Agavi tutorial

[vertical list of authors]

[Michael Stolovitzsky <michael.stolovitzsky@bitextender.com>]

[Noah Fontes <noah.fontes@bitextender.com>]

[Felix Gilcher <felix.gilcher@bitextender.com>]

[cover art/text goes here]

Contents

Introduction 3
- About Agavi 3
- MVC in Agavi 4
- Overview of Agavi services 5
- Overview of application execution flow 6
- A Word About Actions 8
- Application filesystem layout 8
- Overview of application configuration 9

Setting up the initial application 11
- Installing Agavi 11
- Creating an Agavi project 13
- Finishing the setup 14
- Finishing the basic setup 15
- Installing a new copy of your application 16

Adding first code 17
- Creating a new module 17
- Creating a new action 18
- Tying things together - an introduction to routing 19
- Fixing the bloggie routing 22
- Accessing Request Parameters and Validation Basics 23
- Handling validation errors 24

Putting the M in MVC 27
- Creating a new Model 27
- Adapting the Actions and Views 30
- Custom validators 31

Polishing it up 34
- Layers and Layouts 34
- Applying our Layout 36
- What are Slots 38
- Adding the post's title to the url 45
- Routing Callbacks 46
- Using Callbacks for the Title in URLs 47

Connecting to a database 50
- The Database Manager 50

Handling multiple Representations 52
- Output Types 52
- Exception Templates 54
- Generating an RSS Feed 54

Agavi tutorialMichael Stolovitzsky <michael.stolovitzsky@bitextender.com>Noah Fontes <noah.fontes@bitextender.com>Felix Gilcher <felix.gilcher@bitextender.com>

Introduction

About this guide

This guide provides a comprehensive overview of the architecture of the Agavi application framework, examining various common Web application problems and how Agavi addresses them. It does not cover the api documentation which can be found in a separate section of the website.

During the course we will cover the complete development cycle of a PHP blog engine, from a basic skeleton application to a working and themed implementation. It is broken down into stages: each chapter involves the creation of a partial implementation, so you can see the blog software in various stages of development and make comparisons. The stages are available as tarballs.

The final version of the blog engine is also the official demo application of Agavi. It is extremely well documented and showcases many of the features Agavi has to offer.

Note: This guide is written for an experienced developer who has created Web applications in the past. It does not include a detailed discussion of the Model-View-Controller (MVC) pattern or its associated implications, benefits, and disadvantages. There is certain basic knowledge that you must possess to understand and wield the power of a production framework. Agavi is not complicated, but it is certainly vast. If you are a newcomer to Web application development, you should start your quest by studying MVC and HTTP mechanics. HTTP and the MVC architecture have a profound influence on how developers structure software and design user interfaces, sometimes in subtle ways.

Note: Since this tutorial focuses on building a Web application, it will often refer to Agavi features specifically within a Web context. However, Agavi is designed to be able to run in multiple contexts; most of the Web-oriented behaviors described can be easily generalized.
Note: This guide is valid for Agavi 1.0.X. Please note that beta versions and release candidates may differ in some aspects.

Prerequisites

You'll need a webserver and PHP >= 5.2.0. The basics of setting up a webserver and installing PHP will not be covered in this guide, please refer to the respective documentations.

About Agavi

Agavi at a glance

Agavi is a general purpose PHP application framework built around the Model-View-Controller architecture originally based on the Mojavi 3 Web application framework written by Sean Kerr. It provides a rich toolset that solves most of the routine problems in Web application development.

Agavi is designed for serious development. It is not a complete website construction kit but rather a skeleton over which you build your application. The architecture of Agavi allows developers to retain very fine control over their code. Agavi itself is written to be extensible, and powerful. Whereas this documentation serves as an important resource, many developers are comfortable learning from Agavi's source code.

Agavi strives to leave most implementational choices to the developers. Agavi's components are inherently extensible, and the framework itself is designed around a XML-based configuration system that provides a very flexible environment. For you — as a developer — this architectural style means that you are required to perform less hacking (or, in most cases, none at all!) at the framework level and therefore have more time to get your application written.

The framework works for almost all kinds of applications but excels most in large codebases, long-term projects, extreme cases of integration and other special situations.

MVC in Agavi

MVC and high-level application structure

Agavi implements the Model-View-Controller (MVC) architectural design pattern to provide structure to applications. Agavi's interpretation of MVC is quite conservative, and it is very important that you understand both how the three key parts of the MVC concept come together and how Agavi implements them.

MVC architecture defines three distinct parts of an application. Models contain the application's logic. Controllers interpret requests from the user interface and direct Models. Views question Models, and prepare and send output to the user interface.

The Model-View-Controller architecture offers certain advantages. Without in-depth discussion it is sufficient to say that the MVC approach—combined with Agavi's naming conventions and framework facilities—makes it possible to write dependable, testable and extensible code.

MVC in Agavi

A Model represents an unit of logic in your application. In Agavi, a Model is a PHP class. Its methods solve some sort of application-specific problem: for example, sending promotional emails to customers, or maintaining the customer database. Your application may be implemented entirely inside a single Model or across dozens of Models that represent various areas of your software. Some Models act as wrappers for external libraries and even other applications or interfaces. A newly created Agavi Model is just an empty class.

A Controller component is called an Action in Agavi. Just like Models, Actions are PHP classes. Their methods are called by Agavi at the appropriate time.

Actions have one or more corresponding Views. When a Web request arrives, the routing mechanism selects the initial Action to be executed; the Action performs necessary changes in application state by calling the Models; it also selects ("appoints") one of its Views to be executed after it finishes. After that, the appointed View performs rendering of the application's output.

Since HTTP works under the request-response model, Actions correspond to HTTP request handling, and Views to response compilation.

Note: This relationship does not mean that Agavi applications are restricted to HTTP. Agavi can be used successfully for applications in which no Web interface exists at all. From the practical perspective, all this means is that Views are never executed without an Action.

Actions and Views should not implement any part of the application's domain logic. They serve merely as glue between the user interface, request overhead, and the Models. This means that your entire application sans the user interface should be implemented in Models.

Actions, Views and Models are grouped into Modules. A Module is simply a directory that contains Actions, Views, templates, module-specific Models, validators, local configuration and whatever else that is only needed by a specific Module. Agavi Modules can be found in your application's `app/modules/` directory.

Agavi also has global Models and templates. They reside in `app/models/` and `app/templates/`.

Overview of Agavi services

Agavi is a PHP application framework. It is a rich library of PHP classes and supplemental data. This library provides certain services to you, the application developer. This chapter lists the basic Agavi services that a developer would use or adjust most often.

Agavi execution context

Agavi applications execute inside an envelope that's called the "context" of the application. This envelope is exposed to your application as the global singleton `Context` object.

The Context object is available to all of your code. It is an important object, since you use it to access the global services of the framework. For example, if you wanted to retrieve a named database connection from Agavi `DatabaseManager`, you'd do it like this:

$db_conn = $this->context->getDatabaseManager()->getDatabase('mydb');

MVC execution

Developers implement the target applications using the MVC architecture, spending most of their time working with Models, Actions and Views. This tutorial takes a look at these components in great detail.

Application configuration

Agavi configuration system stores names and their corresponding values. It is a general purpose service that's used both by Agavi applications and the framework itself. The configuration system compiles XML configuration files into executable PHP code. Agavi defines some configuration files for its own purposes. Here's some example code:

// Retrieve a configuration value for item "meaning_of_life" and prefix "org.mycompany.myapp" // In settings.xml this would be: // // <settings prefix="org.mycompany.myapp"> // <setting name="meaning_of_life">24</setting> // </setting> // // The second parameter is the value that should be returned instead of null $val = AgaviConfig::get('org.mycompany.myapp.meaning_of_life', 42);

Your application can access the configuration through the global `AgaviConfig` interface. The configuration system supports namespaces, arrays, value interpolation and advanced XML processing features.

Database access

The framework itself does not need to access databases. It does, however, provide a general database access abstraction mechanism.

When enabled, Agavi `DatabaseManager` reads and enacts the application's database configuration. It creates instances of database adapters, through which your Models can access the database servers.

Adapters may interact with low level database APIs or high level third party database libraries. Agavi ships with a number of adapters for most popular APIs, and you can easily write your own adapter if you work with a custom database backend.

$dbm = $this->context->getDatabaseManager(); $db = $dbm->getDatabase('blog');

$conn = $db->getConnection();

Request dispatcher and execution flow

Agavi applications are exposed to the Web server through the front end dispatcher script (typically `pub/index.php`). The dispatcher reads the requests and starts Agavi execution. Agavi routes the request to the appropriate place in your application.

Several major things happen behind the scenes before the execution is turned over to your application. Various Agavi components are involved in the process of handling the request and composing the response.

Routing

Web applications parse request URLs and invoke certain application actions. Agavi applications do this by specifying a routing map, which describes the URL structure of your application.

The routing mechanism can both construct and analyze URLs. It is extensible and can cope with any conceivable URL structure.

$ro = $this->context->getRouting(); // Generate an absolute URL to a named route $url = $ro->gen('posts.edit', array('id' => $post->id), array('relative' => false));

Security and Validation

Validation is Agavi's way to make sure that the request data conforms to certain rules. If these rules are violated, the framework would refuse to execute your code. When used correctly, validation nullifies the threats of malformed data. Validation rules are specified using a special XML language, allowing you to codify the assumptions that your code makes.

On the application level, Agavi provides a security mechanism that understands individual user permissions, allowing to specify who is allowed to access what in your application. Access security is enforced on the framework level as well.

Caching

Agavi has a special caching facility which can spare the execution of unneeded code by reusing data that's already been processed. This gives a great performance boost to real world applications.

Date and translation services

Agavi's internationalization services are designed to solve the common problems of maintaining an application in many languages and locales. Agavi translation services are an interface through which your application can transform certain data into their local representation. Thus, Agavi provides strong infrastructure for development of software with fully functional and utilized internationalization features.

Agavi also ships with a date manipulation library.

Overview of application execution flow

Agavi handles user requests and organizes the execution flow of an application. This chapter gives a brief overview what happens when a web request lands in the Agavi dispatcher. Understanding Agavi architecture is key to write efficient and durable applications.

Request dispatching

The first stage of execution is called the dispatch stage. The dispatcher script - typically `index.php` for Web applications - boots the framework and commands the Agavi Controller to dispatch the Web request.

The Controller looks up the request using the routing map to find out the identity of the Action that should be executed. If no route matches, or the specified Action can not be found, a substitute system action is executed instead to indicate an error condition.

When the requested Action is discovered, the Controller creates a special envelope object, called the Execution Container. Actions and Views execute inside the Execution Container which isolates them from the outside world and provides them with shared space.

As soon as the initial Action is loaded into the Execution Container, the Controller creates the global filter chain which governs the execution of actions. The Execution Container is fed into the filter chain. One of the filters in the chain executes the Container, and the resulting response is returned to the outside world - most often as an HTTP response.

Tip: Filter chains are configurable pipelines which execute filters. A filter is a plugin-like class that accepts an Execution Container and does something with it. Filters are nested, which means they can execute before or after other (inner) filters.

Running the Execution Container

Note: An Action always selects a View that is destined to compose the response. A combination of such Action and View is called an "Action-View chain". It has nothing to do with filter chains.

The Execution Container encapsulates an Action-View chain. It sets up the environment, runs both the Action and one of its Views and then invokes the rendering mechanism to apply templates if needed. The output of is collected and returned.

Internally, Actions/Views are executed inside another filter chain, which is called the "actions" filter chain. Unlike the global FC which is executed once per request, the actions FC is executed once for every Action. Thus, filters of the "actions" FC would be applied to every Action ever called, and the ones in the "global" chain will be applied globally.

The Action itself is invoked through the Execution Filter, which is the last filter in the "actions" chain. Output caching is performed at this stage.

Note: Several other filters may be involved in the action chain: for example, the security filter would refuse to execute an Action that the user isn't authorized to execute.

If the Action requests input validation, the request data is checked by Agavi according to the rules laid out by developers. If the data does not conform to the specified constraints, execution of the Action is aborted and error handling procedure begins instead.

After the validation and caching checks are finished, the Action will be executed. This is where your code comes into play: the Action interacts with Models and tells Agavi which View should be executed.

The appointed View's respectable executeXXX() method is called. The View sets up the configuration for output rendering and prepares any data needed by the rendering mechanism, again querying Models if needed.

After the View has finished executing, the rendering mechanism executes the rendering procedure. This procedure may involve calling other Actions, which in turn get their own Execution Containers and action filter chains. Alternatively, a View may choose to skip the rendering step and compose the response itself (for example, encoding API response into JSON would not require any sort of external templates.)

The resulting output is collected and sent back to the client.

A Word About Actions

Actions interact with the models to implements your applications specific logic. They serve as glue code between the incoming request and the business model laid out in your models. All data retrieval, be it from a database, a flatfile etc. should be encapsulated in actions. An action also handles all validation requirements and all error handling.

RequestMethods and Action execution
RequestMethods are a concept in Agavi that abstracts from the HTTP verbs GET/POST/PUT etc to more general terms that are applicable in any environment. Agavi can be used in SOAP, XML-RPC, console and other contexts where HTTP may not be involved. The most common methodnames are Read and Write, but others like Create and Delete exist and you can define your own methodnames on the fly if required. The default methodnames are chosen to map nicely to the standard HTTP verbs and also fit nicely into all other environments. The default mapping for HTTP verbs is as follows:

- GET is translated to "Read". GET request should not modifiy any data on the server as per HTTP Spec.
- POST is translated to "Write".
- PUT is translated to "Create"
- DELETE is translated to "Delete"

An action may respond to a specific RequestMethod by simply implementing a method named after the RequestMethod: implement a method named `executeRead()` to respond to read requests, implement `executeWrite()` to respond to write requests. An action may respond to as many RequestMethods as you wish simply by implementing more than one method. An action may respond to all request methods by implementing a method `execute()`. An action may skip all exection by simply defining a method `isSimple()` and returning (bool) true. If an action does not respond to a RequestMethod it's `getDefaultView()` method is called.

Application filesystem layout

An Agavi application directory is structured according to Agavi's conventions. The Agavi project configuration system knows about the layout and its commands create or modify files and directories inside the source tree.

An application's `pub/` directory is the document root for your Web server: it contains static resources such as CSS files, javascript source, images and other items that are reachable through the Web server but do not involve invocation of the application.

The application itself—configuration, temporary and cache directories, libraries and source files—are located in `app/`.

`app/modules/`
Contains application Modules and their corresponding contents

`app/cache/`
Action cache and compiled configuration

`app/config/`
Application configuration (see next chapter)

`app/lib/`
Global framework extensions, third party libraries, etc

`app/models/`
Global application Models

`app/log/`
Log files (if logging into files is used)

`app/templates/`
: Global templates

Overview of application configuration

Agavi application configuration is defined in XML. The configuration engine validates and parses the configuration, compiling it into cached PHP source code, which is then applied to initalization of many Agavi classes. The configuration processors allow for validation, including further configuration from external files, XSL transformations, and several other powerful features. Configuration values are made available to Agavi and the application through the `AgaviConfig` class.

The configuration system is the central management facility for Agavi and Agavi applications: it doesn't just provide configuration services to your application but also binds your application together, serving as the glue between various Agavi components. It specifies which implementations of core classes are instantiated so that you can "plug in" extensions without editing the framework's source code.

The configuration system is smart. When the application is executed in development mode, the configuration is recompiled every time the application runs. In production mode, however, the configuration is only compiled once (and is subsequently cached), so the benefits of XML configuration can be kept without having to parse XML on every Web server request.

An important concept of the configuration system is the environment. You can define configuration for many environments, for instance, your personal development setup environment, an automated testing environment, your organization's production environment, or a profiling environment. Configuration data can be extracted from configuration files conditionally according to the environment in which the application is being executed. In this tutorial, we will begin with a single environment.

Agavi has a large number of configuration files, but don't be discouraged by their seeming complexity! There are a few interesting files that we will repeatedly use:

`settings.xml`
: This file contains the global application settings. Various aspects of the application such as enabled services, core Actions and custom configuration are contained here.

