Radiance - A Web Application Environment

Nicolas Hafner
Shirakumo.org
Zürich, Switzerland
shinmera@tymoon.eu

ABSTRACT

Radiance[1] is a set of libraries that provide an environment for web applications. By putting its focus on running multiple web services or applications within the same environment, it has developed features that set it apart from traditional web frameworks. In particular, it shows a new approach to the way different framework features are provided, and how the routing to content is performed. These differences allow applications written against Radiance to transparently share common resources such as the HTTP server, user accounts, authentication, and so forth with each other.

CCS Concepts

•**Software and its engineering** → **Development frameworks and environments;** •**Information systems** → *Web applications;*

Keywords

Common Lisp, web framework, web development, encapsulation, interfacing

1. INTRODUCTION

As the internet evolved, websites began to evolve and became more and more dynamic. Content is no longer a set of static webpages, but rather automatically updated, or even individually composed for the specific user viewing the website. Creating such dynamic websites requires a lot of extra work, much of which is usually done in much the same way for every website. The requested content needs to be retrieved somehow, transformed as needed, and finally assembled into deliverable HTML.

In order to handle these common aspects, web frameworks have been created. These frameworks can come in all kinds of sizes, from the very minimal set of an HTML templating engine and a web server as seen in "micro-frameworks"[2], to extensive sets of utilities to handle user interaction and data of all kinds.

Typically these frameworks are constructed with the intent of helping you develop a single web application, which is then deployed and run standalone. However, this paradigm can lead to issues when multiple applications should be deployed side-by-side. For example, if the framework does not provide explicit support for a particular feature such as user accounts, the two applications will likely have implemented ad-hoc solutions of their own, which are incompatible with each other and thus can't be trivially merged together. Large, "macro-frameworks" may avoid the problems introduced by ad-hoc feature implementation, but instead run the risk of introducing too many features that remain unused.

Radiance initially grew out of the desire to write web applications that could be run together in such a way, that users would not have to create multiple accounts, track multiple logins, and so forth. Another primary goal of it was to write it in such a way, that it would be possible to exchange various parts of the framework without needing to change application code. This separation would allow flexibility on the side of the administrator, to pick and choose the parts that best fit their environment.

2. EXAMPLE REQUIREMENTS OF AN APPLICATION

In order to illustrate the reasoning behind the individual components present in the Radiance system, we are going to make use of a simple, imaginary example web application throughout the paper. This application should provide

a "blog platform," wherein users can create text posts to which they can link their friends. They should also be able to edit the posts again at a later point and customise the look and feel to their preference.

When talking about such an application in the context of Radiance, we think of it as a form of library that is then loaded together with Radiance to form a full webserver. This approach is in contrast to the deployment models of many other frameworks, where the "library" is deployed bundled tightly together with the framework as a single application.

In order to write our blog application, we are going to want a database that stores the posts, a system to handle user accounts and authentication, a cache system for performance, an HTTP server, and a template engine.

Imagine now, if you will, that an administrator also wanted to run a forum on the same host as well, perhaps in order to allow people to discuss and engage with each other about the various posts on the main blog. Requiring users to maintain a login for each service would be annoying, to say the least. Furthermore, as an administrator, we would like to make sure that as much data is kept in the same place as possible, and arranged in a similar way, to ease maintenance.

For our blog to share as many resources as it can with potential third-party applications residing in the same installation, it needs to rely on the framework to provide all of our required features, except perhaps for the template engine. Additionally, it needs to have some system that divides up the URL namespace between applications. As the application writer, we cannot have any presumptions about what the final setup will look like — what kinds of applications will be deployed together, and how the public URLs should resolve.

3. THE RADIANCE SYSTEM

The core of the Radiance system is rather small. Despite this, it can provide a plethora of features in the spirit of a macro-framework, if needed. Most of its features are pluggable, which is done through interfaces.

The division of the URL namespace is provided through the routing system, which allows both a convenient view for developers to work with, and a powerful way for administrators to configure the system to their liking without having to touch application code.

Finally, the management of a Radiance installation is handled through so-called "environments," which specify the configuration of the individual components.

