Scala and the web

Using Lift to write a simple CMS
What are Scala and Lift?

Monday, 27 April 2009

A new language that has a number of interesting features. I want to talk about some of these features, and to put them in context, I'll be using a very small example application.

My current day job is web programming, so it was natural for me to choose a web framework as a route into using Scala; there are other ones (Slinky), but this is the one that is generating the most buzz at the moment, and it makes
Scala

Monday, 27 April 2009

* Language
Scala is a language designed by Martin Odersky: amongst other things, he wrote the Generic Java compiler which became Sun's standard Java compiler starting with v1.3 (Generics were in there, but disabled, until v1.5). It's been around for a wee while: version 2.0 emerged in 2006, and regular updates have been made since then, refining the language and the compiler. version 2.8 is on the horizon now with a number of improvements.

* Strongly Typed
Diminishes your test burden
Better chance of getting it right first time
Better tool support

* Dynamic
Concise and expressive: absence of boilerplate: “Tasteful typing”
Interactive interpreter
Type inferencing

* Object Oriented
Classes
Multiple Inheritance via mix-in traits
Virtual functions
Overloaded functions
Can build your own value types that implement operators: just like the ints
Programming with mutable state: collections, vars

* Functional
Functions are first class objects
Pattern matching
Tail recursion (albeit limited)
Currying
Programming with immutable state: collections, vals
“For” Comprehensions

* Compiles to JVM byte code:
interoperate with other libraries (in both directions)
uses JVM debug format, and thus works with existing debuggers out-of-the-box

* MSIL
Support lags behind the JVM implementation: for the adventurous only
Scala

- A new language (v1.0 in 2003)

Monday, 27 April 2009

* Language
Scala is a language designed by Martin Odersky: amongst other things, he wrote the Generic Java compiler which became Sun’s standard Java compiler starting with v1.3 (Generics were in there, but disabled, until v1.5). It’s been around for a wee while: version 2.0 emerged in 2006, and regular updates have been made since then, refining the language and the compiler. version 2.8 is on the horizon now with a number of improvements.

* Strongly Typed
Diminishes your test burden
Better chance of getting it right first time
Better tool support

* Dynamic
Concise and expressive: absence of boilerplate: “Tasteful typing”
Interactive interpreter
Type inferencing

* Object Oriented
Classes
Multiple Inheritance via mix-in traits
Virtual functions
Overloaded functions
Can build your own value types that implement operators: just like the ints
Programming with mutable state: collections, vars

* Functional
Functions are first class objects
Pattern matching
Tail recursion (albeit limited)
Currying
Programming with immutable state: collections, vals
“For” Comprehensions

* Compiles to JVM byte code:
Interoperate with other libraries (in both directions)
Uses JVM debug format, and thus works with existing debuggers out-of-the-box

* MSIL
Support lags behind the JVM implementation: for the adventurous only
Scala

- A new language (v1.0 in 2003)
- Strongly typed

Monday, 27 April 2009

* Language
Scala is a language designed by Martin Odersky: amongst other things, he wrote the Generic Java compiler which became Sun’s standard Java compiler starting with v1.3 (Generics were in there, but disabled, until v1.5). It’s been around for a wee while: version 2.0 emerged in 2006, and regular updates have been made since then, refining the language and the compiler. version 2.8 is on the horizon now with a number of improvements.

* Strongly Typed
Diminishes your test burden
Better chance of getting it right first time
Better tool support

* Dynamic
Concise and expressive: absence of boilerplate: “Tasteful typing”
Interactive interpreter
Type inferencing

* Object Oriented
Classes
Multiple Inheritance via mix-in traits
Virtual functions
Overloaded functions
Can build your own value types that implement operators: just like the ints
Programming with mutable state: collections, vars

* Functional
Functions are first class objects
Pattern matching
Tail recursion (albeit limited)
Currying
Programming with immutable state: collections, vals
“For” Comprehensions

* Compiles to JVM byte code:
interoperate with other libraries (in both directions)
uses JVM debug format, and thus works with existing debuggers out-of-the-box

* MSIL
Support lags behind the JVM implementation: for the adventurous only
Scala

- A new language (v1.0 in 2003)
- Strongly typed
- …but feels dynamic

Monday, 27 April 2009
* Language
Scala is a language designed by Martin Odersky: amongst other things, he wrote the Generic Java compiler which became Sun's standard Java compiler starting with v1.3 (Generics were in there, but disabled, until v1.5). It's been around for a wee while: version 2.0 emerged in 2006, and regular updates have been made since then, refining the language and the compiler. version 2.8 is on the horizon now with a number of improvements.

* Strongly Typed
Diminishes your test burden
Better chance of getting it right first time
Better tool support

* Dynamic
Concise and expressive: absence of boilerplate: "Tasteful typing"
Interactive interpreter
Type inferencing

* Object Oriented
Classes
Multiple Inheritance via mix-in traits
Virtual functions
Overloaded functions
Can build your own value types that implement operators: just like the ints
Programming with mutable state: collections, vars

* Functional
Functions are first class objects
Pattern matching
Tail recursion (albeit limited)
Currying
Programming with immutable state: collections, vals
"For" Comprehensions

* Compiles to JVM byte code:
interoperate with other libraries (in both directions)
uses JVM debug format, and thus works with existing debuggers out-of-the-box

* MSIL
Support lags behind the JVM implementation: for the adventurous only
Scala

- A new language (v1.0 in 2003)
- Strongly typed
- …but feels dynamic
- Object-oriented

Monday, 27 April 2009

* Language
Scala is a language designed by Martin Odersky: amongst other things, he wrote the Generic Java compiler which became Sun’s standard Java compiler starting with v1.3 (Generics were in there, but disabled, until v1.5). It’s been around for a wee while: version 2.0 emerged in 2006, and regular updates have been made since then, refining the language and the compiler. version 2.8 is on the horizon now with a number of improvements.