`databases.xml`
: This file describes your application's database connections, if any. Agavi has a database manager and a set of adapters to integrate with various libraries and PHP extensions (including Propel, Doctrine, and PDO). You will edit this file to define connection parameters for your blog's database.

`routing.xml`
: This file describes the routing map. The routing map connects URLs and Agavi actions, allowing Agavi to recognize and create URLs. The routing mechanism is very versatile and allows nearly any kind of URL structure; in fact, routes can also match on conditions other than URLs (such as request headers).

`output_types.xml`
: This file describes the application's output types. In our initial application, the only output type is HTML, as it is all that a simple blog application produces. An output type configures Agavi's rendering mechanisms, defining various aspects from template naming conventions to HTTP response headers. Later on, we will add another type to implement RSS feeds. An output type also contains the layout configuration, which we will edit when we integrate a pretty HTML template into our application.

Other configuration files serve more advanced purposes:

`action_filters.xml`
: This file configures filters that run for every execution of an action. Filters are an

advanced mechanism used to organize and modify the execution flow of your application. Your actions are executed inside a Filter; security is enforced by another Filter. Several filters are shipped and configured with a default Agavi application.

`global_filters.xml`
: This file configures filters that run for every execution of an application.

`autoload.xml`
: This file tells the class loading mechanism where to find various class files. When you add a third party library, you typically tell Agavi how to load it using this configuration file.

`config_handlers.xml`
: This file tells the configuration system how to translate XML configuration to PHP source code. The configuration system allows for the definition of custom configuration files, which can be extremely useful when creating large projects.

`compile.xml`
: This file tells Agavi how to compile code caches. Its use is beyond the scope of this tutorial.

`factories.xml`
: This file defines how actual classes map to given core interfaces in Agavi's class factory system. This allows you to extend any of the core Agavi services or facilities without having to edit the framework's source code. For example, if you wanted to replace the security user object implementation, you'd provide the name of your new class here and Agavi will use it instead of its default implementation.

`logging.xml`
: This file configures the logging facility. You can use it in your application to keep logs of the application's activity (although this isn't the only use for the logging subsystem).

`translation.xml`
: This file configures internationalization and format conversion facilities. Agavi comes with fully featured internationalization and localization services, supporting several backends, including gettext.

Setting up the initial application

Overview

In this chapter you will create your application skeleton. By the end of this chapter, you will have a complete initial version of the blog application—a newly configured Agavi application skeleton.

The result of your hard labor this chapter is provided with this tutorial as the `stage1` tarball.

Installing Agavi

Prerequisites

Agavi's minimum requirement is PHP version 5.2.0 or newer. DOM, Reflection and SPL extensions are required, but these are always enabled by default unless you're using one of these weird Linux distributions made by "smart" people who think they know better. You'll find a full list of required and optional extensions below.

Note: It should be pointed out that Agavi is not a framework for beginners. To leverage the functionality and the advantages of it's structure and philosophy, a good knowledge of PHP is required, and it's highly recommended that you are familiar with concepts such as MVC, and (web) application development in general.

Also, you usually need a web server, such as Apache, unless you only want to write a console application. Agavi should work fine with all web servers that support PHP. Apache versions 1 and 2, LightTPD and Microsoft Internet Information Server have been tested and verified to be compatible with the framework and it's components, such as the routing. Other web servers are likely to work as well, maybe with some minor extensions to the framework.

Installation via PEAR requires the PEAR client, which comes with most PHP distributions. Installation from Subversion repository requires a Subversion client. grab it straight from the SVN repository, you'll need a Subversion client installed. The project manager system requires Phing, which is a PHP based clone of Java Ant.

- PHP > 5.2.0, we recommend to keep your system up to date and use a recent version. Please note that some features are not supported in PHP versions < 5.2.8, most importantly running on a system with magic_quotes_gpc enabled is not possible as there are major bugs in how uploaded files are processed.
- libxml
- dom
- SPL
- Reflection
- pcre
- xsl (optional) required only for transformation of pre 1.0 style configuration files.
- tokenizer (optional) used to generate more efficient config-caches.
- session (optional) no php session support means that you'll have to build your own session storage system and hook it into agavi.
- xmlrpc (optional) required for XML-RPC features
- soap (optional) required for SOAP features
- PDO (optional) required for database connectors that use PDO as base
- iconv (optional)
- gettext (optional) required to use gettext-translators in the interationalization feature.
- PHING >= 2.3.1 (optional) required to use the build system.

Installation From PEAR

Installation with PEAR is relatively simple. Discover the Agavi PEAR channel and then install Agavi using the PEAR command line interface:

```
$ pear channel-discover pear.agavi.org

Adding Channel "pear.agavi.org" succeeded
Discovery of channel "pear.agavi.org" succeeded

$ pear config-set auto_discover 1
$ pear install -a agavi/agavi
```

It is also possible to obtain a specific version of Agavi:

```
        $ pear install agavi/agavi-0.11.0
```

Tip: Depending on your system configuration, you might need to run these commands with superuser rights. This is, for instance, often the case with PEAR installs on Mac OS X.

Using a tarball

Installing Agavi by hand is the logical choice if you can't use PEAR, or if you seek to bundle Agavi with your application so that your users or customers do not have to install it separately. Download a release tarball from the Agavi website, unpack it somewhere and move the contents of the `src` folder to a location of your choice. Make sure to adjust PHP's include path to cover this location. Don't forget to copy `etc/agavi-dist` or `etc/agavi.bat-dist` script to an `agavi` or `agavi.bat` executable into a convenient location so you can use the shell commands for creating projects, modules, actions etc. Keep in mind that you must edit the script and enter the path to your Agavi installation (that's the `src` folder you copied earlier) so everything works as intended.

The more usual case, however, will be that you're shipping Agavi together with your application, either because you can't use PEAR/Phing to install Agavi in the production environment, or because you want total control over what version of Agavi is used. In this case, copy the `src` folder to somewhere inside your application's directory structure (we recommend a `libs` folder that holds all libraries your app uses and that sits on the same level as the `app` and `pub` directories) and give it a different name, `agavi` would stand to reason. For that little extra something, copy the agavi-dist or agavi.bat-dist script again, maybe to your application root. Then, your application's`pub/index.php` should be changed to use a relative path for including `agavi.php`, such as `../libs/agavi/agavi.php`. Agavi will auto-determine and remember the path where it was loaded from, and will work without any changes to your environment's include path.

From SVN

Public access to the Subversion repository is possible via `http://svn.agavi.org/`. Specific releases are available through `/agavi/tags`. Development usually occurs in `/branches`, so if you're looking for a latest useful version, look there. `/trunk` contains the bleeding edge version that the core developers are working on at the moment. It is more likely than not to be broken at any given point of time, so don't use it unless you know what you're doing.

You can either perform an `svn checkout`, which creates `.svn` folders that allow you to stay up to date on changes in the repository via `svn update`, or you can simply `svn export` the contents of the repository without Subversion control folders, which is a good idea for example when you want to export a specific release into your own versioning control system.

Once you have obtained a copy from the repository, you can either follow the manual installation procedure, or generate your own PEAR package to be installed by PEAR locally by doing:

```
        $ cd (agavi checkout path)
        $ pear channel-discover pear.agavi.org
        $ phing package-pear
        $ cd build/
        $ pear package
```

This will create a `.tgz` file which can be installed using PEAR. If you already have Agavi installed it might be a good idea to remove it first by running `pear uninstall agavi/agavi`.
> Important: We recommend that you use only released versions for production use as they have gone through a full testing cylcle. We generally try not to break trunk or branches but sometimes it just happens. It's fine to install a development version if you just want to poke around and explore new features.

Testing your installation

A properly deployed Agavi installation enables the Agavi project configuration script, agavi. You should test that the installation worked correctly by executing `agavi` from the command line. This should provide some basic information about your current environment:

```
$ agavi
Agavi > status:

[echo] PHP:
[echo]   Version: 5.2.6-2
[echo]   Include path: .:/usr/share/php:/usr/share/pear
[echo]
[echo] Phing:
[echo]   Version: Phing version 2.3.1RC1
[echo]
[echo] Agavi:
[echo]   Installation directory: /usr/share/php/agavi
[echo]   Version: 1.0.0
[echo]   URL: http://www.agavi.org
[echo]
[echo] Project:
[echo]   (not found)
[echo]
[echo] For a list of possible build targets, call this script with the -l
argument.
```

If you installed agavi by hand you might need to adjust the path to the agavi script. Note that version information may differ slightly. If you are getting errors instead, Agavi isn't installed correctly. If you can't figure out the problem yourself, don't hesitate to ask the core developers.

Creating an Agavi project

Once Agavi is installed on your system, you can begin using the project configuration system to manage your projects. This system can be accessed using the `agavi` shell script.

Creating an empty agavi application

This means that you are ready to create the structure for your new blog application, affectionately called "Bloggie" (feel free to give it a less cuddly name if you can't stand it; the developers recommend "Incredible Scorpion-Powered Blog Application of Doom").

To create a project, first create a directory in your webservers document root directory where the application will reside. For the tutorial we'll assume that you're using a server at localhost and the application directory is named "bloggie". So, if your webservers document root is located at `/srv/www`, create the directory `/srv/www/bloggie`, if you're using windows and the document root is at `C:\xampp\` create `C:\xampp\bloggie`. Then change to that newly created directory and invoke `agavi project-wizard`. Some of the prompts may seem confusing at first—don't worry, you will have a chance to explore their meaning in more depth after your project is created. For the time being only provide a name for the project, a prefix to be used for all project specific classes and just leave the defaults for the rest, so press enter when those prompts appear.

```
www$ mkdir bloggie
www$ cd bloggie
bloggie$ agavi project-wizard

Project base directory [/srv/applications/bloggie]:

Project name [New Agavi Project]: Bloggie
Project prefix (used, for example, in the project base action)
[Bloggiedemo]: Blog
Default template extension [php]:

Name of the environment to bootstrap in dispatcher scripts [development]:

Should an Apache .htaccess file with rewrite rules be generated (y/n)
[n]?

Space-separated list of modules to create for Bloggie [Default]:

Space-separated list of actions to create for Default:

Module for default system action [Default]:

Action for default system action [Index]:

Space-separated list of views to create for Index [Success]:

Module for error_404 system action [Default]:

Module for unavailable system action [Default]:

Module for module_disabled system action [Default]:

Module for secure system action [Default]:

Module for login system action [Default]:

bloggie$
```

Congratulations! You have created a brand new Agavi application. Point your favorite browser to http://localhost/bloggie/pub/ and you should be greeted by the agavi welcome page.

Finishing the setup

Once the project is created some setup tasks are still left to do.

Setting up your editor or IDE

While this may seem trivial it is still important to configure your editor properly. Eclipse for example tends to use a local encoding for files instead of utf8. We strongly recommend using utf8 for various reasons, so now would be a good time to change that and have your favorite editor use utf8 as charset.

Removing the welcome page

The welcome page only serves to check wether creating the project was completed successfully, so we can remove it now. To do so, remove the Welcome module by deleting the directory `bloggie/app/modules/Welcome`. You probably won't need the image resources for the welcome page any more, so remove them by deleting `bloggie/pub/welcome`. Remove the route pointing to the welcome page from `bloggie/app/config/routing.xml`, it's the first one in the file, the comment line should point you right at it.

After you've done that you should see a blank with an "Index" headline.

Adding version control

You should be using version control for any serious project. Now would be a good time to add the project to version control - all the basic setup works are done and we checked that the application is basically working. If you choose not to use version control, you can skip this step but we still recommend you read what's here.

We won't cover the basics of version control in this tutorial or pick a SCM for you, however there's a few things to note that apply to all SCM systems. There are some files in your project that contain information that depends on the environment you're running the application in. The index.php dispatcher file contains the name of the environment in the bootstrap call, and if you chose to generate the .htaccess file, it contains the path relative to the web root. If you check those files in you always have locally modified files in your working copy and you need to be extra careful not to check those in as it would break other developers working copies or even your production environment. The best practice for all version control systems is not to add those files to version control at all but instead add a template file that is copied and adapted as needed. The agavi project generator already added the template files for .htaccess and index.php to `bloggie/dev/pub/` so all you need to do is take care that you don't check in the files `bloggie/pub/index.php` and `bloggie/pub/.htaccess`. Instead add them to your SCMs ignore mechanism, for svn set the svn:ignore property accordingly, for darcs add those files to _darcs/prefs/boring etc.

Oh, and while we're at it: add all files in `bloggie/app/cache/config` to the ignore list as well.

Tip: There is another way of handling the dispatcher file. By reading the environment from a server variable such as $_SERVER[AGAVI_ENVIRONMENT], the server admin can set the environment for you and you can check in the dispatcher file. However, this does not apply to the generated .htaccess file. This way of handling the environment dependency does have it's ups and downs. While it removes the step of manually copying the dispatcher file it forces anyone to use the same dispatcher file. You cannot have multiple environments on the same server without your server admin setting a variable for a specific path and more important, you cannot use any of the config directives that need to be set manually in the dispatcher file. We'll cover those later though, so for the time being, stick to the recommended mechanism of copying the dispatcher file.

Finishing the basic setup

Now would be a good time to perform some setup tasks that might seem a bit unnecessary at the moment but will save you a lot of time and hassle further down the road.

Getting away from pear

While using the pear install is just perfect for the basic project setup and first steps, it is recommended to move away from using the pear install and bundle the agavi lib for further development. This enables your fellow developers to check out your application without having a full agavi install and gives you full control over which agavi version you need for your application. This will come in handy the moment someone upgrades the installed agavi version on the server or you need to install two applications on the same server that require different agavi versions.

The first step is to bundle agavi with your application by placing a copy in `libs/agavi`. You can either have a full export there, use a tarball or just have an external point to the appropriate revision in the agavi svn repository.

Then create a copy of the agavi bin directory in dev/bin. Again you can use an export or use an external, but best use the same method as you used to bundle agavi to ensure consistency. Whatever method you choose, make sure you only check in a template of the agavi build script as the path setup must be done manually in some cases.

For the tutorial we'll be using the 1.0.0 release tarball. Just download it from www.agavi.org/download and extract it. Copy the contents of the src directory to `bloggie/libs/agavi`. Copy the contents of the bin directory to `bloggie/dev/bin`. Copy all the licence files to `bloggie/`. Add all those files to version control. Now copy the agavi build script in `dev/bin` to it's real name. On windows copy agavi.bat-dist to agavi.bat, on linux and other unix derivates copy agavi-dist to agavi. Adapt the path to the agavi directory in the script and make it executable if nessesary. Add the script to your VCS ignore list. We'll be using the bundled version from now on.

Tip: While you can bundle agavi as a svn:external as lot's of people do, we recommend that you use a full export at least for deployment. While we do our very best to ensure constant availability, it would be a pity to have your application deployment interrupted because our sysadmins choose to reboot the subversion server, taking it offline just in the right moment to have you in big trouble.
Remember: When bundling agavi with your application you have to take care to obey it's license. The license is LGPL, so using and bundling is just perfectly fine, even in closed source applications - you just need to bundle the licence file somewhere.

Installing a new copy of your application

Sooner or later you'll face the task of installing a new copy of your application - either you get a new developer on your team or you need to deploy your app to a testing/staging/live environment. With all the work we just invested that's a simple task. Here's what you do:

1. Check out or export your application from the VCS.
2. Copy and adapt the build script in dev/bin/
3. Call dev/bin/agavi public-create and answer the questions (we assume a development-joe-rambo environment here):

```
bloggie$ dev/bin/agavi public-create

Name of the environment to bootstrap in dispatcher scripts [development]:
development-joe-rambo
Should an Apache .htaccess file with rewrite rules be generated (y/n)
[n]?
```

This will copy all files from `dev/pub` to the appropriate place in `app/pub` and adjust the environment to bootstrap. Add configs for your new environment as needed, done.

Adding first code

Now would be a good time to add a little code of your own to the project. We'll start by adding a static index page and a static post detail page. You will learn to create new Actions and Modules and how to use routing and template variables. You'll also learn how to retrive and validate request parameters. The resulting application is also available as stage2 tarball.