3.1 Interfaces

Interfaces represent a form of contract. They allow you to define the signatures of constructs exposed from a package, such as functions, macros, classes, and so forth. These function signatures are accompanied by documentation that specifies their public behaviour. Through this system, Radiance can provide several "standard interfaces" that specify the behaviour of many features that are commonly needed in web applications, without actually having to implement them.

In order to make this system more concrete, let's look at an interface that might provide some sort of caching mechanism.

```
(define-interface cache
  (defun invalidate (name)
    "Causes the cached value of NAME to be re-computed.")
  (defmacro with-caching (name invalidate &body body)
    "Caches the return value if INVALIDATE is non-NIL."))
```

This construct defines a new package called cache, exports the symbols invalidate and with-caching from it, and installs stub definitions for the respective function and macro. With this interface specification in place, an application can start writing code against it. In our imaginary blog service, we could now cache the page for a specific post like so:

```
(defun post-page (id)
  (cache:with-caching id NIL
    (render-post (load-post id))))
```

However, as it currently is, this function will obviously not work. The with-caching macro is not implemented with any useful functionality. As the writer of the blog application, we don't need to know the specifics of how the caching is implemented, though. All we need to do is tell Radiance that we need this interface. This can be done by adding the interface as a dependency to our application's ASDF[3] system definition.

```
(asdf:defsystem blog-service
  ...
  :depends-on (...
              (:interface :cache)))
```

Radiance extends ASDF's dependency resolution mechanism to allow for this kind of declaration. When the blog-service system is now loaded, Radiance will notice the interface dependency and resolve it to an actual system that implements the desired caching functionality. This resolution is defined in the currently active environment, and can thus be configured by the administrator of the Radiance installation.

For completeness, let's look at an implementation of this cache interface. The implementation must be loadable through an ASDF system, so that the above dependency resolution can be done. The actual implementation mechanism is handed to us for free, as Lisp allows redefinition.

```
(defvar cache::*caches* (make-hash-table))

(defun cache:invalidate (name)
  (remhash name *caches*))

(defmacro cache:with-caching (name invalidate &body body)
  (once-only (name)
    `(or (and (not ,invalidate) (gethash ,name *caches*))
         (setf (gethash ,name *caches*)
               (progn ,@body)))))
```

This implementation is a particularly primitive and straightforward one, in order to keep things short. The function and macro are provided by just overriding the stub definitions that were in place previously. The variable definition is not

part of the official interface, and is instead exposed through an unexported symbol, denoting an implementation-dependant extension.

Using direct overwriting of definitions means that all applications must use the same implementation. However, usually this effect is intended, as we want to maximise the sharing between them. If an implementation should have special needs, it can always bypass the interfaces and make direct use of whatever it might depend on. This approach with interfaces does bring some benefits, however. For one, it allows the implementations to be as efficient as possible, as there is no intermediate layer of redirection.

Radiance provides standard interface definitions for all of the components we require for the blog application, and more. A full list of the interfaces and their descriptions is available in the Radiance documentation[1]. Thus, Radiance can be used like a macro-framework, but does not load any features unless specifically required. Additionally, any of the implementations can be exchanged by a system administrator for ones that more closely match their requirements, without having to change anything about the application code.

Finally, Radiance provides a system that allows the programmer to optionally depend on an interface. This mechanism is useful to model something like the user preferences mentioned in our example application. An administrator might not always want to provide an administration or settings panel. To make this dependency optional, we can defer the compilation and evaluation of our relevant logic to a later point, after an implementation has been loaded. For an imaginary `user-settings` interface, an example might look like the following:

```
(define-implement-trigger user-settings
  (user-settings:define-option ...))
```

Since all the symbols are already provided by the interface definition, there are no problems when the reader parses the source. The forms can thus simply be saved away to be evaluated once Radiance notices that an implementation of the interface in question has been loaded.

Ultimately, interfaces are a form of compromise between providing all possible features at once, and almost no features at all. The usefulness of an interface heavily depends on its specification, and for implementations to be really exchangeable without modifying the application code, each implementation and application must strictly adhere to the specification.

3.2 Routes

In order to allow the administrator to change where pages are accessible from, an application cannot hard-code its resource locations and the links in its templates. It is doubly important to do this when it comes to housing multiple applications in one, as the system needs to be set up in such a way that the applications do not clash with each other or potentially confuse pages of one another.