* Strongly Typed
Diminishes your test burden
Better chance of getting it right first time
Better tool support

* Dynamic
Concise and expressive: absence of boilerplate: “Tasteful typing”
Interactive interpreter
Type inferencing

* Object Oriented
Classes
Multiple Inheritance via mix-in traits
Virtual functions
Overloaded functions
Can build your own value types that implement operators: just like the ints
Programming with mutable state: collections, vars

* Functional
Functions are first class objects
Pattern matching
Tail recursion (albeit limited)
Currying
Programming with immutable state: collections, vals
“For” Comprehensions

* Compiles to JVM byte code:
interoperate with other libraries (in both directions)
uses JVM debug format, and thus works with existing debuggers out-of-the-box

* MSIL
Support lags behind the JVM implementation: for the adventurous only
Scala

• A new language (v1.0 in 2003)
• Strongly typed
• …but feels dynamic
• Object-oriented
• …and also functional
Scala

- A new language (v1.0 in 2003)
- Strongly typed
  - …but feels dynamic
- Object-oriented
  - …and also functional
- Runs on the JVM

---

Monday, 27 April 2009

* Language
  Scala is a language designed by Martin Odersky: amongst other things, he wrote the Generic Java compiler which became Sun’s standard Java compiler starting with v1.3 (Generics were in there, but disabled, until v1.5). It’s been around for a wee while: version 2.0 emerged in 2006, and regular updates have been made since then, refining the language and the compiler. version 2.8 is on the horizon now with a number of improvements.

* Strongly Typed
  Diminishes your test burden
  Better chance of getting it right first time
  Better tool support

* Dynamic
  Concise and expressive: absence of boilerplate: “Tasteful typing”
  Interactive interpreter
  Type inferencing

* Object Oriented
  Classes
  Multiple Inheritance via mix-in traits
  Virtual functions
  Overloaded functions
  Can build your own value types that implement operators: just like the ints
  Programming with mutable state: collections, vars

* Functional
  Functions are first class objects
  Pattern matching
  Tail recursion (albeit limited)
  Currying
  Programming with immutable state: collections, vals
  “For” Comprehensions

* Compiles to JVM byte code:
  Interoperate with other libraries (in both directions)
  Uses JVM debug format, and thus works with existing debuggers out-of-the-box

* MSIL
  Support lags behind the JVM implementation: for the adventurous only
Scala

- A new language (v1.0 in 2003)
- Strongly typed
  - …but feels dynamic
- Object-oriented
  - …and also functional
- Runs on the JVM
  - …and on .NET

Monday, 27 April 2009

* Language
Scala is a language designed by Martin Odersky: amongst other things, he wrote the Generic Java compiler which became Sun’s standard Java compiler starting with v1.3 (Generics were in there, but disabled, until v1.5). It’s been around for a wee while: version 2.0 emerged in 2006, and regular updates have been made since then, refining the language and the compiler. version 2.8 is on the horizon now with a number of improvements.

* Strongly Typed
   - Diminishes your test burden
   - Better chance of getting it right first time
   - Better tool support

* Dynamic
   - Concise and expressive: absence of boilerplate: “Tasteful typing”
   - Interactive interpreter
   - Type inferencing

* Object Oriented
   - Classes
   - Multiple Inheritance via mix-in traits
   - Virtual functions
   - Overloaded functions
   - Can build your own value types that implement operators: just like the ints
   - Programming with mutable state: collections, vars

* Functional
   - Functions are first class objects
   - Pattern matching
   - Tail recursion (albeit limited)
   - Currying
   - Programming with immutable state: collections, vals
   - “For” Comprehensions

* Compiles to JVM byte code:
   - Interoperate with other libraries (in both directions)
   - Uses JVM debug format, and thus works with existing debuggers out-of-the-box

* MSIL
   - Support lags behind the JVM implementation: for the adventurous only
“Lift borrows from the best of existing frameworks, providing Seaside's highly granular sessions and security, Rails' fast flash-to-bang, Django's "more than just CRUD is included" and Wicket's designer-friendly templating style”

liftweb.net
Monday, 27 April 2009

* AJAX polling
When used with Jetty: it will also work with the suspend/resume behaviour in v3.0 of the servlet specification

* ...and more
modules for a number of things, all cleanly separated out into packages: you don't pay for what you don't use.
recently hit 1.0, so it's suddenly stable
Lift

- COMET applications using the Scala Actors library

Monday, 27 April 2009

* AJAX polling
When used with Jetty: it will also work with the suspend/resume behaviour in v3.0 of the servlet specification

* ...and more
modules for a number of things, all cleanly separated out into packages: you don't pay for what you don't use.
recently hit 1.0, so it's suddenly stable
**Lift**

- COMET applications using the Scala Actors library
- AJAX polling using Jetty's continuations

Monday, 27 April 2009

* AJAX polling
When used with Jetty: it will also work with the suspend/resume behaviour in v3.0 of the servlet specification

* ...and more
modules for a number of things, all cleanly separated out into packages: you don't pay for what you don't use.
recently hit 1.0, so it's suddenly stable
Lift

- COMET applications using the Scala Actors library
- AJAX polling using Jetty's continuations
- On-the-fly javascript construction using JsCmd types

* AJAX polling
When used with Jetty: it will also work with the suspend/resume behaviour in v3.0 of the servlet specification

* ...and more
modules for a number of things, all cleanly separated out into packages: you don't pay for what you don't use.
recently hit 1.0, so it's suddenly stable
Lift

- COMET applications using the Scala Actors library
- AJAX polling using Jetty's continuations
- On-the-fly javascript construction using JsCmd types
- JQuery & blueprint.css out-of-the-box

Monday, 27 April 2009

* AJAX polling
When used with Jetty: it will also work with the suspend/resume behaviour in v3.0 of the servlet specification

* ...and more
modules for a number of things, all cleanly separated out into packages: you don't pay for what you don't use.
recently hit 1.0, so it's suddenly stable
Lift

- COMET applications using the Scala Actors library
- AJAX polling using Jetty's continuations
- On-the-fly javascript construction using JsCmd types
- JQuery & blueprint.css out-of-the-box
- …and more

Monday, 27 April 2009

* AJAX polling
When used with Jetty: it will also work with the suspend/resume behaviour in v3.0 of the servlet specification

* …and more
modules for a number of things, all cleanly separated out into packages: you don't pay for what you don't use.
recently hit 1.0, so it's suddenly stable
Lift architecture

* Servlet API
  Can be hosted by any servlet container: if you use Jetty you get extra stuff (continuations)

* Dispatch via partial functions
  HTTP requests are routed to handlers using partial functions

* “View-first model”
  This is to contrast it with Rails’ “controller-first” model

* Multiple controllers
  A web page might have several pieces of functionality on it, and thus multiple controllers.