Creating a new module

The index page should contain a list of posts and link to the detail page - don't worry we'll add that one later. For each post, the title and the author is displayed, as well as the date it was posted and the category it was posted in.

Creating a new Module

A module is an organizational unit structuring your applications separate areas of concern. A module structures Actions, Views, Models and Templates into a common subdirectory. It can even bring it's own libraries and some configs specific to this module. Handling the Post and displaying them to the user is one area of concern for our blog, so code related to that should be placed in one Module.

Creating a new Module is a simple task if we use the agavi commandline script. So navigate to the directory where you created your application, invoke `agavi module-wizard` and answer the questions.

```
bloggie$ dev/bin/agavi module-wizard
Module name: Posts
Space-separated list of actions to create for Posts: Index
Space-separated list of views to create for Index [Success]:
bloggie$
```

We created a new module named "Posts" which currently contains a single action named "Index". However, this action is not accessible as there is no route that points to it. We want the list of posts to be on the front page so we just point the default index route to that action. Look for the route that looks like that: <!-- default action for "/" --> <route name="index" pattern="^/$" module="%actions.default_module%" action="%actions.default_action%" /> and change it to point to the new action: <!-- default action for "/" --> <route name="index" pattern="^/$" module="Posts" action="%actions.default_action%" /> Done. The action "Index" in the module "Posts" will now be the index page. We could now remove the old IndexAction, but just keep that one for later, we will need it.

Adding a template variable

We want to display a list of posts on the index page. For now, we're just concerned about getting the basic html to work, so we just forget about where to get the data from for the time being. We'll just pass a static array of posts to the template. A post will need the following information to be displayed:

id
    The posts id.
title
    The posts title.
author_name

The authors name.

posted
: The date the post was created.

category_name
: The name of the category the post was filed under.

url
: The url to the posts detail page. We will leave that one empty but include it so that the attributes are complete.

To pass the information to the template we just need to set it as an attribute in the view or the action. Any attribute set in the global namespace will be available in the template later on. We want to serve the index page on Read (GET) requests so we define the `Posts_IndexAction::executeRead()` as follows - remember that all data retrieval should be done in the action: public function executeRead(AgaviRequestDataHolder $rd) { $posts = array(1 => array('id' => 1, 'title' => 'First post', 'posted' => '2008-07-14 00:01:07', 'category_name' => 'No category', 'author_name' => 'Admin', 'url' => null), 2 => array('id' => 2, 'title' => 'Second post', 'posted' => '2008-07-14 00:01:07', 'category_name' => 'Agavi', 'author_name' => 'Admin', 'url' => null)); $this->setAttribute('posts', $posts); return 'Success'; } And while we're at it - why don't we change the title a little. Have you noted the line in the Posts_IndexSuccessView saying $this->setAttribute('_title', 'Index'); As you might already have suspected, it sets a template variable that we use to control the displayed title. Don't worry if that seems a little like magic at the moment, we'll show you how that works in a little while. For now just change the line to read: $this->setAttribute('_title', 'Latest Posts'); Refresh the page and you should see your new title on the page and in the titlebar.

Now that we passed in the information about the posts, we adapt the template to make use of this information. As mentioned above, all the attributes we have set will be available in the template variable $t in the template. We want to display the posts as a list, so we change the template to look like in the following listing: <ul> <?php foreach ($t['posts'] as $post) { ?> <li> <a href="<?php echo $post['url']; ?>"><?php echo htmlspecialchars($post['title']); ?></a> by <?php echo htmlspecialchars($post['author_name']); ?> @ <?php echo $post['posted']; ?> in <?php echo htmlspecialchars($post['category_name']); ?> </li> <?php } ?> </ul> See how we were able to access the attribute 'posts' that we set earlier on in the action by just using $t['posts']? The PHP Renderer we're using just makes all the template variables available as an associative array with their names as key.

There's a few things to note about that template. See how all output is wrapped in a htmlspecialchars() call? We want valid xhtml, so we can't use any of the characters that are reserved in html. We don't want to escape them before storing the post in a database or the like, because maybe we want to generate output that is not html later on and then we'd have to decode them again. Neither do we want to have the htmlspecialchars call in the view as some templating engines do that implicitly. So converting the specialchars in the view would make it harder to just swap in a new templating engine. And we don't want to use htmlentities() as we're trying to be XML compliant at the same time. If you have any special characters like umlauts, just use utf8 as charset and the regular character. It will save you a lot of hassle later on. If htmlspecialchars() is too long to type for you, we recommend having a function with a shorter name like `h()` or `hss()` that just wraps the call. For the tutorial, we'll stick with the long version.

Creating a new action

Creating a new action

To create a new action invoke `agavi action-wizard` and answer the questions.

```
bloggie$ dev/bin/agavi action-wizard
Module name: Posts
Action name: Post.Show
Space-separated list of views to create for Post.Show [Success]:
bloggie$
```

We created a new action named "Post.Show" in the Posts module. The dot in the name denotes a subaction. Subactions can be nested to arbitrary depth and are useful to organize your actions in a tree-like structure. Now we just need to make that action accessible with an url. Once again we use a route for that. Hang on a litte, we'll explain routing in a moment. For now, just insert this route in the routing.xml, right after the index route: <route name="post" pattern="^/post$" module="Posts" action="Post.Show" /> Done. The action "Post.Show" in the module "Posts" will now be accessible by the url /post. Go ahead, open your browser and try it. You should see a page with a big headline saying "Post.Show".

Adding content

You already learned about template variables in the last parts, so we make this a quick one - we just add the information for a single post to the template, containing the same information as in the list plus the body. We allow for the body to contain html, so we can format the text a little. Data retrieval goes in the action, so add the Posts_Post_ShowAction::executeRead() to look like this.

public function executeRead(AgaviRequestDataHolder $rd) { $post = array('id' => 1, 'title' => 'First post', 'posted' => '2008-07-14 00:01:07', 'category_name' => 'No category', 'author_name' => 'Admin', 'url' => null, 'content' => '<p>Terrific! This is our first post!</p><p>This is just a first post. It has no actual contents. If you are reading this, things must be working.</p>'); $this->setAttribute('post', $post); return 'Success'; } And we should adapt the title here as well. So open the the Posts_PostSuccessView and change the line saying $this->setAttribute('_title', 'Index'); to $post = $this->getAttribute('post'); $this->setAttribute('_title', $post['title']);

An then the template: <?php // alias the post, to make access shorter $post = $t['post']; ?> <span class="author">by <?php echo htmlspecialchars($post['author_name']); ?></span> <span class="category">in <?php echo htmlspecialchars($post['category_name']); ?></span> <span class="posted"><?php htmlspecialchars($post['posted']); ?></span> <div class="content"><?php echo $post['content']; ?></div> Open your browser and point it to http://localhost/bloggie/pub/post and you should see the stub page we just created.

Tying things together - an introduction to routing

So far we have been concerned with how to display some information to the user, but not with how the user finds the content we want to show. But as there are many ways to access data, there are several ways to describe where to find what - Websites and REST use URLs, SOAP uses the SOAPAction and so on. And then, the application itself organizes itself into modules and actions. Routing ties all of this together. It maps URLs to modules and actions for websites, extracting parameters as needed, handles XML-RPC and SOAP and even provides a way to use agavi on the commandline.

For each of these tasks exists a specialized subclass of AgaviRouting, this chapter will mainly explain the basics and the web-specific AgaviWebRouting that maps urls to actions.

Routing basics

Routing is executed very early in the request processing and can access all data known to the process at that time. The Webrouting can access headers, cookies and the url to determine which action should be executed.

The routing uses rules based on regular expressions to match the input and extract parameters as it finds them. For more elaborate examination of input data routing callbacks can be used.

If no appropriated action is found, the configured Error404 action is set as default. The result of the routing execution is a Execution Container on page which is then executed by Agavi.

Routing configuration

The routing configuration is done in the `app/config/routing.xml`. Let's have a look at our current routing.xml. It should look like this: <?xml version="1.0" encoding="UTF-8"?> <ae:configurations xmlns:ae="http://agavi.org/agavi/config/global/envelope/1.0" xmlns="http://agavi.org/agavi/config/parts/routing/1.0"> <ae:configuration> <routes> <!-- default action for "/" --> <route name="index" pattern="^/$" module="Posts" action="%actions.default_action%" /> <route name="post" pattern="^/post$" module="Posts" action="Post.Show" /> <!-- an example for a CRUD-style set of routes --> <route name="products" pattern="^/products" module="Products"> <!-- do not put the action into the parent route, because that one is not anchored at the end of the pattern! --> <route name=".index" pattern="^$" action="Index" /> <route name=".latest" pattern="^/latest$" action="Latest" /> <route name=".create" pattern="^/add$" action="Add" /> <!-- "Product" is not an action, but just a folder with sub-actions. if only this route, without children, matches, then the action cannot be resolved and a 404 is shown - exactly what we want! --> <route name=".product" pattern="^/(id:\d+)" action="Product"> <route name=".view" pattern="^$" action=".View" /> <route name=".edit" pattern="^/edit$" action=".Edit" /> <route name=".delete" pattern="^/delete$" action=".Delete" /> <!-- the gallery page is optional here, but the request parameter should not contain the leading slash, so our special syntax is in order --> <route name=".gallery" pattern="^/gallery(/{page:\d+})?$" action=".Gallery"> <!-- assume the "1" by default and tell the routing what the rest of the string will look like when generating a URL --> <default for="page">/{1}</default> </route> </route> </route> </routes> </ae:configuration> </ae:configurations> The first two routes are the index route that we adapted earlier on and the one we created in the last step. There's a couple of things you can notice on them:

- Both routes have a name attribute. The name is used to reference the route in your code when you need to generate a url.
- Both routes have a pattern attribute. The pattern is applied against the input to check whether the route matches. An empty pattern always matches - to match the empty string use "^$".
- Both routes have a module attribute. It controls which module is selected if this route matches.
- Both routes have an action attribute. It controls which action is run if this route matches.
- An attribute that is not used here but still important enough to mention is "stop". It takes a boolean value and defaults to true. If set to false, matching does not stop after the route matches, instead the matched part is stripped from the input and matching continues.

These are a few, but probably the most important attributes for a route. We'll discuss the other attributes for a route later on.

Nesting Routes

Routes can be nested as you can see in the generated sample part of the routing.xml.

Child-routes inherit the information set by the parent route and can overwrite it or append to it, depending on the type of information. It is not possible to append to any attribute other than name and action. Any value that starts with a dot (.) is appended to the parent value, so if the parent is named "products" and the child ".index" as in our routing file, the full name for the child route would be "products.index". Any parent route cannot have the "stop" attribute set to false.

The input for a child-route is the parent's input stripped by the part matched by the parent.

Routing Patterns

Routing uses special type of pattern matching language which is similar to PCRE regular expressions but simpler. Routing patterns are optimized for URL matching so that you do not have to escape common URL parts every time.

Just like with PCRE, a pattern can begin with a circumflex (^) and/or end with a dollar sign ($). By default, the pattern matches *any* part of the examined value. Use of these anchors forces the pattern to match only in the beginning and the end of the examined value respectively. So, **"foo"** matches any string that contains the word "foo", **"^foo"** matches only strings that *begin* with "foo" and **"foo$"** will match strings that *end* with "foo". Using both anchors means that the entire examined value must match the pattern.

The body of the pattern is a plain string which matches the examined value literally. To match parts with a regular expression, you have to enclose the relevant parts of the pattern in parens (called capture groups) which behave almost the same way as in true PCRE. Capture groups are also used to identify the parts of URL which become Action parameters, as you can see in the products.product route. It defines an Action parameter named "id" that consist of one or more digits. Capture groups may contain an optional prefix and/or postfix, in that case the contained regular expression must be enclosed in curly braces as seen in the products.product.gallery route.

We'll get to a more detailed description of routing patterns later.

How Routes are matched

The definitions in the routing.xml are processed top to bottom, the first matching route definition ends the processing unless it has its "stop" attribute set to false. If a parent route is matched, matching continues until all child routes have been tried and stops after that. When no matching definition is found, the configured Error404 page is displayed.

Organizing routes for best performance

There are a few things to keep in mind when organizing routes.

- Non-stopping routes go to the place where they are required, mostly to the top.
- Avoid routes that match partial strings of other routes at the same level. This forces an order on routes that may be confusing - once you start shuffling things around, other things break. It also enforces an order that may be suboptimal for performance.
- Anchor patterns. Keep in mind that patterns that are anchored are way faster to match and that errors are less likely to occur.
- Reduce the number of first level routes. The less patterns need to be matched in a typical szenario, the better, so a deeper nesting is better than lots of first level branches.
- Move often calles routes to the top. The earlier a match is found, the better. Keep in mind that the index route ("/") can account for large percentages of your hits and is extremly fast to match. In most cases, it should be the first route in your routing.xml.
- Move routes that are quick to match to the top. This includes all short and simple patterns, pure string matches with no included parameters etc.

- Take care that your routes are anchored to the right as well. This prevents arbitrary urls from being matched.

Let's go over the rest of the routing.xml and explain what each part does. It's a sample for a CRUD-style set of routes for a fictional product catalogue. The parent route is named "products" and just contains the pattern "^/products" and a module defintion. If the rules are applied against any url that starts with "/products", the module is set to "Products" and the leading part is stripped from the input. If none of the child routes matches, the 404 page is displayed, because the action is not set. The products.index route has a pattern that matches the empty string and sets the action to "Index", so that the url "/products" is routed to the Products_IndexAction. We could achieve the same effect if we just set the action to Index in the products route, but then any url starting with "/products" that is not matched by a child-route would still go to the Products_IndexAction and this is not what we intended - we want the url "/productsfoobar" to go to the 404 page.

The next important thing to note is the products.product route. It sets the action to "Product" but as the comment explains, this action itself should not exist. It's just a folder for all the subactions for a product. The pattern matches and url that contains a slash followed by an id of only digits. So the url "/products/123" would first get matched by the route "products" where the string "/products" is cut from the url. This leaves us with "/123" which will be matched by the products.product route. This would still send us to the 404 page as the Products_ProductAction does not exist, but the remaining empty string will get matched by the products.product.view route. So the final action is Products_ProductViewAction.

The last interesting route is the products.product.gallery route. It contains an optional paging parameter and by using the defaults tag sets a default value in case it's not provided by the user. Note how the optional page parameter contains a "/" as prefix and how the prefix must be placed in the default as well.

As you can see, all routes that actually map to an existing Module/Action pair are anchored on the right. That prevents that urls are matched that start out with a valid part but contain trailing garbage.

Fixing the bloggie routing

Armed with what we learned we can now create a proper set of routing rules that will remain maintainable when we go on extending the blog we're building. We should create the basic layout for a CRUD-style set of routes as the sample product routes.

So, let's reorganize our routes:

1. Create a parent route for all posts that just sets the module. Name it "posts", the pattern should be "^/posts" (remember the anchor on the left) and it should set the module to "Posts". It should be the first route after the index route.
2. As we're currently not having a page that displays all posts page by page, we'll go and create the a route named "posts.post" as a child of the posts route. Remember that the concatenation rules for the route name allow you to leave the first part out, so the name attribute will be set to ".post". The pattern should match a post-id, so we'll use "^/(post:\d+)". Don't forget the left anchor. This route sets the action to "Post", even though that action does not exist in itself. It will serve as a container for all it's subactions.
3. The next route will server for displaying a single post. Create "posts.post.show" as a child of "posts.post". The name attribute is ".show", the pattern "^$" - we want to match the empty string only. Ste the action attribute to ".Show" - it will be appended to the parents route "Posts" to the fully qualified name "Post.Show".
4. Remove the old "post" route and the example routes. Your routing.xml should now look like this:

```
<?xml version="1.0" encoding="UTF-8"?> <ae:configurations
xmlns:ae="http://agavi.org/agavi/config/global/envelope/1.0"
xmlns="http://agavi.org/agavi/config/parts/routing/1.0"> <ae:configuration> <routes> <!--
default action for "/" --> <route name="index" pattern="^/$" module="Posts"
action="%actions.default_action%" /> <route name="posts" pattern="^/posts"
module="Posts"> <route name=".post" pattern="^/(post:\d+)" action="Post"> <route
name=".show" pattern="^$" action=".Show" /> </route> </route> </routes>
</ae:configuration> </ae:configurations>
```

Open your browser and point it to http://localhost/bloggie/pub/posts/123 and you should see the stub post page we created earlier on. Note that now you need to provide a post id to access the page, if you leave it out, you'll be directed to the 404 page.