---

[1]https://github.com/Shirakumo/radiance/#interface

In many frameworks, like for example Symfony[4], this problem is solved by naming every resource in the system by a tag, and then allowing the configuration of what each tag resolves to individually. Radiance takes a different approach. It introduces the idea of two separate namespaces: an external one, which is what a visitor of a website sees and interacts with, and an internal one, which is what the programmer of an application deals with.

This separation allows application programmers to model their pages from a perspective that looks much more like the external view. Instead of dealing with named tags, they deal in concrete internal URLs that, in a development setup, often have a relatively direct one-to-one mapping to the external URLs. This makes it easier to visualise and think about the application structure. From the administrator's point of view this setup is more convenient too, as they simply need to think in terms of translations, rather than specific tag instances.

The translation between the two namespaces is the responsibility of the routing system. The way it is connected into the life-cycle of a request is illustrated in Figure 1.
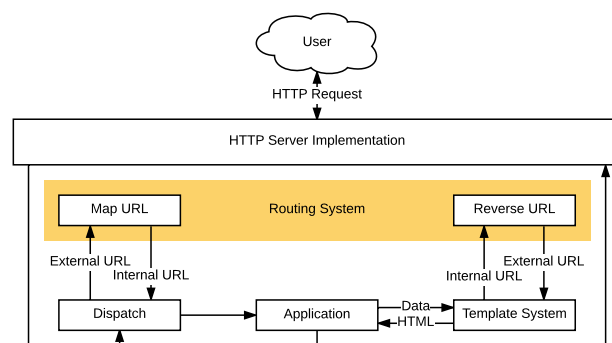

Figure 1: Standard request life-cycle

When a request is dispatched by Radiance, it first parses the request URL into an object presentation that makes it easier to modify. It then sends it through the mapping function of the routing system, which turn this external URL into an internal one. The request is then dispatched to the application that "owns" the URL in the internal representation.

Since Radiance has full control over the organisation of the internal representation, it can make strict demands as to how applications need to structure their URLs. Specifically, it requires each application to put all of its pages on a domain that is named after their application.

In our sample application, we might write the view page something like this, where `blog/view` is the internal URL that the view page is reachable on.

```
(define-page view "blog/view" ()
  (render-template (template "view.html")
                   (get-post (get-var "post-id"))))
```

If, as an administrator of an installation of this blog application, we now wanted to reach this page through the

URL `www.example.com/blog/view`, the mapping route functions would have to strip away the `www.example.com` domain, recognise the `blog` folder, and put that as the URL's domain instead. We can achieve this behaviour relatively easily with the following definition.

```
(define-route blog :mapping (uri)
  (when (begins-with "blog/" (path uri))
    (setf (domains uri) '("blog")
          (path uri) (subseq (path uri) 5))))
```

Alternatively, we could also use the even simpler string-based definition.

```
(define-string-route blog :mapping
  "/blog/(.*)" "blog/\\1")
```

Naturally, if you wanted different behaviour depending on which domain the request came from, you'd have to write a more specific translation.

With the mapping alone the case is not yet solved, though. When a page is emitted that contains links, the links must be translated as well, but in the opposite way. Unfortunately, because routes can be arbitrarily complex, it is not possible for the system to figure out a reversal route automatically. The logic of a reversal route will usually be much the same as it was for the mapping route and should thus not be a problem to figure out, though.

The reversal of URLs should receive special support from the templating system, as it is very frequently needed and should thus be short. Radiance does not dictate any specific template system, but offers extensions to some existing systems like Clip[5] to simplify this process. As a brief example, in Clip, URLs can be denoted through special attributes like this:

```
<a @href="blog/list">Latest Blog Posts</a>
<form method="post">
  <textarea name="text"></textarea>
  <input type="submit" @formaction="blog/submit" />
</form>
```

The templating engine takes care of reading out the `@` attribute values, running them through the reversal functions of the routing system, and putting the resulting URL into the corresponding attribute to produce a valid link.

3.3 Environments

Radiance's "environment" encapsulates the configuration of a Radiance installation. Through it, applications can provide settings that the administrator can set to their liking. It also provides the information necessary to figure out which implementation to use for an interface.