* Snippets and bind points
  A snippet declares bind points, which the snippet controller can bind data and controls to
Lift architecture

- Servlet API

  * Servlet API
  Can be hosted by any servlet container: if you use Jetty you get extra stuff (continuations)

  * Dispatch via partial functions
  HTTP requests are routed to handlers using partial functions

  * "View-first model"
  This is to contrast it with Rails' "controller-first" model

  * Multiple controllers
  A web page might have several pieces of functionality on it, and thus multiple controllers.

  * Snippets and bind points
  A snippet declares bind points, which the snippet controller can bind data and controls to
Lift architecture

- Servlet API
- Dispatch via partial functions

Monday, 27 April 2009
David Pollak

* Servlet API
  Can be hosted by any servlet container: if you use Jetty you get extra stuff (continuations)

* Dispatch via partial functions
  HTTP requests are routed to handlers using partial functions

* “View-first model”
  This is to contrast it with Rails' "controller-first" model

* Multiple controllers
  A web page might have several pieces of functionality on it, and thus multiple controllers.

* Snippets and bind points
  A snippet declares bind points, which the snippet controller can bind data and controls to
Lift architecture

- Servlet API
- Dispatch via partial functions
- "View-first" model

* Servlet API
  Can be hosted by any servlet container: if you use Jetty you get extra stuff (continuations)

* Dispatch via partial functions
  HTTP requests are routed to handlers using partial functions

* “View-first model”
  This is to contrast it with Rails' "controller-first" model

* Multiple controllers
  A web page might have several pieces of functionality on it, and thus multiple controllers.

* Snippets and bind points
  A snippet declares bind points, which the snippet controller can bind data and controls to
Lift architecture

- Servlet API
- Dispatch via partial functions
- "View-first" model
  - Multiple controllers for a single view

* Servlet API
  Can be hosted by any servlet container: if you use Jetty you get extra stuff (continuations)

* Dispatch via partial functions
  HTTP requests are routed to handlers using partial functions

* "View-first model"
  This is to contrast it with Rails' "controller-first" model

* Multiple controllers
  A web page might have several pieces of functionality on it, and thus multiple controllers.

* Snippets and bind points
  A snippet declares bind points, which the snippet controller can bind data and controls to
Lift architecture

- Servlet API
- Dispatch via partial functions
- "View-first" model
  - Multiple controllers for a single view
  - No logic in view templates

Monday, 27 April 2009
David Pollak

* Servlet API
  Can be hosted by any servlet container: if you use Jetty you get extra stuff (continuations)

* Dispatch via partial functions
  HTTP requests are routed to handlers using partial functions

* "View-first model"
  This is to contrast it with Rails' "controller-first" model

* Multiple controllers
  A web page might have several pieces of functionality on it, and thus multiple controllers.

* Snippets and bind points
  A snippet declares bind points, which the snippet controller can bind data and controls to
Lift architecture

- Servlet API
- Dispatch via partial functions
- "View-first" model
  - Multiple controllers for a single view
  - No logic in view templates
  - Snippets + Bind points

Monday, 27 April 2009
David Pollak

* Servlet API
  Can be hosted by any servlet container: if you use Jetty you get extra stuff (continuations)

* Dispatch via partial functions
  HTTP requests are routed to handlers using partial functions

* “View-first model”
  This is to contrast it with Rails' "controller-first" model

* Multiple controllers
  A web page might have several pieces of functionality on it, and thus multiple controllers.

* Snippets and bind points
  A snippet declares bind points, which the snippet controller can bind data and controls to
Lift architecture

• Choice of persistence model is a separate concern

• ...although Mapper is a good starting point
cmless
• Tree of pages
• Tree of pages

• Create/edit/delete
cmless

- Tree of pages
- Create/edit/delete
- Page URL reflects its place in the hierarchy
Monday, 27 April 2009

* Sitemap
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn't the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality
* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it's not there.

** Retrieval: controlled by rewrite()

* PageInfo
This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page

* Page
stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM

** Rendering
* pages.html
The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.

* snippets()
called to create each snippets content in pages.html
Monday, 27 April 2009

* Sitemap
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn't the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality

* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it's not there.

** Retrieval: controlled by rewrite()

* PageInfo
This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page

* Page
stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM

** Rendering

* pages.html
The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.

* snippets()
called to create each snippets content in pages.html
Monday, 27 April 2009

* Sitemap
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn't the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality

* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it’s not there.

** Retrieval: controlled by rewrite()

* PageInfo
This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page

* Page
stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM

** Rendering

* pages.html
The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.

* snippets()
called to create each snippets content in pages.html
Monday, 27 April 2009

*Sitemap*

This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn't the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality
* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it’s not there.

** Retrieval: controlled by rewrite()
* PageInfo
This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page

* Page
stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM

** Rendering
* pages.html
The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.
* snippets()
called to create each snippets content in pages.html
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
- displays the component parts of your application in a menu hierarchy
- plugs those components into the lift dispatch rules table for you
- provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn't the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality
* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it's not there.

** Retrieval: controlled by rewrite()
* PageInfo
  This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page
* Page
  stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM

** Rendering
* pages.html
  The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.
* snippets()
  called to create each snippets content in pages.html
Monday, 27 April 2009
* Sitemap
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn't the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality

* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it's not there.

** Retrieval: controlled by rewrite()

* PageInfo
This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page

* Page
stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM

** Rendering

* pages.html
The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.