Generating URLs

So far we did not link the posts on the index page to the proper detail page. Generating an url from a route is a simple task, just call `AgaviRouting::gen()` with the routename, add the parameters you want to set and that's it. Let's fix our IndexSuccessView to generate the proper urls. We'll read the posts attribute, modify the entries and write it back.

```
<?php class Posts_IndexSuccessView extends BlogPostsBaseView { public function
executeHtml(AgaviRequestDataHolder $rd) { $this->setupHtml($rd); $ro =
$this->getContext()->getRouting(); $posts = array(); foreach($this->getAttribute('posts')
as $p) { $p['url'] = $ro->gen('posts.post.show', array('post' => $p['id'])); $posts[] = $p; }
$this->setAttribute('posts', $posts); $this->setAttribute('_title', 'Latest Posts'); } } ?>
```

Now all posts on the front page link to the appropriate detail page.

Accessing Request Parameters and Validation Basics

Now that we intruduced a parameter to pass the posts ID we could display the proper post on the detail page. All incoming parameters, including cookies, headers and files are passed to the view's `executeHtml()` method via a container object, an `AgaviRequestDataHolder()`. In the web context, a more specific subclass, an `AgaviWebRequestDataHolder()` is being passed. Any parameter extracted from the url, any GET or POST value can be accessed via `AgaviRequestDataHolder::getParameter(<parametername>, <defaultvalue>)`.

Validation Basics

However, in the default setup, strict validation is turned on. That means that only validated parameters may be accessed, all other parameters are removed by the framework. While it is possible to change that behavior, we strongly recommend keeping it that way. Validation is an important part in keeping your application safe from attacks of any kind or from failure due to unexpected input. Validation makes you as a developer think about what input is valid. Always validate as strict as possible. So let's see, what's our expected input?

- The post-id must be an integer.
- The post-id must be in the valid range, we have two posts with the ids 1 and 2.

So let's put that into code.

How Validation works

Validation is executed when an Action responds to the incoming request type. Validation rules are per action and may differ between read and write requests. While there are other ways of defining validation rules, in most cases they are laid out in an xml-configuration file, so that's all we'll do at this point. We'll deal with the other methods later. In the default setup the xml file is named like the action and placed in the folder `validate/` in the corresponding module. The build system created a stub file for us, it's located in `app/modules/Posts/validate/Posts/Show.xml` and for now only contains a stub. Let's extend it to look like that:

```
<?xml version="1.0" encoding="UTF-8"?> <ae:configurations
xmlns="http://agavi.org/agavi/config/parts/validators/1.0"
xmlns:ae="http://agavi.org/agavi/config/global/envelope/1.0"
parent="%core.module_dir%/Posts/config/validators.xml" > <ae:configuration>
<validators> <validator class="number" name="post"> <arguments>
<argument>post</argument> </arguments> <errors> <error>The parameter post must
contain a number in the range of 1 - 2</error> </errors> <ae:parameters> <ae:parameter
name="type">int</ae:parameter> <ae:parameter name="min">1</ae:parameter>
<ae:parameter name="max">2</ae:parameter> </ae:parameters> </validator>
</validators> </ae:configuration> </ae:configurations>
```

This defines a single validator with the name "post" and the class "number". The name is optional and can be used to get detailed information about where validation failed. In most cases you won't need it. The class-attribute controls which validator is run. It can either be a predefined shortname or real classname in case you want to use a custom validator. The <arguments> block defines on which parameters the validator is run. Some validators may work on multiple parameters, for example a validator checking that two parameters are equal. In our case we only need a single parameter named "post". The <errors> block controls which error message is emitted if the validator fails. We could have different messages for different errors, one if the value was not provided at all, one if it was no valid integer, another one if it was out of range. In this case a generic message is just fine. The parameters block allows more fine-grained control over the validator behavior. The parameters a validator accepts differ from validator to validator and may sometimes seem a little complex, but the `AgaviNumberValidator` we're using here is pretty straightforward. It allows a type, a minimum value and a maximum value.

With this validation file we've checked that the incoming data is all fine and we can rely in the rest of the application on that.

Displaying the right post

Now that we validated the incoming post-id we can display the requested post on our detail page. For the time being we just copy the static posts array that we used for the list and pick the right one of the two. Change the `Posts_PostShowAction::executeRead()` to look like this:

```
public function executeRead(AgaviRequestDataHolder $rd) { $posts = array( 1 => array(
'id' => 1, 'title' => 'First post', 'posted' => '2008-07-14 00:01:07', 'category_name' => 'No
category', 'author_name' => 'Admin', 'url' => null, 'content' => '<p>Terrific! This is our first
post!</p><p>This is just a first post. It has no actual contents. If you are reading this,
things must be working.</p>', ), 2 => array( 'id' => 2, 'title' => 'Second post', 'posted' =>
'2008-07-14 00:01:07', 'category_name' => 'Agavi', 'author_name' => 'Admin', 'url' => null,
'content' => '<p>It looks like our blog application is working, yay!</p>', ) ); $postId =
$rd->getParameter('post'); $post = $posts[$postId]; $this->setAttribute('post', $post);
return 'Success'; }
```

We don't have to check whether the array key exists because we validated the incoming parameter and the constraints are set strict enough that the valid cannot be out of range. And see how the separation of display and logic already worked in our favor? There was no need to adapt the view or the template even though we changed the logic reading the data.

Handling validation errors

So far we have only been concerned about how to keep users from passing invalid data but not about what happens if users passes invalid data - in our case calls an invalid URL to a post's detail page. If you try that now you'll see a shiny `AgaviException` explaining that it can't find a certain view's file. We'd rather see a "404 Not Found" page in that moment.

The handle*Error Methods

Whenever validation fails the framework calls a method on the action to handle the error. As with the execute* methods the framework first looks for a method named `handle<RequestMethod>Error()` and if no such method is declared it calls a generic `handleError()`. So if an error occurs on a Write request, agavi looks if `handleWriteError()` was declared, if validation fails on a Read request, `it looks for handleReadError()`. In both cases it would fall back to `handleError()` if no specific method was declared. If an action does not declare any error handling at all, the default `handleError()` method declared in `AgaviAction` is executed. The signature for all these methods is the same as for the `execute()` method, they accept an `AgaviRequestDataHolder` as single parameter and return a ViewName. Please refer to the API-documentation for `AgaviAction::handleError()` for further details.

In our case we'll rely on the standard method predefined in Agavi. It executes no logic and just returns 'Error' as the view's name. We don't have that view yet, so let's go create one.

Creating a new View

Creating a new view is most easily done using the agavi build system:

```
bloggie$ dev/bin/agavi view-create
Module name: Posts
Action name: Post.Show
View name: Error
bloggie$
```

Forwarding to another Action

A validation failure in this specific case means that the requested post does not exist or is otherwise inaccessible, so instead of displaying an error message to the user we should display the 404-Not-Found page. We don't want to implement this page in all and every place where such errors occur and we alread have an implementation for it so all we need to do is tell Agavi to continue with the default 404 action. This is called a "forward" as the client never gets to know that the last executed action is not the action that was actually requested. It's important to differentiate a forward and a redirect, the latter one uses the HTTP 301/302 header to tell the client to go looking somewhere else for the requested ressource.

Forwards should happen in the view as the response may depend on the output type. To forward in a view, we just return a new `AgaviExecutionContainer` with the module and action we want to forward to. To create such a container we use the convienience method `AgaviView::createForwardContainer()`. It will create a new container based on the current view's container and initialize it properly. Please refer to the API-Documentation for further details on this method.

<?php class Posts_Post_ShowErrorView extends BlogPostsBaseView { public function executeHtml(AgaviRequestDataHolder $rd) { return $this->createForwardContainer(AgaviConfig::get('actions.error404_module'), AgaviConfig::get('actions.error404_action')); } } ?>

The calls to `AgaviConfig::get()` just read the configured directives from the settings.xml and otherwise that's all we need to display the 404 page. Go ahead, have a look - enter an invalid post id and see how the right page is displayed. It even emits the right set of headers.

Putting the M in MVC

So far we created two static pages that feed on data that's held in the view. That's not what MVC is about. Instead we should put the logic of where to obtain the data in our business model and the view should only be concerned with how to display it. It's about time we change that. In this chapter, we'll move the code to retrieve and handle posts into separate classes and start to created our business model.

Models in agavi are just plain classes. While they usually extend AgaviModel in one way or another, this is not a requirement. You can create your business model completely independent of Agavi and then use it in your application. However, you'll loose a couple of benefits that the framework has to offer, so consider carefully whether you'll ever have the need to use your Models in a context without using agavi. Unless you have a very specific reason we recommend that your models extend the applications BaseModel which has been created by the project wizard.

We'll be using two models to handle our business logic. First a "Post" model that represents a single post and a "PostManager" that handles all storage and retrieval operations for the post. That way we can easily exchange the PostManager for a mock implementation in testing contexts. That way it's possible to have tests run even without a database dependency.

Creating a new Model

We'll use the agavi build script to generate the two models we need. As they belong to the posts handling part of our application, we'll create them in the in the Posts module:

```
bloggie$ dev/bin/agavi model-create
Module name: Posts
Model name: Post
bloggie$
```

This will create the Posts_PostModel in the directory `app/modules/Posts/models/`. The class will be empty for now, we'll add code to it in a second. First, let's repeat the same steps for the PostManager model.

```
bloggie$ dev/bin/agavi model-create
Module name: Posts
Model name: PostManager
bloggie$
```

Now you should have two files in `app/modules/Posts/models/`, one named PostModel.class.php, one named PostManagerModel.class.php containing the empty classes Posts_PostModel and Posts_PostManagerModel. Note how both classnames are prefixed with the module's name and that they both end in "Model". This convention is used troughout agavi. While it is possible to reference models that have different naming schemes we recommend sticking to this convention.

Both models extend the BlogPostsBaseModel class that was created when we created the module.

Instantiating Models

Now that we created the model classes, we need a way to create instances. We could go the long route with including the class file, creating a new instance with new Posts_PostModel and then calling the required initialize() method. However, there is an easier way. Any object with access to the `AgaviContext` (which is pretty much every object in the framework) can just call $ctx->getModel(<modelname>, <modulename>, <optional parameters>) to create a fully initialized instance of the given model. The context will then locate the proper class file, load it if required, and return an initialized instance of the requested model.

Fleshing out the Models

Let's add a bit of functionality to the models. The Post model currently is just a container for the data a Post can have, so we'll primarily add the attributes and getters and setters for those attributes and throw in two methods to create a post from an array of values and reverse. Your final Posts_PostModel should look like that (the comments are left out to keep it short):

```
<?php

class Posts_PostModel extends BlogPostsBaseModel
{
        private $id;
        private $title;
        private $posted;
        private $categoryName;
        private $authorName;
        private $content;

        public function __construct(array $data = null)
        {
                if(!empty($data))
                {
                        $this->fromArray($data);
                }
        }

        public function getId()
        {
                return $this->id;
        }

        public function setId($id)
        {
                $this->id = $id;
        }

        public function getTitle()
        {
                return $this->title;
        }

        public function setTitle($title)
        {
                $this->title = $title;
        }

        public function getPosted()
        {
                return $this->posted;
        }

        public function setPosted($posted)
        {
                $this->posted = $posted;
        }

        public function getCategoryName()
        {
                return $this->categoryName;
        }

        public function setCategoryName($name)
        {
                $this->categoryName = $name;
        }

        public function getAuthorName()
        {
                return $this->authorName;
        }

        public function setAuthorName($name)
        {
                $this->authorName = $name;
        }

        public function getContent()
        {
                return $this->content;
        }

        public function setContent($content)
        {
                $this->content = $content;
        }

        public function fromArray(array $data)
        {
                $this->setId($data['id']);
                $this->setTitle($data['title']);
                $this->setPosted($data['posted']);
                $this->setCategoryName($data['category_name']);
                $this->setAuthorName($data['author_name']);
                $this->setContent($data['content']);
        }

        public function toArray()
        {
                $data = array();
                $data['id'] = $this->getId();
                $data['title'] = $this->getTitle();
                $data['posted'] = $this->getPosted();
                $data['category_name'] = $this->getCategoryName();
                $data['author_name'] = $this->getAuthorName();
                $data['content'] = $this->getContent();

                return $data;
        }
}

?>
```

The Posts_PostManageModel is responsible for all storage and retrieval operations. We currently need two operations: Retrieve a single Post and retrieve the latest posts for the frontpage. As we now do have a model encapsulating all the data retrieval we can move our mock-data into that model. The endresult should look like this:

```
<?php

class Posts_PostManagerModel extends BlogPostsBaseModel
{
        private $posts = array(
                1 => array(
                        'id' => 1,
                        'title' => 'First post',
                        'posted' => '2008-07-14 00:01:07',
                        'category_name' => 'No category',
                        'author_name' => 'Admin',
                        'content' => '<p>Terrific! This is our first
post!</p><p>This is just a first post. It has no actual contents. If you
are reading this, things must be working.</p>',
                ),
                2 => array(
                        'id' => 2,
                        'title' => 'Second post',
                        'posted' => '2008-07-14 00:01:07',
                        'category_name' => 'Agavi',
                        'author_name' => 'Admin',
                        'content' => '<p>It looks like our blog
application is working, yay!</p>',
                )
        );

        public function retrieveById($id)
        {
                if (isset($this->posts[$id]))
                {
                        return $this->getContext()->getModel('Post',
'Posts', array($this->posts[$id]));
                }
                return null;
        }

        public function retrieveLatest($limit = 5)
        {
                $cnt = 0;
                reset($this->posts);

                $posts = array();

                foreach($this->posts as $post) {
                        $posts[] = $this->getContext()->getModel('Post',
'Posts', array($post));
                        $cnt++;
                        if($cnt >= $limit) {
                                break;
                        }
                }

                return $posts;
        }

}

?>
```

Adapting the Actions and Views

There are two changes we need to introduce. First we want our actions and views to use the models we created in the last step. Second we should move all data retrieval code to the place where it belongs - in the Action. Let's start with the Post.Show Action.

As our Post.IndexAction should respond to GET - Request, we'll need to adapt `executeRead()` that contains our data retrieval logic. Using the Posts_PostManagerModel we created in the last step this is very little work:

```
<?php

class Posts_IndexAction extends BlogPostsBaseAction
{
        /**
         * Serves Read (GET) requests
         *
         * @param      AgaviRequestDataHolder the incoming request data
         *
         * @return     mixed <ul>
         *                     <li>A string containing the view name
associated
         *                     with this action; or</li>
         *                     <li>An array with two indices: the parent
module
         *                     of the view to be executed and the view to
be
         *                     executed.</li>
         *                   </ul>
         */
        public function executeRead(AgaviRequestDataHolder $rd)
        {
                $manager = $this->getContext()->getModel('PostManager',
'Posts');
                $this->setAttribute('posts',
$manager->retrieveLatest(10));

                return 'Success';
        }

        /**
         * Returns the default view if the action does not serve the
request
         * method used.
         *
         * @return     mixed <ul>
         *                     <li>A string containing the view name
associated
         *                     with this action; or</li>
         *                     <li>An array with two indices: the parent
module
         *                     of the view to be executed and the view to
be
         *                      executed.</li>
         *                    </ul>
         */
        public function getDefaultViewName()
        {
                return 'Success';
        }
}

?>
```

Now we need to adapt the view accordingly:

```
<?php

class Posts_IndexSuccessView extends BlogPostsBaseView
{
        public function executeHtml(AgaviRequestDataHolder $rd)
        {
                $this->setupHtml($rd);

                $ro = $this->getContext()->getRouting();

                $posts = array();

                foreach($this->getAttribute('posts') as $p)
                {
                        $post = $p->toArray();
                        $post['url'] = $ro->gen('posts.post.show',
array('post' => $p->getId()));
                        $posts[] = $post;
                }

                $this->setAttribute('posts', $posts);

                $this->setAttribute('_title', 'Latest Posts');
        }
}

?>
```

Note how we transform the objects we retrieved from our PostManager to arrays. While it may seem easier to pass objects to the template it removes a lot of flexibilty. Some template engines don't handle objects gracefully. There is little added overhead and little code we had to add for that as we need to add some information anyways - in our case the url to the post detail page. We could have added the url in the PostModels toArray() method but that would be bad practice - the url is probably only required in this specific context and would be useless clutter in all other occasions.

