. The next example outputs something else if the condition evaluates to false.

```
{if obj[some_value]}
    {some_value}
{end if}
{if else}
    No value.
{end if}
```

The `{if else}` tag tells SimpleTemplate to match the opposite of the previous `{if expression}` tag. There is no "else if" tag.

Filters

By default, SimpleTemplate tags that output values pass those values to the `htmlentities_compat()` function defined in `saf.Functions` before producing them. This is called filtering of output, and provides an added security measure for Sitellite developers, reducing the possibility of accidental cross-site scripting attacks.

The function used can be changed to use an alternate function, or to use no function at all, via the following syntax:

```
Default:
{some_value}
```

```
Using an alternate function:
{filter strtoupper}{some_value}{end filter}
```

```
Using no filter:
{filter none}{some_value}{end filter}
```

The `{end filter}` tag returns the filter to the default `htmlentities_compat()` function.

Additional tags

`{exec}` and `{php}`

To output a PHP expression, which can be extremely useful for testing, use the `{php}` tag:

```
{php obj[some_value]}
```

Alternately, to output no value but still execute the expression, which can be useful for many simple inline PHP statements, use the `{exec}` tag:

```
{exec obj[some_value] += 1}
```

`{intl}`

To output text within a template in multiple languages, use the `{intl}` tag, which makes it translatable via Sitellite's translation capabilities. For example:

```
{intl This text will be translated.} This text won't be.
```

{box}

Just like you can import boxes from other boxes, you can also call boxes from directly within a SimpleTemplate file. For example:

```
{box sitellite/nav/breadcrumb}
```

And to pass parameters to a box, SimpleTemplate uses a URL-like scheme:

```
{box myapp/hello?message=Hi Bob}
```

{alt}

One final, handy tag is the {alt} tag. This tag creates an object which alternates between a number of values, which is most often useful for alternating the background colour of rows within a data table. For example:

```
{alt #fff #eee}

<table cellpadding="3" cellspacing="1">
  {loop obj[items]}
    <tr style="background: {alt/next}">
      <td>{loop/_value}</td>
    </tr>
  {end loop}
</table>
```

8. How to get help

Sitellite.org

- The complete Sitellite documentation:
<http://www.sitellite.org/index/docs>
- The complete Sitellite API reference:
<http://www.sitellite.org/docs>
- Community forum:
<http://www.sitellite.org/index/siteforum-app>

Simian.ca

- Simian contact information:
<http://www.simian.ca/index/contact>
- Technical support page:
<http://www.simian.ca/index/support>

Copyright © 2004 Simian Systems Inc.
All rights reserved.