* snippets()
called to create each snippets content in pages.html
Monday, 27 April 2009

* Sitemap
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn’t the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality
* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it’s not there.

** Retrieval: controlled by rewrite()

* PageInfo
This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page

* Page
stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM

** Rendering

* pages.html
The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.

* snippets()
called to create each snippets content in pages.html
Monday, 27 April 2009

* Sitemap
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn't the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality

* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it’s not there.

** Retrieval: controlled by rewrite()

* PageInfo
This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page

* Page
stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM

** Rendering
* pages.html
The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.

* snippets()
called to create each snippets content in pages.html
* Sitemap
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn’t the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality

* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it’s not there.

** Retrieval: controlled by rewrite()

* PageInfo
This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page

* Page
stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM

** Rendering

* pages.html
The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.

* snippets()
called to create each snippets content in pages.html
Monday, 27 April 2009

* Sitemap
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (demo.liftweb.net <http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn’t the path that most lift tutorials will take you down!

* Loc[T] is the trait you extend to implement this functionality
* There are two phases to the process, handled separately by two partial functions in PageLoc: mapping the URL into a possible data item, and rendering the data or displaying an error if it’s not there.

** Retrieval: controlled by rewrite()
* PageInfo
This is a type we provide to store the data we want to render later; in this case, PageInfo associates a path relative to our component with a Page

* Page
stores the title and content of a page: persisted to an SQL DB using lift’s Mapper ORM

** Rendering
* pages.html
The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.

* snippets()
called to create each snippets content in pages.html
This is a lift component that provides a simple way to plug-in different components of a web-app into a single navigational structure. As an example, show the demo lift application (<http://demo.liftweb.net/>)

It:
* displays the component parts of your application in a menu hierarchy
* plugs those components into the lift dispatch rules table for you
* provides a means to translate an HTTP request into a typed parameter that can subsequently be used in the display code

Building an entire app through a single sitemap handler is probably a bit daft, but it let me keep a lot of stuff in what place, which was handy for exposition purposes: this isn't the path that most lift tutorials will take you down!

* `Loc[T]` is the trait you extend to implement this functionality
* There are two phases to the process, handled separately by two partial functions in `PageLoc`: mapping the URL into a possible data item, and rendering the data or displaying an error if it's not there.
  ** Retrieval: controlled by `rewrite()`
* `PagelInfo`
  This is a type we provide to store the data we want to render later; in this case, `PagelInfo` associates a path relative to our component with a `Page`
* `Page`
  stores the title and content of a page: persisted to an SQL DB using lift's Mapper ORM
  ** Rendering
  * `pages.html`
  The template used to display a page; contains snippets for viewing the contents, create new pages, edit and delete the current one.
  * `snippets()`
  called to create each snippets content in `pages.html`
case class PageInfo(path: List[String]) {
  private def url(path: List[String]) = path.map(urlEncode _).mkString("/")

  val urlpath = url(path)

  lazy val page = {
    val p = PageInfo.findPage(path)
    p match {
      case Failure(msg, _, _) => S.error(urlpath + ": " + msg)
      case Empty => S.error(urlpath + ": Page not found")
      case _ => ()
    }
    p
  }

  def create(title: String): Box[Page] = 
    page.flatMap(PageInfo.createPage(_, title))
}
A rubbish example
class Litter(name: String, grams: Int) {
private val _name = name
private val _grams = grams

  def name(): String = { _name }
  def grams(): Int = { _grams }

  override def toString(): String = _name + "(" + _grams + ")"
}

object Main {
  def main(args: Array[String]): Unit = {
    val crisps = new Litter("crisp wrapper", 5)
    println("name "+ crisps.name)
    println("weight "+ crisps.grams)
    println(crisps)
  }
}

Monday, 27 April 2009
* A simple class without any behaviour
* Things to note:
  * def declares a function
  * val declares a variable
  * the class declaration is also the initialiser (you can declare other initialisers)
* Litter derives from scala.lang.Any, which contains toString, hashCode and equals
* We need to _explicitly_ override toString: scala won’t let you override otherwise
* You don’t have to provide empty parens for method calls that don’t have any parameters
* object: scala doesn’t have static members, instead it has singleton objects. Thus Main#main above is equivalent to a static method.

* We seem to be writing a lot for something that doesn’t do much
* First, all members occupy the same namespace, and since we can omit parens on getter-type methods, we can replace the calls with the vals themselves: this is fine, since if we need to run code, we can always go back to def
class Litter(val name: String, val grams: Int) {
    override def toString() = name + "(" + grams + ")"
}

* Here's something with some state
* vars can be reassigned; vals cannot
* Things to note:
* Calling a base class initialiser is done in the extends clause
* pickUp's return type is Unit (as is main's); this can be shortened (show them)
* just as with vals, we can have public vars: let's allow maxLitterGrams to be changeable
* whoops: constraint violation! vars automatically generate a getter and a setter: we can override these.
## Classes

```scala
class Litter(val name: String, val grams: Int) {
    override def toString() = name + "(" + grams + "g")"
}

class Womble(val name: String, val maxLitterGrams: Int) {
    private var _litter: List[Litter] = Nil
    private var _litterGrams: Int = 0

    class TooMuchLitter extends RuntimeException("Too heavy for me!")

    def pickUp(litter: Litter): Unit = {
        if ((_litterGrams + litter.grams) > maxLitterGrams)
            throw new TooMuchLitter
        _litter = litter :: _litter
        _litterGrams += litter.grams
    }
}
```

* Here's something with some state
* vars can be reassigned; vals cannot
* Things to note:
* Calling a base class initialiser is done in the extends clause
* pickUp's return type is Unit (as is main's); this can be shortened (show them)
* just as with vals, we can have public vars: let’s allow maxLitterGrams to be changeable
* whoops: constraint violation! vars automatically generate a getter and a setter: we can override these.
Classes

class Womble(val name: String, private var _maxLitterGrams: Int) {
  private var _litter: List[Litter] = Nil
  private var _litterGrams: Int = 0

  class TooMuchLitter extends RuntimeException("Too heavy for me!")