Let's just go over our Post.Show action and adapt that, using our models it will be a breeze:

class Posts_Post_ShowAction extends BlogPostsBaseAction { public function executeRead(AgaviRequestDataHolder $rd) { $manager = $this->getContext()->getModel('PostManager', 'Posts'); $this->setAttribute('post', $manager->retrieveById($rd->getParameter('post'))); return 'Success'; } public function getDefaultViewName() { return 'Success'; } } class Posts_Post_ShowSuccessView extends BlogPostsBaseView { public function executeHtml(AgaviRequestDataHolder $rd) { $this->setupHtml($rd); $post = $this->getAttribute('post')->toArray(); $this->setAttribute('post', $post); $this->setAttribute('_title', $post['title']); } }

And that's it. Now our actions and views are completely decoupled from the data retrieval, they don't care about whether data is loaded from a flatfile, retrieved from a database, from a webserver or just faked with a static array like we do. You can bet that this will come in handy at some later point.

Custom validators

Now that our actions dont' depend on the data retrieval any more we need to tackle the validation. Currently we're doing a plain check that the post-id is a valid integer in the range of 1-2 but we're depending on the knowledge that the posts are stored in an array indexed by id and that there's just those two ids. If we're to add a third post or change the storage mechanism we'd be in trouble. So we could just relax our validation to check that the id is just a valid positive integer but then we'd need error handling code in the action - a user could just pass an id that does not exist. However, there's an easier way. We just write our own custom validator that does the job for us - don't worry it's a simple thing.

A validator is just a class that is based on AgaviValidator. It needs to implement a single method `validate()` that either returns true if the validation succeeded or false if it failed. That's about all we need to do, let's get to work. We'll be using the PostManagerModel we created earlier to do the checks. As this validator belongs to the Posts module, we'll place it in `app/modules/Posts/lib/validator/PostValidator.class.php`.

<?php class Posts_PostValidator extends AgaviValidator { /** * Validates the input * * @return bool The input is valid number according to given parameters. */ protected function validate() { $postId = $this->getData('post'); $manager = $this->getContext()->getModel('PostManager', 'Posts'); $post = $manager->retrieveById($postId); if (null == $post) { $this->throwError(); return false; } return true; } } ?>

That's all. There's three methods you don't know yet: The `getArgument()` method retrieves the configured argument name, the `getData()` method retrieves the request data for the given argument name and the `throwError()` method emits an error message. Please check the API-documentation for a more detailed description.

But hey, wait. Now we retrieve the post twice, once in the validator to check whether the id is actually valid and once in the action to actually use it. That does not seem reasonable - it duplicates code and even worse, if retrieving the post is a slow operation, it will cost us performance. We wouldn't want that. Good for us, there is a way around that. A validator may export a validated value to any parameter name we choose. This can be the incoming parameter name but it is no requirement. This is done by simply calling the `export()` method that's defined in `AgaviValidator()`. This is how our finished validator looks like.

```
<?php

class Posts_PostValidator extends AgaviValidator
{
        /**
         * Validates the input
         *
         * @return      bool The input is valid number according to given
parameters.
         */
        protected function validate()
        {
                $parameterName = $this->getArgument();
                $postId = $this->getData($parameterName);

                $manager = $this->getContext()->getModel('PostManager',
'Posts');

                $post = $manager->retrieveById($postId);

                if (null == $post)
                {
                        $this->throwError();
                        return false;
                }

                $this->export($post);
                return true;
        }
}

?>
```

Now we need to make the class known to the framework. This is most easily done by adding it to the autoloader config you'll find in `app/modules/Posts/config/autoload.xml`. Add the following line to the end of the autoload definitions:

```
<autoload
name="Posts_PostValidator">%core.module_dir%/Posts/lib/validator/PostValidator.class.php</autoload>
```

The complete file should look like this:

```
<?xml version="1.0" encoding="UTF-8"?> <ae:configurations
xmlns="http://agavi.org/agavi/config/parts/autoload/1.0"
xmlns:ae="http://agavi.org/agavi/config/global/envelope/1.0"> <ae:configuration>
<autoload
name="BlogPostsBaseAction">%core.module_dir%/Posts/lib/action/BlogPostsBaseAction.class.php</auto
<autoload
name="BlogPostsBaseModel">%core.module_dir%/Posts/lib/model/BlogPostsBaseModel.class.php</auto
<autoload
name="BlogPostsBaseView">%core.module_dir%/Posts/lib/view/BlogPostsBaseView.class.php</autoloac
<autoload
name="Posts_PostValidator">%core.module_dir%/Posts/lib/validator/PostValidator.class.php</autoload>
</ae:configuration> </ae:configurations>
```

Now that the class is known to the frameworks autoloader we need to adapt our validation config to use the new validator and configure where to export the post object. We'll just use "post" as name, as the incoming parameter name. Adapt the `app/modules/Posts/validate/Post/Show.xml` to look like this:

```
<?xml version="1.0" encoding="UTF-8"?> <ae:configurations
xmlns="http://agavi.org/agavi/config/parts/validators/1.0"
xmlns:ae="http://agavi.org/agavi/config/global/envelope/1.0"
parent="%core.module_dir%/Posts/config/validators.xml" > <ae:configuration>
<validators> <validator class="Posts_PostValidator" name="post"> <arguments>
<argument>post</argument> </arguments> <errors> <error>The parameter post must
contain a valid post id</error> </errors> <ae:parameters> <ae:parameter
name="export">post</ae:parameter> </ae:parameters> </validator> </validators>
</ae:configuration> </ae:configurations>
```

So, kick out the redundant code from the action and we're done. While we could just leave the `Posts_Post_ShowAction::executeRead()` method empty and retrieve the post in the view itself it's better practice to do it in the action. If we're to change something about the logic how a post is retrieved, maybe add some checks that the post may or may not be displayed etc, we'd have to change the action and the view as opposed to only changing the action. The `Posts_Post_ShowAction::executeRead()` should now look like this:

```
public function executeRead(AgaviRequestDataHolder $rd) { $this->setAttribute('post',
$rd->getParameter('post')); return 'Success'; }
```

Polishing it up

Now that we have the basics working just fine we should do some polishing. We'd like the title of the post to be displayed in the url to make our friends from the SEO department happy and we'll make the whole blog look a little better. Let's start with the looks.

You might have noticed that our templates consisted of only the most basic html code needed to display the list or respectively the post. However, when you looked at the page there was a title tag that we were able to manipulate through a template variable and the page title was also set. If you had a look at the HTML source you'd have noticed that it is indeed valid xhtml. How did that happen?

Old style PHP applications often made use of included snippets of HTML code, often called header and footer. Those were included in the templates to factor out common html code. Agavi has a mechanism to factor out common code but it works far better than using includes. It's called layers. When we created the application using the agavi build script a very basic set of layers was created and configured for us. In the following chapter we'll go through those to understand what this is all about.

We'll be using the open source Emplode template by Arcsin, which can be obtained from the `resources/template/` directory of this tutorial or from http://templates.arcsin.se/emplode-website-template/. The result of this chapter is the stage4 application.

Layers and Layouts

Layers are templates that are organized in a matryoshka doll like fashion. Each layer wraps the one a level deeper and has access to the full rendered content and all it's template variables. The innermost layer is the template belonging to the view that was executed. The outer layers are called decorators because it wraps around the output of the inner layer adding content and thus decorating it.

Layers can be created at runtime in the view or predefined. A predefined set of layers is called a layout. Layouts are tied to an output type (we'll explain later what that is) and thus defined in `app/config/output_types.xml`. Let's go and have a look at the relevant section of our output_types.xml:

```
<layouts default="standard"> <!-- standard layout with a content and a decorator layer -->
<layout name="standard"> <!-- content layer without further params. this means the
standard template is used, i.e. the one with the same name as the current view --> <layer
name="content" /> <!-- decorator layer with the HTML skeleton, navigation etc; set to a
specific template here --> <layer name="decorator"> <ae:parameter
name="directory">%core.template_dir%</ae:parameter> <ae:parameter
name="template">Master</ae:parameter> </layer> </layout> <!-- another example layout
that has an intermediate wrapper layer in between content and decorator --> <!-- it also
shows how to use slots etc --> <layout name="wrapped"> <!-- content layer without
further params. this means the standard template is used, i.e. the one with the same
name as the current view --> <layer name="content" /> <layer name="wrapper"> <!-- use
CurrentView.wrapper.php instead of CurrentView.php as the template for this one -->
<ae:parameter name="extension">.wrapper.php</ae:parameter> </layer> <!-- decorator
layer with the HTML skeleton, navigation etc; set to a specific template here --> <layer
name="decorator"> <ae:parameter
name="directory">%core.template_dir%</ae:parameter> <ae:parameter
name="template">Master</ae:parameter> <!-- an example for a slot --> <slot
name="nav" module="Default" action="Widgets.Navigation" /> </layer> </layout> <!--
special layout for slots that only has a content layer to prevent the obvious infinite loop
that would otherwise occur if the decorator layer has slots assigned in the layout; this is
```

loaded automatically by ProjectBaseView::setupHtml() in case the current container is run as a slot --> <layout name="simple"> <layer name="content" /> </layout> </layouts>

Let's not be confused by the whole block and concentrate on the simplest layout defined in this section.

<layout name="simple"> <layer name="content" /> </layout>

This layout defines a single layer named "content". It does not pass any other parameters to the layer, most importantly no template is tied to the layer. That means that the template for the current view will be used. This layer would produce only the template's html code as output. We'll use that later on.

The <layouts> tag defines a default layout to use, it's called standard. This is the one we're currently using everywhere, so let's explain what this is about.

<layout name="standard"> <layer name="content" /> <layer name="decorator"> <ae:parameter name="directory">%core.template_dir%</ae:parameter> <ae:parameter name="template">Master</ae:parameter> </layer> </layout>
This layout defines two layers, one named "content" that works like the one in the simple layout we've seen above. In addition, it defines a second layer called "decorator" that wraps around the first layer. For this layer we're specifying two parameters - the template's name and the directory to search the template in. The directive '%core.template_dir%' resolves to `app/templates`, so let's have a look at `app/templates/Master.php`.
Tip: Note that the layer's name must be unique within the defined layout but multiple layouts can use the same name. In general it's good practice to use the same name for layers performing the same function.

<!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> <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/> <base href="<?php echo $ro->getBaseHref(); ?>" /> <title><?php if(isset($t['_title'])) echo htmlspecialchars($t['_title']) . ' - '; echo AgaviConfig::get('core.app_name'); ?></title> </head> <body> <?php if(isset($t['_title'])) echo '<h1>' . htmlspecialchars($t['_title']) . '</h1>'; ?> <?php echo $inner; ?> </body> </html>

See how the decorator template accesses the variable _title that we modified ealier - all outer templates can access the the template variables set by the view and the action. Note the special variable $inner - it contains the output of the content-layer. In general $inner contains the output of the previously rendered template with a single exception: for the innermost template ("content" in this case), $inner contains the return value of the view's `executeHtml()` method.

The last defined layout is a little more complex and we'll only explain it partially. Don't worry, we'll come back later to explain.

```
                                        <layout name="wrapped">
                                                <layer name="content" />
                                                <layer name="wrapper">
                                                        <ae:parameter
name="extension">.wrapper.php</ae:parameter>
                                                </layer>
                                                <layer name="decorator">
                                                        <ae:parameter
name="directory">%core.template_dir%</ae:parameter>
                                                        <ae:parameter
name="template">Master</ae:parameter>
                                                        <slot name="nav"
module="Default" action="Widgets.Navigation" />
                                                </layer>
                                        </layout>
```

Let's forget about the <slot> definition and focus on the rest. We've seen the content and the decorator template before, those should be clear. There's a third layer called "wrapper" that uses the standard template name (the view's name) but a different extension. So in case our view is named PostSuccessView the layer's template will be PostSuccess.wrapper.php.

Loading Layouts

Loading layouts is a simple call to `AgaviView::loadLayout(<layoutname>)`. If you pass "null" as the layoutname, the configured default layout will be used. The applications baseview that was created when we first created our application does this in it's setupHtml() method.

Applying our Layout

So let's go apply the pieces from the emplode template to our blog. First we need to copy the ressources like images and css files to our public directory. We'll just copy all images to `pub/img` and the style.css to `pub/css/style.css`. As we're moving the stylesheet to a different location we'll need to adjust it a little, the stylesheet directives referring to 'img' must be changed to '../img'.

Next we'll add the stylesheet to our decorator template in `app/templates/Master.php`. You'll notice that the master template reads the base href from the routing so that's taken care of. All we need to to is insert the link tag. The Master.php should now look like this:

```
<!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> <meta
http-equiv="Content-Type" content="text/html; charset=utf-8"/> <base href="<?php echo
$ro->getBaseHref(); ?>" /> <title><?php if(isset($t['_title'])) echo
htmlspecialchars($t['_title']) . ' - '; echo AgaviConfig::get('core.app_name'); ?></title> <link
rel="stylesheet" type="text/css" href="css/style.css" /> </head> <body> <?php
if(isset($t['_title'])) echo '<h1>' . htmlspecialchars($t['_title']) . '</h1>'; ?> <?php echo
$inner; ?> </body> </html>
```

