Web frameworks

Lucian Mogosanu, Mihai Maruseac

July 12, 2013

Static site generation

- Pros
  - Speed
  - Security
  - Focus on *content*
- Cons
  - Lack of functionality

Static site generators

- Jekyll, nanoc (Ruby)
- Hyde (Python)
- Templer (Perl)
- Hakyll (Haskell)

Installing Hakyll

```
$ cabal install hakyll
```

- For local installations, add `.cabal/bin` to `$PATH`

Using Hakyll

- Init and compile site

```
$ hakyll-init mycoolsite
$ cd mycoolsite
$ ghc site.hs
```

- Build `_site`

```
$ ./site build
```

Using Hakyll (2)

- Live preview and autocompile

```
$ ./site preview
```

- Clean generated site

```
$ ./site clean
```

- Others

```
$ ./site help
```

Hakyll basics: templates

- Template pages
  - Content + **fields**
  - e.g. `$body$`, `$author$`
- Fields belong to a `Context`
  - `defaultContext`: `$body$`, `$url$`, post metadata etc.
  - Lists of items: use `$for(field)$`

Hakyll basics: the `Rules` monad

- `match`: globbing on content
- `create`: create page from scratch
- `route`: generate file path
- `compile`: add compilation rule

Hakyll basics: content

- "Content is king"
- Write in any language supported by Pandoc
- Compile using `pandocCompiler`

Hakyll basics: conclusion

- Easy to configure for blog generation
  - Ideal for small to medium sites
- Can be integrated with commenting services
  - e.g. Disqus, IntenseDebate
- Doesn't scale well for large sites

Yesod

- Haskell web framework
- type-safe
- high performance
- RESTful web application

Async

- Web is async
- Haskell runtime is async
- lightweight green threads
- event-based system calls
- non-blocking code
- async is really easy

No boilerplate

- templates, routes, database connections
- remember DRY principle
- DSLs = Domain Specific Languages
- compile-time checked for bugs

Widgets

- small parts of application
- footer, header, Scroll to top button
- bits of HTML, CSS, Javascript
- some in header
- some in body
- compose them nicely
- compile unique and compressed CSS file
- reuse resources

```
mainWdg = menuWdg >> contentWdg >> footerWdg
```

Shakespeare

- [CSS] Cassius and Lucius
- [JavaScript] Julius
- [HTML] Hamlet

```
toWidgetHeader cassiusFile "button.cassius"
toWidgetHeader juliusFile "button.julius"
toWidget       hamletFile "buttonTemplate.hamlet"
```

Bootstrap/Scaffolding. First Site

```
$ yesod init
$ yesod devel
```

Important paths

- `config/routes` - map URL –> Code
- `Handler/` - code called when a URL is accessed.
- `templates/` - HTML, js and CSS templates.
- `config/models` - persistent objects (database tables).

Adding handlers

```
$ yesod add-handler

getEchoR :: String -> Handler RepHtml
getEchoR theText = defaultLayout [whamlet|<h1>#{theText}|]
```

Bulletproof

Visit `http://localhost:3000/echo/I'm`
`<script>alert(\"Bad!\");</script>`

Good practices

- `Data.Text` instead of `String`
- use templates

Mirror

```haskell
getMirrorR :: Handler RepHtml
getMirrorR = defaultLayout $(widgetFile "mirror")

postMirrorR :: Handler RepHtml
postMirrorR =  do
    postedText <- runInputPost $ ireq textField "content"
    defaultLayout $(widgetFile "posted")
```

Mirror

```
<h1> Enter your text
<form method=post action=@{MirrorR}>
    <input type=text name=content>
    <input type=submit>
```

Mirror

```
<h1>You've just posted
<p>#{postedText}#{T.reverse postedText}
<hr>
<p><a href=@{MirrorR}>Get back
```

Scaffolding vs Full Code

- scaffolding helps in fast startup
- scaffolding provides a set of good practices
- scaffolding gets in the way of learning Yesod

Other aspects

- internationalisation
- copyright & GA
- testing via `yesod test`