  def pickUp(litter: Litter) = {
    if ((_litterGrams + litter.grams) > maxLitterGrams)
      throw new TooMuchLitter
    _litter = litter :: _litter
    _litterGrams += litter.grams
  }

  def maxLitterGrams = _maxLitterGrams
  def maxLitterGrams_=(grams: Int) =
    if (grams > _maxLitterGrams)
      _maxLitterGrams = grams

  override def toString =
    "Womble(" + name + ", " + maxLitterGrams + 
    "g) is carrying " + _litterGrams + "g: " + 
    _litter.mkString("","")
  }
}

Monday, 27 April 2009

* Here's something with some state
* vars can be reassigned; vals cannot
* Things to note:
* pickUp's return type is Unit (as is main's); this can be shortened (show them)
* just as with vals, we can have public vars: let's allow maxLitterGrams to be changeable
* whoops: constraint violation! vars automatically generate a getter and a setter: we can override these
Factory methods

```kotlin
class Litter(val name: String, val grams: Int)
val crisps = new Litter("crisp wrapper", 5)
```

* We want to restrict litter creation to a factory method: only that can create litter
Factory methods

```scala
class Litter(val name: String, val grams: Int)
val crisps = new Litter("crisp wrapper", 5)

class Litter private (val name: String, val grams: Int)
object Litter {
  def makeLitter(name: String, grams: Int) = new Litter(name, grams)
}
val crisps = Litter.makeLitter("crisp wrapper", 5)
```

* We want to restrict litter creation to a factory method: only that can create litter.
Factory methods

```scala
class Litter(val name: String, val grams: Int)
val crisps = new Litter("crisp wrapper", 5)

class Litter private (val name: String, val grams: Int)
object Litter {
  def makeLitter(name: String, grams: Int) = new Litter(name, grams)
}
val crisps = Litter.makeLitter("crisp wrapper", 5)

class Litter private (val name: String, val grams: Int)
object Litter {
  def apply(name: String, grams: Int) = new Litter(name, grams)
}
val crisps = Litter("crisp wrapper", 5)
```

* We want to restrict litter creation to a factory method: only that can create litter
apply(…) 

- apply is special
- C++’s operator()
- Python’s __call__
Classes

• val
• var
• def
• Initialisers
• Factory methods

Monday, 27 April 2009

We’ve just looked at val, var and def and initialisers. Most of this was just syntactic differences to other languages, although these differences allow a conciseness of expression without sacrificing the ability to transparently make changes later on.
Case Classes

class Litter private (val name: String, val grams: Int)
object Litter {
    def apply(name: String, grams: Int) = new Litter(name, grams)
}
val crisps = Litter("crisp wrapper", 5)

* We can make the code for Litter even more concise
* In fact, case is (more or less) shorthand for this:
* Things to note:
* Sensible overrides of toString, hashCode and equals (deep comparison)
* unapply: what's that? What's an Option?
* It's a thing: the important thing to note is, like many of the things we've already seen, you can always implement it the long way round. unapply is an extractor, and it enables a class to be used for Pattern Matching.
Case Classes

class Litter private (val name: String, val grams: Int)
object Litter {
  def apply(name: String, grams: Int) = new Litter(name, grams)
}
val crisps = Litter("crisp wrapper", 5)

case class Litter(name: String, grams: Int)

* We can make the code for Litter even more concise
* In fact, case is (more or less) shorthand for this:
* Things to note:
* Sensible overrides of toString, hashCode and equals (deep comparison)
* unapply: what's that? What's an Option?
* It’s a thing: the important thing to note is, like many of the things we’ve already seen, you can always implement it the long way round. unapply is an extractor, and it enables a class to be used for Pattern Matching.
Monday, 27 April 2009

* We can make the code for Litter even more concise
* In fact, case is (more or less) shorthand for this:
* Things to note:
* Sensible overrides of toString, hashCode and equals (deep comparison)
* unapply: what's that? What's an Option?
* It’s a thing: the important thing to note is, like many of the things we’ve already seen, you can always implement it the long way round. unapply is an extractor, and it enables a class to be used for Pattern Matching.
### Case Classes

```scala
class Litter private (val name: String, val grams: Int)
object Litter {
  def apply(name: String, grams: Int) = new Litter(name, grams)
}
val crisps = Litter("crisp wrapper", 5)
```

class Litter(val name: String, val grams: Int) {
  override def toString = ...
  override def hashCode = ...
  override def equals(that: Any) = ...
}
object Litter {
  def apply(name: String, grams: Int) = new Litter(name, grams)
  def unapply(litter: Litter): Option[(String, Int)] =
    Some((litter.name, litter.grams))
}

* We can make the code for Litter even more concise
* In fact, case is (more or less) shorthand for this:
* Things to note:
  * Sensible overrides of toString, hashCode and equals (deep comparison)
  * unapply: what’s that? What’s an Option?
* It’s a thing: the important thing to note is, like many of the things we’ve already seen, you can always implement it the long way round. unapply is an extractor, and it enables a class to be used for Pattern Matching.
Pattern matching

• Superficially similar to Java/C++ switch statements
• Not just primitive types
Pattern matching

```scala
val i = 5
val s = i match {
  case 5 => "half"
  case 10 => "full"
  case something => something
}
```

* Things to note:
* A lower-case variable name in a case statement binds the variable to the value
* Case classes give us this power
* Guard expressions
Pattern matching

```scala
val i = 5
val s = i match {
  case 5 => "half"
  case 10 => "full"
  case something => something
}

val crisps = Litter("crisp wrapper", 5)
crisps match {
  case Litter(_, 5) => println("light litter")
  case Litter(_, 10) => println("heavier litter")
  case Litter(what, _) => println("unclassified: "+ what)
}
```