So let's add the required html-code. We don't have most of the elements yet so for the header, navigation and all missing elements we'll keep the raw html source:

```
<!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> <meta
http-equiv="Content-Type" content="text/html; charset=utf-8"/> <base href="<?php echo
$ro->getBaseHref(); ?>" /> <title><?php if(isset($t['_title'])) echo
htmlspecialchars($t['_title']) . ' - '; echo AgaviConfig::get('core.app_name'); ?></title> <link
rel="stylesheet" type="text/css" href="css/style.css" /> </head> <body> <div id="header">
<div class="center_wrapper"> <div id="toplinks"> <div id="toplinks_inner"> <a
href="#">Sitemap</a> | <a href="#">Privacy Policy</a> | <a href="#">FAQ</a> </div>
</div> <div class="clearer">&nbsp;</div> <div id="site_title"> <h1><a
href="#">Bloggie</a></h1> <p>A demo blog application built on agavi</p> </div> </div>
</div> <div id="navigation"> <div class="center_wrapper"> <ul> <li
class="current_page_item"><a href="index.html">Home</a></li> <li><a
href="blog_post.html">Blog Post</a></li> <li><a href="style_demo.html">Style
Demo</a></li> <li><a href="archives.html">Archives</a></li> <li><a
href="empty_page.html">Empty Page</a></li> </ul> <div class="clearer">&nbsp;</div>
</div> </div> <div id="main_wrapper_outer"> <div id="main_wrapper_inner"> <div
class="center_wrapper"> <div class="left" id="main"> <div id="main_content"> <!-- this is
where the main content will be placed --> <?php echo $inner; ?> </div> </div> <div
class="right" id="sidebar"> <div id="sidebar_content"> <div class="box"> <div
class="box_title">About</div> <div class="box_content"> Aenean sit amet dui at felis
lobortis dignissim. Pellentesque risus nibh, feugiat in, convallis id, congue ac, sem. Sed
tempor neque in quam. </div> </div> <div class="box"> <div
```

class="box_title">Categories</div> <div class="box_content"> <ul> <li><a href="http://templates.arcsin.se/category/website-templates/">Website Templates</a></li> <li><a href="http://templates.arcsin.se/category/wordpress-themes/">Wordpress Themes</a></li> <li><a href="http://templates.arcsin.se/professional-templates/">Professional Templates</a></li> <li><a href="http://templates.arcsin.se/category/blogger-templates/">Blogger Templates</a></li> <li><a href="http://templates.arcsin.se/category/joomla-templates/">Joomla Templates</a></li> </ul> </div> </div> <div class="box"> <div class="box_title">Resources</div> <div class="box_content"> <ul> <li><a href="http://templates.arcsin.se/">Arcsin Web Templates</a></li> <li><a href="http://www.google.com/">Google</a> - Web Search</li> <li><a href="http://www.w3schools.com/">W3Schools</a> - Online Web Tutorials</li> <li><a href="http://www.wordpress.org/">WordPress</a> - Blog Platform</li> <li><a href="http://cakephp.org/">CakePHP</a> - PHP Framework</li> </ul> </div> </div> <div class="box"> <div class="box_title">Gallery</div> <div class="box_content"> <div class="thumbnails"> <a href="#" class="thumb"><img src="sample-thumbnail.jpg" width="75" height="75" alt="" /></a> <a href="#" class="thumb"><img src="sample-thumbnail.jpg" width="75" height="75" alt="" /></a> <a href="#" class="thumb"><img src="sample-thumbnail.jpg" width="75" height="75" alt="" /></a> <a href="#" class="thumb"><img src="sample-thumbnail.jpg" width="75" height="75" alt="" /></a> <a href="#" class="thumb"><img src="sample-thumbnail.jpg" width="75" height="75" alt="" /></a> <a href="#" class="thumb"><img src="sample-thumbnail.jpg" width="75" height="75" alt="" /></a> <div class="clearer">&nbsp;</div> </div> </div> </div> </div> </div> <div class="clearer">&nbsp;</div> </div> </div> </div> <div id="dashboard"> <div id="dashboard_content"> <div class="center_wrapper"> <div class="col3 left"> <div class="col3_content"> <h4>Tincidunt</h4> <ul> <li><a href="#">Consequat molestie</a></li> <li><a href="#">Sem justo</a></li> <li><a href="#">Semper eros</a></li> <li><a href="#">Magna sed purus</a></li> <li><a href="#">Tincidunt morbi</a></li> </ul> </div> </div> <div class="col3mid left"> <div class="col3_content"> <h4>Fermentum</h4> <ul> <li><a href="#">Semper fermentum</a></li> <li><a href="#">Sem justo</a></li> <li><a href="#">Magna sed purus</a></li> <li><a href="#">Tincidunt nisl</a></li> <li><a href="#">Consequat molestie</a></li> </ul> </div> </div> <div class="col3 right"> <div class="col3_content"> <h4>Praesent</h4> <ul> <li><a href="#">Semper lobortis</a></li> <li><a href="#">Consequat molestie</a></li> <li><a href="#">Magna sed purus</a></li> <li><a href="#">Sem morbi</a></li> <li><a href="#">Tincidunt sed</a></li> </ul> </div> </div> <div class="clearer">&nbsp;</div> </div> </div> </div> <div id="footer"> <div class="center_wrapper"> <div class="left"> &copy; 2008 Website.com - Your Website Slogan </div> <div class="right"> <a href="http://templates.arcsin.se/">Website template</a> by <a href="http://arcsin.se/">Arcsin</a> </div> <div class="clearer">&nbsp;</div> </div> </div> </body> </html>

Far better, but not perfect yet. We'll need to adapt the Index template and the Post.Show template as well. The `app/modules/Posts/templates/IndexSuccess.php` should look like this:

```
<?php
foreach ($t['posts'] as $post)
{
?>
<div class="post">

        <div class="post_title"><h2><a href="<?php echo $post['url'];
?>"><?php echo htmlspecialchars($post['title']); ?></a></h2></div>
        <div class="post_date">Posted on <?php echo $post['posted']; ?>
by <a href="#"><?php echo htmlspecialchars($post['author_name']);
?></a></div>

        <div class="post_body">
                <?php echo $post['content']; ?>
        </div>

        <div class="post_meta">
```

```
                <a href="#">5 comments</a> | Tagged: <?php echo
htmlspecialchars($post['category_name']); ?>
        </div>

</div>
<?php
}
?>
```

And for the Post.Show page we'll leave the comments out for the moment. So `app/modules/Posts/templates/Posts/ShowSuccess.php` should look like this:

```
<?php
// alias the post, to make access shorter
$post = $t['post'];
?>
<div class="post">

        <div class="post_title"><h1><?php echo
htmlspecialchars($post['title']); ?></h1></div>
        <div class="post_date">Posted on <?php
htmlspecialchars($post['posted']); ?> by <a href="#"><?php echo
htmlspecialchars($post['author_name']); ?></a></div>

        <div class="post_body">
                <?php echo $post['content']; ?>
        </div>

        <div class="post_meta">
                Tagged: <a href="#"><?php echo
htmlspecialchars($post['category_name']); ?></a>
        </div>

</div>
```

That's it for now.

Note: There's different opinions about whether it's better to first skin the application and then continue building it or the other way round. We'd recommend to first build major parts of the application and then skin it. Just add a rough layout and then move on to implementing functionality. That way you can have someone test the clickflows and the overall logic while you make it look good. However, for the course of this tutorial that would imply that you'd have to learn a lot of more difficult things before actually returning to this point.

What are Slots

Slots are a named placeholder in your template that's tied to the result of a specific action execution. Slots can be used to create reusable components for your project or to split up a page into manageable parts. Slots can be registered at runtime or by using a layout. Slots are registered on a template layer and must have a name that's unique for this layer. All slot output is collected in an associative array named *$slots* with the slot's name as key.

Slots are executed after the main action's view has been executed but before the template is rendered. The main action's view can feed arbitrary data as parameter to the slot and the slot can access the global request data. The slot's execution environment is otherwise separated from the main actions or any other slots execution environment, most important is that the slot has no access to the main actions template variables - unlike the decorator layers.

Slots are often called "partials" in other frameworks

Registering a Slot in a layout.

Often slots are used to structure an application and factor out common logic and html-code from individual actions. A slot that should be available on all pages and does not depend on input from the main action should be registered in the default layout - a standard example for such a slot is the navigation - it usually shares it's logic across all pages and usually only depends on the matched route. Let's extract our static navigation and register it as a slot for our default layout.

A slot is a regular action so we create one. However it's a good idea to move your shared elements to a separate module so we invoke the module wizard:

```
bloggie$ dev/bin/agavi module-wizard
Module name: Widgets
Space-separated list of actions to create for Widgets: Navigation
Space-separated list of views to create for Navigation [Success]:
bloggie$
```

So now that we created a new action we need to register it as a slot with ourstandard layout in `app/config/output_types.xml`:

<!-- standard layout with a content and a decorator layer --> <layout name="standard"> <!-- content layer without further params. this means the standard template is used, i.e. the one with the same name as the current view --> <layer name="content" /> <!-- decorator layer with the HTML skeleton, navigation etc; set to a specific template here --> <layer name="decorator"> <ae:parameter name="directory">%core.template_dir%</ae:parameter> <ae:parameter name="template">Master</ae:parameter> <slots> <slot name="navigation" module="Widgets" action="Navigation" /> </slots> </layer> </layout>

The <slot> tag registers a slot named "navigation" on the decorator layer with the action named "Navigation" located in the module "Widgets". Whenever the decorator layer is rendered this action is executed and the output can be inserted into the layer's rendering. So now we can remove the HTML code for the navigation and place it in the template for the Navigation action. The template `app/modules/Widgets/templates/NavigationSuccess.php` should now look like this:

<div id="navigation"> <div class="center_wrapper"> <ul> <li class="current_page_item"><a href="index.html">Home</a></li> <li><a href="blog_post.html">Blog Post</a></li> <li><a href="style_demo.html">Style Demo</a></li> <li><a href="archives.html">Archives</a></li> <li><a href="empty_page.html">Empty Page</a></li> </ul> <div class="clearer">&nbsp;</div> </div> </div>

and the respective part in our decorator template `app/templates/Master.php` must be replaced by the slot output:

<!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> <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/> <base href="<?php echo $ro->getBaseHref(); ?>" /> <title><?php if(isset($t['_title'])) echo htmlspecialchars($t['_title']) . ' - '; echo AgaviConfig::get('core.app_name'); ?></title> <link rel="stylesheet" type="text/css" href="css/style.css" /> </head> <body> <div id="header"> <div class="center_wrapper"> <div id="toplinks"> <div id="toplinks_inner"> <a href="#">Sitemap</a> | <a href="#">Privacy Policy</a> | <a href="#">FAQ</a> </div> </div> <div class="clearer">&nbsp;</div> <div id="site_title"> <h1><a href="#">Bloggie</a></h1> <p>A demo blog application built on agavi</p> </div> </div> </div> <?php echo $slots['navigation']; ?> <div id="main_wrapper_outer"> <div id="main_wrapper_inner"> <div class="center_wrapper"> ...

Done. Let's start and identify other commonly used elements. There's the header and the footer, the sidebar column and the dashboard at the bottom. Let's start extratcting those - the process is the same as with the navigation, so I'll keep that short. Let's first create four actions:

```
bloggie$ dev/bin/agavi action-wizard
Module name: Widgets
Action name: Header
Space-separated list of views to create for Header [Success]:
bloggie$
```

```
bloggie$ dev/bin/agavi action-wizard
Module name: Widgets
Action name: Footer
Space-separated list of views to create for Footer [Success]:
bloggie$
```

```
bloggie$ dev/bin/agavi action-wizard
Module name: Widgets
Action name: Dashboard
Space-separated list of views to create for Dashboard [Success]:
bloggie$
```

```
bloggie$ dev/bin/agavi action-wizard
Module name: Widgets
Action name: Sidebar
Space-separated list of views to create for Sidebar [Success]:
bloggie$
```

Next register the slots:

<!-- standard layout with a content and a decorator layer --> <layout name="standard"> <!-- content layer without further params. this means the standard template is used, i.e. the one with the same name as the current view --> <layer name="content" /> <!-- decorator layer with the HTML skeleton, navigation etc; set to a specific template here --> <layer name="decorator"> <ae:parameter name="directory">%core.template_dir%</ae:parameter> <ae:parameter name="template">Master</ae:parameter> <slots> <slot name="navigation" module="Widgets" action="Navigation" /> <slot name="header" module="Widgets" action="Header" /> <slot name="footer" module="Widgets" action="Footer" /> <slot name="dashboard" module="Widgets" action="Dashboard" /> <slot name="sidebar" module="Widgets" action="Sidebar" /> </slots> </layer> </layout>

and now move the HTML code to the repective templates.

`app/modules/Widgets/templates/HeaderSuccess.php`:

<div id="header"> <div class="center_wrapper"> <div id="toplinks"> <div id="toplinks_inner"> <a href="#">Sitemap</a> | <a href="#">Privacy Policy</a> | <a href="#">FAQ</a> </div> </div> <div class="clearer">&nbsp;</div> <div id="site_title"> <h1><a href="#">Bloggie</a></h1> <p>A demo blog application built on agavi</p> </div> </div> </div>

`app/modules/Widgets/templates/FooterSuccess.php`:

<div id="footer"> <div class="center_wrapper"> <div class="left"> &copy; 2008 Website.com - Your Website Slogan </div> <div class="right"> <a

href="http://templates.arcsin.se/">Website template</a> by <a href="http://arcsin.se/">Arcsin</a> </div> <div class="clearer">&nbsp;</div> </div> </div>

`app/modules/Widgets/templates/DashboardSuccess.php`:

```
<div id="dashboard">
        <div id="dashboard_content">
                <div class="center_wrapper">

                        <div class="col3 left">
                                <div class="col3_content">

                                        <h4>Tincidunt</h4>
                                        <ul>
                                                <li><a href="#">Consequat
molestie</a></li>
                                                <li><a href="#">Sem
justo</a></li>
                                                <li><a href="#">Semper
eros</a></li>
                                                <li><a href="#">Magna sed
purus</a></li>
                                                <li><a href="#">Tincidunt
morbi</a></li>
                                        </ul>

                                </div>
                        </div>

                        <div class="col3mid left">
                                <div class="col3_content">

                                        <h4>Fermentum</h4>
                                        <ul>
                                                <li><a href="#">Semper
fermentum</a></li>
                                                <li><a href="#">Sem
justo</a></li>
                                                <li><a href="#">Magna sed
purus</a></li>
                                                <li><a href="#">Tincidunt
nisl</a></li>
                                                <li><a href="#">Consequat
molestie</a></li>
                                        </ul>

                                </div>
                        </div>

                        <div class="col3 right">
                                <div class="col3_content">

                                        <h4>Praesent</h4>
                                        <ul>
                                                <li><a href="#">Semper
lobortis</a></li>
                                                <li><a href="#">Consequat
molestie</a></li>
                                                <li><a href="#">Magna sed
purus</a></li>
                                                <li><a href="#">Sem
morbi</a></li>
                                                <li><a href="#">Tincidunt
sed</a></li>
                                        </ul>

                                </div>
                        </div>

                        <div class="clearer">&nbsp;</div>

                </div>
        </div>
</div>
```

`app/modules/Widgets/templates/SidebarSuccess.php`:

<div class="right" id="sidebar"> <div id="sidebar_content"> <div class="box"> <div class="box_title">About</div> <div class="box_content"> Aenean sit amet dui at felis lobortis dignissim. Pellentesque risus nibh, feugiat in, convallis id, congue ac, sem. Sed tempor neque in quam. </div> </div> <div class="box"> <div

```
class="box_title">Categories</div> <div class="box_content"> <ul> <li><a
href="http://templates.arcsin.se/category/website-templates/">Website
Templates</a></li> <li><a
href="http://templates.arcsin.se/category/wordpress-themes/">Wordpress
Themes</a></li> <li><a
href="http://templates.arcsin.se/professional-templates/">Professional
Templates</a></li> <li><a
href="http://templates.arcsin.se/category/blogger-templates/">Blogger
Templates</a></li> <li><a
href="http://templates.arcsin.se/category/joomla-templates/">Joomla Templates</a></li>
</ul> </div> </div> <div class="box"> <div class="box_title">Resources</div> <div
class="box_content"> <ul> <li><a href="http://templates.arcsin.se/">Arcsin Web
Templates</a></li> <li><a href="http://www.google.com/">Google</a> - Web Search</li>
<li><a href="http://www.w3schools.com/">W3Schools</a> - Online Web Tutorials</li>
<li><a href="http://www.wordpress.org/">WordPress</a> - Blog Platform</li> <li><a
href="http://cakephp.org/">CakePHP</a> - PHP Framework</li> </ul> </div> </div> <div
class="box"> <div class="box_title">Gallery</div> <div class="box_content"> <div
class="thumbnails"> <a href="#" class="thumb"><img src="sample-thumbnail.jpg"
width="75" height="75" alt="" /></a> <a href="#" class="thumb"><img
src="sample-thumbnail.jpg" width="75" height="75" alt="" /></a> <a href="#"
class="thumb"><img src="sample-thumbnail.jpg" width="75" height="75" alt="" /></a> <a
href="#" class="thumb"><img src="sample-thumbnail.jpg" width="75" height="75" alt=""
/></a> <a href="#" class="thumb"><img src="sample-thumbnail.jpg" width="75"
height="75" alt="" /></a> <a href="#" class="thumb"><img src="sample-thumbnail.jpg"
width="75" height="75" alt="" /></a> <div class="clearer">&nbsp;</div> </div> </div>
</div> </div> </div>
```

And finally our modified `app/modules/templates/Master.php`:

```
<!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> <meta
http-equiv="Content-Type" content="text/html; charset=utf-8"/> <base href="<?php echo
$ro->getBaseHref(); ?>" /> <title><?php if(isset($t['_title'])) echo
htmlspecialchars($t['_title']) . ' - '; echo AgaviConfig::get('core.app_name'); ?></title> <link
rel="stylesheet" type="text/css" href="css/style.css" /> </head> <body> <?php echo
$slots['header']; ?> <?php echo $slots['navigation']; ?> <div id="main_wrapper_outer">
<div id="main_wrapper_inner"> <div class="center_wrapper"> <div class="left"
id="main"> <div id="main_content"> <!-- this is where the main content will be placed -->
<?php echo $inner; ?> </div> </div> <?php echo $slots['sidebar']; ?> <div
class="clearer">&nbsp;</div> </div> </div> </div> <?php echo $slots['dashboard']; ?>
<?php echo $slots['footer']; ?> </body> </html>
```

That's far shorter and nicely displays how the whole page is constructed from small elements. Looks like we have everything accounted for, do we? No, wait.

Dynamically Registering a Slot in a View

There's another set of elements that are very similar. Let's have a look at the display of posts on the front page and on the post's detail page. Overall, they look pretty similar, however theres some minor differences:

- The number of comments is listed on the frontpage, on the detail page we want to display the comments themselves.
- The headline tag is h2 on the frontpage while it is h1 on the detail page.
- The headline is linked on the frontpage while it is not on the detail page.

Still, we'd really like to factor that out. If we could only use the detail page for the list display as well. Well, it turns out we can. A view can register arbitrary slots on any defined layer and pass any parameter it wishes. Here's how it works. We change our `app/modules/Posts/views/IndexSuccessView.class.php` to register the slots on it's main layer which is named "content":

<?php class Posts_IndexSuccessView extends BlogPostsBaseView { public function executeHtml(AgaviRequestDataHolder $rd) { $this->setupHtml($rd); $contentLayer = $this->getLayer('content'); $ro = $this->getContext()->getRouting(); $posts = array(); foreach($this->getAttribute('posts') as $p) { /* register one slot per post */ $contentLayer->setSlot('post'.$p->getId(), $this->createSlotContainer('Posts', 'Post.Show', array('post' => $p->getId()))); } $this->setAttribute('_title', 'Latest Posts'); } } ?>

and then change the template in `app/modules/Posts/templates/IndexSuccess.php` to just display the slots:

```
<?php
foreach ($slots as $slot)
{
        echo $slot;
}
?>
```

The call to `AgaviView::getLayer()` retrieves the layer named 'content' from the list of configured layers. The call to `AgaviView::createSlotContainer()` creates a new slotcontainer that is then registered on the layer via `AgaviTemplateLayer::setSlot()`. We create a unique name for each slot we're creating and we're passing the post-id as parameter when creating the slot.

That takes care of most of our problems - if you have a look at the index page you'll see that the list of posts is nicely displayed - with the mentioned drawbacks. The headline is <h1> and not <h2>, the number of comments not shown. Let's take care of that.

We need a way to distinguish where we are - is the detail page included as a slot or is it the main action. We could either pass in a new parameter or find something else. Well, there is already something else. Each action execution is isolated in an `AgaviExecutionContainer` and it does have access to that container. Creating a slot container automagically sets a parameter on the container and we can read that parameter in our view and react accordingly (`app/modules/Posts/views/Post/ShowSuccessView.class.php`):

<?php class Posts_Post_ShowSuccessView extends BlogPostsBaseView { public function executeHtml(AgaviRequestDataHolder $rd) { $this->setupHtml($rd); $p = $this->getAttribute('post'); $post = $p->toArray(); $post['url'] = $this->getContext()->getRouting()->gen('posts.post.show', array('post' => $p->getId())); $this->setAttribute('post', $post); $isList = $this->getContainer()->getParameter('is_slot', false); if($isList) { $headlineSize = 2; $linkHeadline = true; $displayComments = false; } else { $headlineSize = 1; $linkHeadline = false; $displayComments = true; } $this->setAttribute('headline_size', $headlineSize); $this->setAttribute('link_headline', $linkHeadline); $this->setAttribute('display_comments', $displayComments); $this->setAttribute('_title', $post['title']); } } ?>

Now we only need to adapt the template in `app/modules/Posts/templates/Post/ShowSuccess.php` accordingly and we're all set:

<?php // alias the post, to make access shorter $post = $t['post']; $headline = htmlspecialchars($post['title']); if($t['link_headline']) { $headline = sprintf('<a href="%1$s">%2$s</a>', $post['url'], $headline); } $headline = sprintf('<h%1$s>%2$s</h%1$s>', $t['headline_size'], $headline); ?> <div class="post"> <div class="post_title"><?php echo $headline; ?></div> <div class="post_date">Posted on <?php htmlspecialchars($post['posted']); ?> by <a href="#"><?php echo htmlspecialchars($post['author_name']); ?></a></div> <div class="post_body"> <?php echo $post['content']; ?> </div> <div class="post_meta"> <?php if(!$t['display_comments']) { ?> <a href="#">5 comments</a> | <?php } ?> Tagged: <a href="#"><?php echo htmlspecialchars($post['category_name']); ?></a> </div> </div>

Done. Fire up your browser and check - things should be just fine.

Slots, Request Data access and Validation

The usual rules for validation apply for slots as well: No access to any piece of request data unless it's validated. This includes any kind of request data, the global data passed by the client and the additional arguments passed when registering the slot. The above piece of code only works because we already have validation defined for the parameter named "post".

However, there is an exception: Slots that are marked as "simple" have full access to all arguments that were passed at registration time even without validation. Simple slots however have no access at all to the global request data passed by the client.

Slots and Layouts

Slots can use layouts themselves. However, there's some things that you need to take into account. If slot A uses a layout that includes slot B and slot B uses a layout that includes slot A you'll get an infinite loop. That's called a layout recusion. If you ever get an error message saying "Too many execution runs detected" chances are high that you've just created a layout recursion.

The generated projectspecific BaseView contains some minimal protection against layout recursions: In case it detects that it's run inside a slot it attempts to load a different standard layout - the one named 'slot' instead of the configured default.

Slots and Performance

On the first sight slots lower the performance. They introduce some management overhead, consume a little memory for a couple of extra objects and so on. However benchmarks show that the overhead is very low and is easily countered by increased developer productivity (remember that developer time is way more expensive than cpu time). There are also other effects that counter the overhead introduced by slots. Slots can be individually cached by agavi, each one with it's own set of rules and cache times. This raises performance by a far larger margin than the initial overhead. Pages using slots scale far better than pages without.

However, there are some things to keep in mind when splitting up your layouts into slots:

- Often slots are used for display purposes only and have no own logic. The actions belonging to those slots should be marked simple to reduce the overhead to the lowest possible level. This is the case for our widgets - Header, Footer, Navigation, Dashboard and Sidebar slots.
- Be careful with validation, running full validation each time you call a slot may bring your page to a grinding halt, especially when the validation is an expensive operation such as contacting the database or a webservice.

So let's check our application. We haven't marked our Widgets as simple, so we should do that now but instead of implementing isSimple() on all of those four classes we just go to our `app/modules/Widgets/lib/action/BlogWidgetsBaseAction.class.php` and implement it there. All actions in the Widgets module extend this base class and we can safely assume that pretty much all Widgets are for display only and thus never will contain any business logic - if an exception comes along we can still overwrite the behavior in that specific class.

<?php /** * The base action from which all Widgets module actions inherit. */ class BlogWidgetsBaseAction extends BlogBaseAction { /** * Whether or not this action is "simple", i.e. doesn't use validation etc. * * @return bool true, if this action should act in simple mode, or false. */ public function isSimple() { return false; } } ?>

That takes care of disabling all validation and action execution for all Widgets. That validation topic however is a bit a tricky one. Think about it: our IndexAction does a query to retrieve a list of valid posts and then, when the slots are executed the validation once again does a query to check each single one. We're just doing array lookups so far, that's fast but what if we'd have to connect to a SOAP service that needs 0.5 seconds for a reply? That would be 5 seconds overhead for a list of 10 items, that's way to much and it's completely wasted, we ourselves pass the post and we do know it's valid.

We could perhaps implement our `Posts_PostShowAction::isSimple()` in a way that returns true if the action is called in a slot and false if not. Something along the lines of

/** * Whether or not this action is "simple", i.e. doesn't use validation etc. * * @return bool true, if this action should act in simple mode, or false. */ public function isSimple() { if($this->getContainer()->getParameter('is_slot', false)) { return true; } return false; }

That's a technique that works often but it's not going to help us here - we do some work in our executeRead() method and marking the action as "simple" would skip that. But we can change our validation. We modify our validator so that if the parameter is already an object type PostModel it just short-circuits and returns true. No client could ever pass a valid object via GET or POST so it must have been passed from within the framework. And if we ourselves pass an invalid object, well then no validation is going to safe us. Let's change the validator in `app/modules/Posts/lib/validator/PostValidator.class.php`:

<?php class Posts_PostValidator extends AgaviValidator { /** * Validates the input * * @return bool The input is valid number according to given parameters. */ protected function validate() { $parameterName = $this->getArgument(); $data = $this->getData($parameterName); if($data instanceof Posts_PostModel) { $post = $data; } else { $manager = $this->getContext()->getModel('PostManager', 'Posts'); $post = $manager->retrieveById($data); if (null == $post) { $this->throwError(); return false; } } $this->export($post); return true; } } ?>

Now we need to change the `Posts_PostIndexSuccessView::executeHtml()` to pass a full `Posts_PostModel` object instead of the id:

```
<?php

class Posts_IndexSuccessView extends BlogPostsBaseView
{
        public function executeHtml(AgaviRequestDataHolder $rd)
        {
                $this->setupHtml($rd);

                $contentLayer = $this->getLayer('content');

                $ro = $this->getContext()->getRouting();

                $posts = array();

                foreach($this->getAttribute('posts') as $p)
                {
                        /* register one slot per post */
                        $contentLayer->setSlot('post'.$p->getId(),
$this->createSlotContainer('Posts', 'Post.Show', array('post' => $p)));
                }

                $this->setAttribute('_title', 'Latest Posts');
        }
}

?>
```

and that's it. While this does not remove the full validation it still removes most of the overhead introduced by it. This technique is often used on dual-purpose actions.

Adding the post's title to the url

Adding the post's title to the url has multiple advantages. It allows the user to see where the url is pointing to and search engines will improve your ranking.

The first thing we need to do is adapt our routing rules to allow for a second parameter in the url. We'd like the title to be an optional part and we'll restrict it to word-chars and dashes only:

```
                                <route name=".post"
pattern="^/(post:\d+)(-{title:[-\w]+})?" action="Post">
                                        <route name=".show" pattern="^$"
action=".Show" />
                                </route>
```

Now that we made it possible to inject the title we just need to adapt the places where we generate the urls to actually pass the title. Currently this is only in the Posts_IndexSuccessView, so let's change that. Remember that we only allowed word chars in the route, so we need to make shure that we only pass word chars in - we'll just replace anything that's not a word char by a dash:

```
<?php

class Posts_IndexSuccessView extends BlogPostsBaseView
{

        public function executeHtml(AgaviRequestDataHolder $rd)
        {
                $this->setupHtml($rd);
                $ro = $this->getContext()->getRouting();

                $this->setAttribute('_title', 'Latest Posts');


                $posts = array();

                foreach($this->getAttribute('posts') as $p)
                {
                        $post = $p->toArray();
                        $post['url'] = $ro->gen(
                                'posts.post.show',
                                array(
                                        'post' => $p->getId(),
                                        'title' => preg_replace('/\W/',
'-', $p->getTitle())
                                )
                        );

                        $posts[] = $post;
                }

                $this->setAttribute('posts', $posts);
        }
}

?>
```

Hmm. That works, but it might get a bit tedious to always remember that if you generate the url to a post, you'll have to pass in the post-id and the title and replace all non-word chars by dashes. Well, surprise, there's a better way.

Routing Callbacks

A routing callback is a class that can intercept certain routing actions and modify parameters or other options. It is notified when

- A route is generated
- A route has been matched
- An attempt was made to match a route but it has not been matched

A routing callback is not notified if maching stopped before there was an attempt to match the route. The routing notifies the callbacks by calling the appropriate methods on the callback - `AgaviRoutingCallback::onGenerate()` when the route is being generated, `AgaviRoutingCallback::onMatched()` when the route matched, `AgaviRoutingCallback::onNotMatched()` in case an attempt to match the route failed. A route may define any number of callbacks that are executed in the order they're defined in, however excessive use of callbacks can slow your routing.

AgaviRoutingCallback::onGenerate()

This method is called when the route is about to be reverse generated into an URL. It can be used to modify the parameters and options passed to the call of `AgaviRouting::gen()` by the user. This method can be used to turn objects into routing parameters, strip undesired characters from parameters etc. If this method returns false the url-snippet generated by this route is not generated.

AgaviRoutingCallback::onMatched()

This method is called when the regular expression of the route matched. It may inspect and modify any of the parameters extracted from the route or add new parameters. If this method returns false, the route counts as not matched and all parameters modified will be reset. The callback may modify the execution container as well. If the callback returns a response object the resulting response is sent immediatly without any action execution. This behavior can be used for quick redirects.

AgaviRoutingCallback::onNotMatched()

This method is called when the regular expression of the route did not match. The callback may modify the execution container as well. If the callback returns a response object the resulting response is sent immediatly without any action execution. This behavior can be used for quick redirects.

Adding Callbacks to a Route

A list of callbacks may be added to a route by using the <callbacks> tag. The class for the callback must either be included before the routing is executed or more preferably added to the global autoload definition. Any number of parameters may be passed to the callback by adding <ae:parameter> tags as required.

<route name=".post" pattern="^/(post:\d+)(-{title:[-\w]+})?" action="Post"> <callbacks> <callback class="PostRoutingCallback" /> </callbacks> <route name=".show" pattern="^$" action=".Show" /> </route>

Routing Values

An `AgaviRoutingValue` is a value object that encapsulates all parameters passed to the routing. It's purpose is to store some meta information about the parameter value, mainly whether it's been encoded or not. All parameters passed to `AgaviRoutingCallback::onGenerate()` are wrapped in `AgaviRoutingValue` objects and all parameters changed by this method must be wrapped as well.

Using Callbacks for the Title in URLs

We can use a RoutingCallback's onGenerate method to factor out the code needed to generate a URL for the post's detail page. We'll just pass the PostModel in and the callback takes care of extracting the id and the title for the url. Let's create a callback in `app/modules/Posts/lib/routing/PostRoutingCallback.class.php`.

<?php class PostRoutingCallback extends AgaviRoutingCallback { /** * Gets executed when the route of this callback is about to be reverse * generated into an URL. * * @param array The default parameters stored in the route. * @param array The parameters the user supplied to AgaviRouting::gen(). * @param array The options the user supplied to AgaviRouting::gen(). * * @return bool Whether this route part should be generated. */ public function onGenerate(array $defaultParameters, array &$userParameters, array &$userOptions) { $post = $userParameters['post']->getValue(); $routing = $this->getContext()->getRouting(); $userParameters['post'] = $routing->createValue($post->getId()); $userParameters['title'] = $routing->createValue(preg_replace('/\W/', '-', $post->getTitle())); return true; } } ?>

The callback must be available in the global autoload file in `app/config/autoload.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
        <ae:configurations
xmlns="http://agavi.org/agavi/config/parts/autoload/1.0"
xmlns:ae="http://agavi.org/agavi/config/global/envelope/1.0"
parent="%core.system_config_dir%/autoload.xml">
                <ae:configuration>

                        <autoload
name="BlogBaseAction">%core.lib_dir%/action/BlogBaseAction.class.php</autoload>
                        <autoload
name="BlogBaseModel">%core.lib_dir%/model/BlogBaseModel.class.php</autoload>
                        <autoload
name="BlogBaseView">%core.lib_dir%/view/BlogBaseView.class.php</autoload>
                        <autoload
name="PostRoutingCallback">%core.module_dir%/Posts/lib/routing/PostRoutingCallback.clas

                </ae:configuration>
        </ae:configurations>
```

and registered in `app/config/routing.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
        <ae:configurations
xmlns:ae="http://agavi.org/agavi/config/global/envelope/1.0"
xmlns="http://agavi.org/agavi/config/parts/routing/1.0">
                <ae:configuration>
                        <routes>

                                <!-- default action for "/" -->
                                <route name="index" pattern="^/$"
module="Posts" action="%actions.default_action%" />

                                <route name="posts" pattern="^/posts"
module="Posts">
                                        <route name=".post"
pattern="^/(post:\d+)(-{title:[-\w]+})?" action="Post">
                                                <callbacks>
                                                        <callback
class="PostRoutingCallback" />
                                                </callbacks>
                                                <route name=".show"
pattern="^$" action=".Show" />
                                        </route>
                                </route>

                        </routes>
                </ae:configuration>
        </ae:configurations>
```

Now we need to pass a PostModel instead of id and title in our IndexSuccessView:

```
<?php

class Posts_IndexSuccessView extends BlogPostsBaseView
{

        public function executeHtml(AgaviRequestDataHolder $rd)
        {
                $this->setupHtml($rd);
                $ro = $this->getContext()->getRouting();

                $this->setAttribute('_title', 'Latest Posts');


                $posts = array();

                foreach($this->getAttribute('posts') as $p)
                {
                        $post = $p->toArray();
                        $post['url'] = $ro->gen(
                                'posts.post.show',
                                array(
                                        'post' => $p
                                )
                        );

                        $posts[] = $post;
                }
                $this->setAttribute('posts', $posts);
        }
}
?>
```

and that's it. Now the no part of the application itself needs to know about how the urls to the posts are contructed and which parameters are required. That's all nicely encapsulated in the routing.