* Things to note:
  * A lower-case variable name in a case statement binds the variable to the value
  * Case classes give us this power
  * Guard expressions
Pattern matching

```scala
val i = 5
val s = i match {
  case 5 => "half"
  case 10 => "full"
  case something => something
}

val crisps = Litter("crisp wrapper", 5)
crisps match {
  case Litter(_, 5) => println("light litter")
  case Litter(_, 10) => println("heavier litter")
  case Litter(what, _) => println("unclassified: " + what)
}

val crisps = Litter("crisp wrapper", 7)
crisps match {
  case Litter(_, weight) if weight <= 5 => println("light litter")
  case Litter(_, weight) if weight <= 10 => println("heavier litter")
  case Litter(what, _) => println("unclassified: " + what)
}
```

Monday, 27 April 2009

* Things to note:
* A lower-case variable name in a case statement binds the variable to the value
* Case classes give us this power
* Guard expressions
Pattern matching

```scala
class Womble(val name: String, val maxLitterGrams: Int) {
  private var _litter: List[Litter] = Nil

  class TooMuchLitter extends RuntimeException("Too heavy for me!")

  def pickUp(litter: Litter): Unit = {
    if ((litterGrams + litter.grams) > maxLitterGrams)
      throw new TooMuchLitter
    _litter = litter :: _litter
  }

  private def litterGrams(litter: List[Litter]): Int = litter match {
    case l :: ls => l.grams + litterGrams(ls)
    case Nil => 0
  }

  def litterGrams: Int = litterGrams(_litter)
}
```

Monday, 27 April 2009

* Most of scala's core classes implement pattern matching, including list
* pickUp: Point out the list cons operator
* Cons can also be used in pattern matching: the details of the mechanism are too much to go into here
* For some reason, we've decided to trade-off time for space, and changed litterGrams to be a method that iterates over the litter list.
* Things to note:
  * Pattern matching
  * Tail recursion
Tail recursion

- Scala supports tail recursion
- Only works with calls to the calling method
- 2.8.0 will bring the @tailrec annotation
Parameterized Types

- `class` `Container[T](val t: T)
- Still limited by erasure
- But not as limited as Java
  - Upper and lower bounds `[T <: Womble]
  - View bounds `[T <% Womble]
- No wildcards at point of use i.e. declaration-site variance

* Lose run-time type information through erasure
* Upper and lower bounds are sort-of equivalent to the Java `<T extends Womble>` syntax
Erasure

object Main {
    case class Container[T](val t: T)

    def contents(c: Any) = c match {
        case c: Container[Int] => println(c.t)
        case c: Container[String] => println(c.t)
    }

    def main(args: Array[String]) {
        val ic = new Container(1)
        val sc = new Container("foo")
        contents(ic)
        contents(sc)
    }
}

* Erasure: this is an example of using a typed pattern: it results in compiler warnings (underlined):
  * non variable type-argument Int in type pattern is unchecked since it is eliminated by erasure
Erasure

```scala
object Main {
  case class Container[T](val t: T)

  def contents(c: Any) = c match {
    case c: Container[Int] => println(c.t)
    case c: Container[String] => println(c.t)
  }

  def main(args: Array[String]) {
    val ic = new Container(1)
    val sc = new Container("foo")
    contents(ic)
    contents(sc)
  }
}
```

TestApp.scala:5: warning: non variable type-argument Int in type pattern is unchecked since it is eliminated by erasure
  case c: Container[Int] => println(c.t)
^ TestApp.scala:6: warning: non variable type-argument String in type pattern is unchecked since it is eliminated by erasure
  case c: Container[String] => println(c.t)
    ^ TestApp.scala:6: error: unreachable code
  case c: Container[String] => println(c.t)
    ^
```

* Erasure: this is an example of using a typed pattern: it results in compiler warnings (underlined):
  * non variable type-argument Int in type pattern is unchecked since it is eliminated by erasure
case class Litter(val name: String, val grams: Int)
extends Ordered[Litter] {
  def compare(that: Litter) = grams - that.grams
}

def max[T <: Ordered[T]](elements: List[T]): T =
  elements match {
    case List() =>
      throw new IllegalArgumentException("Empty!")
    case List(x) => x
    case x :: xs =>
      val m = max(xs)
      if (x > m) x else m
  }
null is a poor choice for a result value
* it’s not obvious when it’s expected for something to return null and when it isn’t
* Get it wrong and you’ll only detect it when you get a NullPointerException, and that could happen anywhere
* Option[T] makes it obvious
Option[T]

- Express absence of a value

- null is a poor choice for a result value
- it’s not obvious when it’s expected for something to return null and when it isn’t
- Get it wrong and you’ll only detect it when you get a NullPointerException, and that could happen anywhere
- Option[T] makes it obvious
Option[T]

- Express absence of a value
- …without using null

* null is a poor choice for a result value
* it’s not obvious when it’s expected for something to return null and when it isn’t
* Get it wrong and you’ll only detect it when you get a NullPointerException, and that could happen anywhere
* Option[T] makes it obvious
Option[T]

- Express absence of a value
- ...without using null

```scala
def getName(id: Int): String
```
Option[T]

- Express absence of a value
- ...without using null

```scala
def getName(id: Int): String

def getName(id: Int): Option[String]
```

* null is a poor choice for a result value
* it’s not obvious when it’s expected for something to return null and when it isn’t
* Get it wrong and you’ll only detect it when you get a NullPointerException, and that could happen anywhere
* Option[T] makes it obvious
Box[T]

* Box is lift's take on Option
* It adds Failure to the possible values
* Failure can contain a chained failure/exception
*
Box[T]

• Lift’s version of Option[T]

Monday, 27 April 2009
* Box is lift’s take on Option
* It adds Failure to the possible values
* Failure can contain a chained failure/exception
*
• Lift’s version of Option[T]
• Full(t)
Box[T]

- Lift’s version of Option[T]
- Full(t)
- Empty

Monday, 27 April 2009
* Box is lift’s take on Option
* It adds Failure to the possible values
* Failure can contain a chained failure/exception
*
• Lift’s version of Option[T]
• Full(t)
• Empty
• Failure(reason)
Option and Box

- Effectively containers with a max. size of 1
- Implement conventional container methods:
  - map(), flatMap() and filter()
  - None/Empty/Failure values ripple up through these methods
- getOrElse() allows getting at values safely
- (They’re monads)
Option and Box
Option and Box

scala> val some = Some(1)
Option and Box

scala> val some = Some(1)

some: Some[Int] = Some(1)
Option and Box

scala> val some = Some(1)

some: Some[Int] = Some(1)

scala> val none: Option[Int] = None
Option and Box

scala> val some = Some(1)

some: Some[Int] = Some(1)

scala> val none: Option[Int] = None

none: Option[Int] = None
Option and Box

scala> val some = Some(1)
some: Some[Int] = Some(1)

scala> val none: Option[Int] = None

none: Option[Int] = None

scala> none.map(_ + 1)
Option and Box