Connecting to a database

So far we've been working with static data and it's about time we change that and connect our blog to a database. Otherwise we'd have a hard time giving the readers a way of commeting on our posts or creating an admin page where we can add or edit posts.

Unlike many other frameworks agavi does not supply a database abstraction library, instead it offers you the freedom of choice. Whether you prefer Propel over Doctrine, are in love with Zend_DB or ez_components, Agavi provides an appropriate adapter. If your poison of choice is something that agavi does not know about yet then it's easy to provide an adapter.

For sake of simplicity we'll be using plain PDO and handwritten SQL to connect to a database though in a serious project we'd be using a serious ORM.

The data files and schema description required for this step are located in the `dev/sql` directory of the stage5 app.

The Database Manager

All connections to databases are controlled by the database manager. It establishes all connections when started and provides access to the connections via `AgaviDatabaseManager::getDatabase()`.

The database subsystem is enabled by setting "use_database" to "true" in `app/config/settings.xml`:

```
... <settings> <setting name="app_name">Bloggie</setting> <setting
name="available">true</setting> <setting name="debug">false</setting> <setting
name="use_database">true</setting> <setting name="use_logging">false</setting>
<setting name="use_security">true</setting> <setting
name="use_translation">false</setting> </settings> ...
```

The connection data is then configured in `app/config/databases.xml`. We'll assume a mysql database named "bloggie" on localhost with the user "bloggie" and the password "bloggie".

```
<?xml version="1.0" encoding="UTF-8"?> <ae:configurations
xmlns:ae="http://agavi.org/agavi/config/global/envelope/1.0"
xmlns="http://agavi.org/agavi/config/parts/databases/1.0"> <ae:configuration
environment="development"> <databases default="pdo_mysql_main"> <database
name="pdo_mysql_main" class="AgaviPdoDatabase"> <ae:parameter
name="dsn">mysql:host=localhost;dbname=bloggie</ae:parameter> <ae:parameter
name="username">bloggie</ae:parameter> <ae:parameter
name="password">bloggie</ae:parameter> </database> </databases>
</ae:configuration> </ae:configurations>
```

This configures a single database connection named "pdo_mysql_main" that uses the `AgaviPdoDatabase` adapter. It also sets that connection as default connection.

Accessing a Database connection

A database connection can be accessed via `AgaviDatabaseManager::getDatabase($name)`. If "null" is passed as connection name the default connection is used.

Let's adapt our application to read it's data from that database. You'll see how it pays off that we structured our application nicely - all we need to do is adapt our

`Posts_PostManager` to read it's data from the appropriate tables:

<?php class Posts_PostManagerModel extends BlogPostsBaseModel { public function retrieveById($id) { $sql = 'SELECT p.*, a.screen_name AS author_name, c.name AS category_name FROM posts p LEFT JOIN admin_users a ON p.author_id = a.id LEFT JOIN categories c ON p.category_id = c.id WHERE p.id = ?'; $stmt = $this->getContext()->getDatabaseManager()->getDatabase()->getConnection()->prepare($sql); $stmt->bindValue(1, $id, PDO::PARAM_INT); $stmt->execute(); $result = $stmt->fetch(PDO::FETCH_ASSOC); if (false != $result) { return $this->getContext()->getModel('Post', 'Posts', array($result)); } return null; } public function retrieveLatest($limit = 5) { $sql = 'SELECT p.*, a.screen_name AS author_name, c.name AS category_name FROM posts p LEFT JOIN admin_users a ON p.author_id = a.id LEFT JOIN categories c ON p.category_id = c.id ORDER BY posted DESC LIMIT ?'; $stmt = $this->getContext()->getDatabaseManager()->getDatabase()->getConnection()->prepare($sql); $stmt->bindValue(1, $limit, PDO::PARAM_INT); $stmt->execute(); $result = $stmt->fetchAll(); foreach($result as $post) { $posts[] = $this->getContext()->getModel('Post', 'Posts', array($post)); } return $posts; } } ?>Well, that's it. We could certainly do a little cleanup and move the lengthy call to `$this->getContext()->getDatabaseManager()->getDatabase()->getConnection()` to an extra method but that won't change the principle. If you fire up your browser, you'll note no difference.

Handling multiple Representations

In this chapter we'll deal with handling multiple representations of the same ressource. The same list of posts or comments could be available as HTML, as a printable HTML version, as a whitelabel snippet for inclusion in other websites or as an rss feed. It would be annoying if we had to duplicate the application logic to handle this, so the framework offers a facility to deal with this exact problem. It's called "output types".

Output Types

Output types are the Agavi way of dealing with the requirement of having multiple representation formats for a ressource. A list of items could for example be delivered to the client as a regular plain html list, a json serialized array of objects, as a SOAP response or even rendered as an image or chart if required.

Output types are configured in `app/config/output_types.xml`. An output type configuration can specify a variety of parameters:

- A name. Each output type must have a unique name.
- A set of HTTP headers to send with the response, most notably the content-type header. These headers can be overwritten later but it's still good practice to define a reasonable default
- A specific exeption template to use in case an uncaught exception occurs.
- A renderer configuration, including the renderer to use and any parameters required for rendering.
- Any number of preconfigured layouts.

Except for the output type name all of these parameters are optional. There's quite a few output types where using a renderer wouldn't make sense - for example a json serialized response.

Let's have a look at the stock html output type from the standard generated application:

```
... <output_types default="html"> <output_type name="html"
exception_template="%core.template_dir%/exceptions/web-html.php"> <renderers
default="php"> <renderer name="php" class="AgaviPhpRenderer"> <ae:parameter
name="assigns"> <ae:parameter name="routing">ro</ae:parameter> <ae:parameter
name="request">rq</ae:parameter> <ae:parameter
name="controller">ct</ae:parameter> <ae:parameter name="user">us</ae:parameter>
<ae:parameter name="translation_manager">tm</ae:parameter> <ae:parameter
name="request_data">rd</ae:parameter> </ae:parameter> <ae:parameter
name="default_extension">.php</ae:parameter> <!-- this changes the name of the
variable with all template attributes from the default $template to $t --> <ae:parameter
name="var_name">t</ae:parameter> </renderer> </renderers> <layouts
default="standard"> <!-- standard layout with a content and a decorator layer --> <layout
name="standard"> <!-- content layer without further params. this means the standard
template is used, i.e. the one with the same name as the current view --> <layer
name="content" /> <!-- decorator layer with the HTML skeleton, navigation etc; set to a
specific template here --> <layer name="decorator"> <ae:parameter
name="directory">%core.template_dir%</ae:parameter> <ae:parameter
name="template">Master</ae:parameter> <slots> <slot name="navigation"
module="Widgets" action="Navigation" /> <slot name="header" module="Widgets"
action="Header" /> <slot name="footer" module="Widgets" action="Footer" /> <slot
name="dashboard" module="Widgets" action="Dashboard" /> <slot name="sidebar"
module="Widgets" action="Sidebar" /> </slots> </layer> </layout> <!-- another example
layout that has an intermediate wrapper layer in between content and decorator --> <!-- it
also shows how to use slots etc --> <layout name="wrapped"> <!-- content layer without
further params. this means the standard template is used, i.e. the one with the same
name as the current view --> <layer name="content" /> <layer name="wrapper"> <!-- use
```

CurrentView.wrapper.php instead of CurrentView.php as the template for this one --> <ae:parameter name="extension">.wrapper.php</ae:parameter> </layer> <!-- decorator layer with the HTML skeleton, navigation etc; set to a specific template here --> <layer name="decorator"> <ae:parameter name="directory">%core.template_dir%</ae:parameter> <ae:parameter name="template">Master</ae:parameter> <!-- an example for a slot --> <slot name="nav" module="Default" action="Widgets.Navigation" /> </layer> </layout> <!-- special layout for slots that only has a content layer to prevent the obvious infinite loop that would otherwise occur if the decorator layer has slots assigned in the layout; this is loaded automatically by ProjectBaseView::setupHtml() in case the current container is run as a slot --> <layout name="slot"> <layer name="content" /> </layout> </layouts> <ae:parameter name="http_headers"> <ae:parameter name="Content-Type">text/html; charset=UTF-8</ae:parameter> </ae:parameter> </output_type> </output_types> ...
We've already had a look at the layout definition so we'll skip that at this point. Please refer to the chapter about layouts. The parameter "exception_template" sets a specific exception template for this output type.
CAUTION: You should use a custom exception template - and if you do so double check that it's sending a correct 500 Internal Server Error status code. You don't want your error page to end up in caches or even worse on google.
The section <renderers> specifies a list of renderers:

<renderers default="php"> <renderer name="php" class="AgaviPhpRenderer"> <ae:parameter name="assigns"> <ae:parameter name="routing">ro</ae:parameter> <ae:parameter name="request">rq</ae:parameter> <ae:parameter name="controller">ct</ae:parameter> <ae:parameter name="user">us</ae:parameter> <ae:parameter name="translation_manager">tm</ae:parameter> <ae:parameter name="request_data">rd</ae:parameter> </ae:parameter> <ae:parameter name="default_extension">.php</ae:parameter> <!-- this changes the name of the variable with all template attributes from the default $template to $t --> <ae:parameter name="var_name">t</ae:parameter> </renderer> </renderers>

This specifies a single renderer named "php" that uses the default `AgaviPhpRenderer`. This renderer is the default renderer used. The <assigns> tag controls which internal objects are available in the template and their respective variable name. The parameters named "default_extension" sets the expected template extension to ".php". The "var_name" parameter controls the name of the template variable.

Then there's the last section named "http_headers":

<ae:parameter name="http_headers"> <ae:parameter name="Content-Type">text/html; charset=UTF-8</ae:parameter> </ae:parameter>

This section contains a list of default HTTP headers to send. In most cases it's reasonable to at least list a default content-type header here.

Rendering for a specific output type

The framework calls a method named execute<OutputTypeName> on the view to render. So for an HTML output type the method "executeHtml()" is called on a view, for a JSON output type "executeJson()" etc. The method can either return a valid response content or set up layers to render or even both. Valid response content may be

- A string.
- An open file pointer. Agavi will stream the contents of that file to the user with the most efficient method available.

Using a specific output type

The output type to use can either be set on the container using the `setOutputType` method, by passing an output type name to `createSlotContainer()` when creating a slot, by passing the output type to `createForwardContainer()` when doing an internal forward or by the routing.

The any route can use the "output_type" attribute to set the active output type. A standard example is setting output types based on the file extension using a non-stopping route:

```
<route name="ot_xml" pattern=".xml$" cut="true" output_type="xml" />
```

This route would match any url ending in ".xml", strip the extension, set the output type to "xml" and then continue processing. The output type must be configured to to use this facility.

Exception Templates

Exception templates are rendered when an unhandled exception occurs during the request processing. Exception templates can be configured in `app/config/settings.xml` and overwritten in `app/config/output_types.xml`. The configuration in settings.xml aims at setting up a reasonable default and is used if no configuration can be loaded from output_types.xml. This can happen in a variety of cases:

- The error occurs before the configuration from output_types.xml is loaded, i.e. during the framework bootstrap.
- There is no specific configuration for the active output type in output_type.xml

Configuration in settings.xml

The configration in settings.xml uses a special <exception_templates> block. Each entry can contain a context name and must specify a valid path to an exception template. The entry with an empty name is used as default for all contexts.

<exception_templates>
<exception_template>%core.agavi_dir%/exception/templates/shiny.php</exception_template>
<exception_template
context="console">%core.agavi_dir%/exception/templates/plaintext.php</exception_template>
</exception_templates>
CAUTION: While the shiny exception template provides a beautiful stacktrace and is very useful for debugging we strictly recommend against using it in production. Design your own error template an log the error instead of displaying it to the user.

Configuration in output_types.xml

The <output_type> tag can have an optional exception_template attribute. It contains the path to the template to use for this specific output type. The value supersedes any value from settings.xml.

Generating an RSS Feed

We'll be using Zend_Feed to generate the RSS feed, so we need the relevant parts of the Zend Framework installed. Just drop the pieces you need in `libs/Zend` and have an include path point at `libs/`. This is best done by adding a line to `app/config.php`:

```
ini_set('include_path', dirname(dirname(__FILE__)) . DIRECTORY_SEPARATOR
. 'libs' . PATH_SEPARATOR . ini_get('include_path'));
```

First we need to define a suitable output type for the RSS feed. We don't need a renderer or a layout as we'll use a generator for the feed. We should provide a proper content-type header as well - we'll be using "text/xml". So our output type definition looks like this:

<output_type name="rss"> <ae:parameter name="http_headers"> <ae:parameter name="Content-Type">text/xml; charset=UTF-8</ae:parameter> </ae:parameter> </output_type>

Now we need to add a route that set's the output type. We want any url that ends in ".rss" to be mapped to the "rss" output type. So we place this route at the top of our routing.xml

```
<route name="ot_rss" pattern=".rss$" cut="true" stop="false"
output_type="rss" />
```

All that's left to do now is implement the appropriate executeRss() method on the views where we'd like to provide an rss feed. Let's start with the index page. All we need to do is provide a good implementation for `Posts_IndexSuccessView::executeRss()`:

```
public function executeRss(AgaviRequestDataHolder $rd)
{
        $ro = $this->getContext()->getRouting();
        $entries = array();

        foreach($this->getAttribute('posts') as $p)
        {
                $entries[] = array(
                        'title'        => $p->getTitle(), //required
                        'lastUpdate'  => time(), // optional
                        'link'         => $ro->gen('posts.post.show',
array('post' => $p), array('relative' => false)), //required
                        'charset'      => 'UTF-8', // required
                        'published'    => time(), //optional
                        'description' => $p->getContent(), //optional
                        'author'       => $p->getAuthorName(), //optional
                        'language'     => 'en', // optional
                );
        }

        $data = array(
                'title'        => 'Latest Posts', //required
                'lastUpdate'  => null, // optional
                'link'         => $ro->gen(null, array(), array('relative'
=> false)), //required
                'charset'      => 'UTF-8', // required
                'published'    => time(), //optional
                'description' => 'The latest posts from the agavi
tutorial blog application', //optional
                'author'       => null, //optional
                'language'     => 'en', // optional
                'entries'      => $entries
        );


        require 'Zend/Feed.php';
        $feed = Zend_Feed::importArray($data, 'rss');

        return $feed->saveXml();
}
```

I won't go into the details of Zend_Feed but all we did was construct the proper data array and have the library construct the feed to us. The `Zend_Feed_Element::saveXml()` returns the generated xml which we use as return value for the `executeRss()` method. As there is no renderer involved the return value gets sent straight to the client. We didn't use the `Zend_Feed_Abstract::send()` method as sending headers and content directly would cause major problems with the framework, especially with the caching mechanism.

Note how we did not use the post's posted date in the feed - Zend_Feed expects a timestamp at this place and we're storing a string. We'll get to that when we're handling the calendar library and the translation subsystem.