```scala
scala> val some = Some(1)
some: Some[Int] = Some(1)

scala> val none: Option[Int] = None
none: Option[Int] = None

scala> none.map(_ + 1)
res1: Option[Int] = None
```
Option and Box

scala> val some = Some(1)
some: Some[Int] = Some(1)

scala> val none: Option[Int] = None
none: Option[Int] = None

scala> none.map(_ + 1)
res1: Option[Int] = None

scala> none.map(_ + 1).map(_ + 2)
scala> val some = Some(1)
some: Some[Int] = Some(1)

scala> val none: Option[Int] = None
none: Option[Int] = None

scala> none.map(_ + 1)
res1: Option[Int] = None

scala> none.map(_ + 1).map(_ + 2)
res2: Option[Int] = None
Option and Box

scala> val some = Some(1)
some: Some[Int] = Some(1)

scala> val none: Option[Int] = None
none: Option[Int] = None

scala> none.map(_ + 1)
res1: Option[Int] = None

scala> none.map(_ + 1).map(_ + 2)
res2: Option[Int] = None

scala> some.map(_ + 1).map(_ + 2)
Option and Box

scala> val some = Some(1)
some: Some[Int] = Some(1)

scala> val none: Option[Int] = None
none: Option[Int] = None

scala> none.map(_ + 1)
res1: Option[Int] = None

scala> none.map(_ + 1).map(_ + 2)
res2: Option[Int] = None

scala> some.map(_ + 1).map(_ + 2)
res3: Option[Int] = Some(4)
case class PageInfo(path: List[String]) {
    private def url(path: List[String]) = path.map(urlEncode _).mkString("/")

    val urlpath = url(path)

    lazy val page = {
        val p = PageInfo.findPage(path)
        p match {
            case Failure(msg, _, _) => S.error(urlpath + "": " + msg)
            case Empty => S.error(urlpath + "": Page not found")
            case _ => ()
        }
    }

    def create(title: String): Box[Page] =
        page.flatMap(PageInfo.createPage(_, title))
    }

Monday, 27 April 2009

* Things to note:
path is a constructor param
urlpath is part of the initializer
page uses pattern matching on a Box returned by a findPage method
create: creates a child of the current page and returns it as a Box[Page]; if page is None, then
createPage won’t be called.
First-class functions

• Function literals:
  • (x: Int) => x + 1
  • (_: Int) + 1
  • val inc: (Int) => Int = _ + 1
  • val inc: Function[Int, Int] = _ + 1

* Already seen the inc _ form in Box and Option
First-class functions

• Partially applied functions:
  • def inc(x: Int) = x + 1
    inc _
  • def sum(x: Int, y: Int) = x + y
    val inc = sum(_, 1)
Closures

• Behave as you would (hopefully) expect

• Referring to an in-scope variable in a function body closes over it:

  val a = 1
  val inca = (x: Int) => x + a
  List(1, 2, 3).filter(_ == a)

• Closes over the instance not the value
Closures

• Close over the instance not the value

• Can close over vars:

```scala
var a = 1
val inca = (x: Int) => x + a
inca(1) == 2
a = 2
inca(1) == 3
```
Partial functions

- A function that may not be defined for all possible input values
- Case sequences are function literals
- val pf: Int => Int = { case 2 => 2 }
- pf(3) throws a MatchError
- val pf: PartialFunction[Int, Int] = { case 2 => 2 }
- pf.isDefinedAt(3) returns false
Getting the data
class PageLoc extends Loc[PageInfo] {
val name = "pages"
// ...
override val rewrite: LocRewrite =
  Full(NamedPF("Pages rewrite") {
    case RewriteRequest(ParsePath(head :: tail, _, _, _), _, httpRequest)
      if head == name =>
        ( RewriteResponse(ParsePath(head :: Nil, "", true, false),
          Map.empty, true),
          PageInfo(tail))
  })
// ...
}

* ParamType is the type we want to store our retrieved data in, which in our case is PageInfo
* This is a partial function: note the guard on the case: it’s defined for head == name
* This handles the mapping of an HTTP request to something concrete i.e. an instance of PageInfo
**rewrite**

```scala
class PageLoc extends Loc[PageInfo] {
  val name = "pages"
  // ...

  override val rewrite: LocRewrite =
    Full(NamedPF("Pages rewrite") {
      case RewriteRequest(ParsePath(head :: tail, _, _, _), _, httpRequest) if head == name =>
        (RewriteResponse(ParsePath(head :: Nil, ",", true, false),
          Map.empty, true),
          PageInfo(tail))
      }
    })
  // ...
}

case class ParsePath(partPath: List[String], suffix: String, absolute: Boolean, endSlash: Boolean)

case class RewriteRequest(path: ParsePath, requestType: RequestType, httpRequest: HttpServletRequest)

case class RewriteResponse(path: ParsePath, params: Map[String, String], stopRewriting: Boolean)

type LocRewrite =
  Box[PartialFunction[RewriteRequest, (RewriteResponse, ParamType)]]
```

* ParamType is the type we want to store our retrieved data in, which in our case is PageInfo
* This is a partial function: note the guard on the case: it's defined for head == name
* This handles the mapping of an HTTP request to something concrete i.e. an instance of PageInfo
Displaying our results
override def snippets: SnippetTest = {
  case ("read", Full(pageInfo)) => read(pageInfo, _)
  case ("create", Full(pageInfo)) => create(pageInfo, _)
  case ("update", Full(pageInfo)) => update(pageInfo, _)
  case ("delete", Full(pageInfo)) => delete(pageInfo, _)
}

private def read(pageInfo: PageInfo, content: NodeSeq): NodeSeq = {
  def join[A](xs: List[A], sep: A): List[A] = ...
  def children(page: Page) = ...
  def ancestors: List[Elem] = ...

  pageInfo.page match {
    case Full(page) =>
      bind("read", content,
        "title" -> Text(page.title.is),
        "ancestors" -> join(ancestors, Text(" >> ")),
        "children" -> join(children(page), Text(" :: ")),
        "content" -> Text(page.content.is))
    case _ => NodeSeq.Empty
  }
}

Monday, 27 April 2009
* Things to note:
* read()
  * nested functions
* lift#bind(snippet-name, contents, BindParam*)
def bind(namespace: String, xml: NodeSeq, 
    params: BindParam*): NodeSeq

• binds xml items in the specified namespace

• BindParam associates a snippet parameter with a replacement

• “a” -> replacement is an overloaded function on SuperArrowAssoc, which has an implicit conversion from string

Monday, 27 April 2009
mvn -o scala:console
scala> import net.liftweb.util.BindHelpers._
scala> bind("user", <user:hello>foo</user:hello>, "hello" -> <h1>bar</h1>)
bind("user", <user:hello>foo</user:hello>, "hello" -> <h1>bar</h1>)
res2: scala.xml.NodeSeq = <h1>bar</h1>
def join[A](xs: List[A], sep: A): List[A] = xs match {
  case Nil => Nil
  case x :: Nil => x :: Nil
  case x :: xs => x :: sep :: join(xs, sep)
}

def children(page: Page) =
  for (c <- page.children)
    yield <a href={ url(pageInfo.path ::: List(c.title.is)) }>
      { c.title.is }</a>

def ancestors: List[Elem] = {
  import scala.collection.mutable.ListBuffer
  val path = new ListBuffer[String]()
  val home = <a href={ url(path.toList) }>{PageInfo.home.title.is}</a>
  val rest = if (pageInfo.path.size > 1) {
    for (a <- pageInfo.path.dropRight(1)) yield {
      path += a
      <a href={ url(path.toList) }>{a}</a>
    }
  } else { Nil
  }
  home :: rest
}

Monday, 27 April 2009
* Things to note:
  * join is functional and recursive
  * children is functional
  * ancestors is imperative (I was going to make it functional, but thought it was worth leaving in as an example)
**bind & forms**

- `bind()` is also used to bind input data

    ```scala
    private object createTitle extends RequestVar(""")

    private def create(pageInfo: PageInfo, content: NodeSeq): NodeSeq =
        pageInfo.page match {
        case Full(page) =>
            bind("create", content,
                "title" -> SHtml.text(createTitle.is, createTitle(_)),
                "submit" -> SHtml.submit("Create a new page",
                    { () => doCreate(pageInfo,
                        createTitle.is) })),
            case _ => NodeSeq.Empty
        }
    ```
private object createTitle extends RequestVar("")
SHtml.text(createTitle.is, createTitle(_))

• createTitle is like a ThreadLocal, but per Request
• Registers createTitle(_) as a callback in session state
• Generates <input ... id="callback-id"/>
• POST request looks up the callback and executes it
What have I talked about?

• Scala:
  • Conciseness of classes
  • Power of pattern matching
  • Flexibility of functions
• Lift:
  • Partial functions for dispatching
  • Model: Simple Mapper ORM
  • Views: Snippets
  • Controller: bind()
What haven’t I talked about?

- Traits
- XML literals
- Duck typing using anonymous classes
- Implicits
- Co/contravariance specification at declaration point enforces LSP

- DSL-supporting features
  - operator definition
  - method call syntax doesn't require dots, and single argument method calls can be made without parens
  - ScalaTest & ScalaSpec's BDD specs

Monday, 27 April 2009

...just a small selection of language features.
There's also an increasing number of libraries: the actors library, combinator parser, scalaz, scalax
...and tools: ScalaCheck (a port of Haskell’s QuickCheck)
Scala 2.8.0

- Redesigned collection libraries (mostly backwards compatible)
- Redesigned combinator parser library
- Named & default parameters
- Continuations
- Revamped REPL, including completion
- Source compatible with old code (but not binary)
You'll never find a programming language that frees you from the burden of clarifying your ideas.

But I know what I mean!

The uncomfortable truths well.

http://xkcd.com/568/
object Lunar extends Baysick {
  def main(args:Array[String]) = {
    10 PRINT "Welcome to Baysick Lunar Lander v0.9"
    20 LET ('dist := 100)
    30 LET ('v := 1)
    40 LET ('fuel := 1000)
    50 LET ('mass := 1000)

    60 PRINT "You are drifting towards the moon."
    70 PRINT "You must decide how much fuel to burn."
    80 PRINT "To accelerate enter a positive number"
    90 PRINT "To decelerate a negative"

    100 PRINT "Distance " % 'dist % " km, " % "Velocity " % 'v % " km/s, " % "Fuel " % 'fuel
    110 INPUT 'burn
    120 IF ABS('burn) <= 'fuel THEN 150
    130 PRINT "You don't have that much fuel"
    140 GOTO 100

    150 LET ('v := 'v + 'burn * 10 / ('fuel + 'mass))
    160 LET ('fuel := 'fuel - ABS('burn))
    170 LET ('dist := 'dist - 'v)
    180 IF 'dist > 0 THEN 100
    190 PRINT "You have hit the surface"
    200 IF 'v < 3 THEN 240
    210 PRINT "Hit surface too fast (" % 'v % ") km/s"
    220 PRINT "You Crashed!"
    230 GOTO 250
    240 PRINT "Well done"

    250 END
  }
}

Monday, 27 April 2009
Find out more

- http://www.scala-lang.org
- http://www.liftweb.net
- Programming in Scala Odersky, Spoon, Venners
  Artima 2008
- Monads are Elephants James Iry

Monday, 27 April 2009