The environment itself simply dictates a directory that contains all of these configuration files. Each application in the system automatically receives its own directory in which it can store both configuration and temporary data files. By simply switching out this environment directory, the system can then be loaded into completely different configurations, for example allowing you to easily keep "development" and "production" setups.

An administrator of a system will not have to touch any source code, and instead can configure the system through a set of human-readable configuration files. It is, of course, still possible to configure the system through usual Lisp code as well, should one prefer this approach.

In our example application we might want to give the administrator the ability to define a global blog title. We could provide a default value for it like this:

```
(define-trigger startup ()
  (defaulted-config "Irradiant Blogs" :title))
```

We need to stick the configuration update into a trigger, in order to defer the evaluation to when Radiance is being started up and the environment has been decided. During the loading of the system, the environment might not have been set yet, and we would not be able to access the configuration storage.

While Radiance does provide default implementations for all of its interfaces, it is likely that some of them are not usable for a production setting. In order to change, say, the implementation of the database interface, we would then have to modify the Radiance core's configuration file. We can either modify the file directly like so:

```
((:interfaces (:database . "i-postmodern")
              ...)
 ...)
```

...or instead use the programmatical way, like so:

```
(setf (mconfig :radiance-core :interfaces :database)
      "i-postmodern")
```

`i-postmodern` here is the name of a standard implementation of the database interface for PostgreSQL databases.

Routes can also be configured through the core configuration file. Our previous example mapping route could be set up in the configuration file like this.

```
((:routes (blog :mapping "/blog/(.*)" "blog/\\1"))
 ...)
```

Radiance will take care of converting the configuration data into an actual route definition like we saw above.

The individual configuration of every application can be changed in much the same way.

4. CONCLUSION

Radiance provides a web framework that adjusts itself depending on how many features the application requires. By separating the applications from the implementations of these features with an interface, it allows the application programmer to write their software against a well-specified API, and retains the ability for the administrator to decide which implementation is most suitable for their setup.

By maintaining a strict separation of the URL namespace and providing an automated URL rewriting mechanism, Radiance allows for easy sharing of the namespace between an arbitrary number of applications, while at the same time giving the administrator a convenient way to modify the behaviour of the translation to fit their specific server configuration.

Through the environment system, Radiance standardises the way each application is configured by the administrator and how the system is pieced together when it is loaded. As a consequence of that standardisation it becomes trivial to switch between different setups of a Radiance installation. The simple, human-readable configuration format used allows even users without intimate knowledge of Lisp to set up an instance.

5. FURTHER WORK

Currently, while it is possible to dynamically load applications, it is not possible to dynamically unload them. Doing so is troublesome, as the system of an application might potentially modify any other part of the Lisp process. However, if a constraint on what kind of changes can be rolled back is introduced, it should be possible to provide a usable form of this kind of feature.

Radiance also does not allow you to change the implementation of an interface on the fly. This feature has much the same problems as the previous issue. In addition, though, the system needs to ensure that all dependant applications, including optionally dependant ones, are reloaded to make macro redefinitions take effect. Furthermore, since some interfaces expose classes and instances thereof, the system would either have to be able to migrate objects between interface implementations, or somehow invalidate the old instances if they are retained in another part of the application.

Ultimately it should be made possible to switch out the environment of a Radiance installation on the fly, for example to switch between development and production setups without having to restart completely. Doing so could massively improve the time needed to discover differences between different setups.

6. ACKNOWLEDGEMENTS

7. REFERENCES

[1] N. Hafner. Radiance, a web application envrionment. [Online; accessed 2016.12.16] http://shirakumo.org/projects/radiance.
[2] Various. Microframework. [Online; accessed 2016.12.16] https://en.wikipedia.org/wiki/Microframework.
[3] Various. Another system definition facility. [Online; accessed 2017.3.14] https://en.wikipedia.org/wiki/Another_System_Definition_Facility.
[4] Various. Symfony. [Online; accessed 2017.3.13] https://en.wikipedia.org/wiki/Symfony.
[5] N. Hafner. Clip, a common lisp html templating engine. [Online; accessed 2016.12.16] https://github.com/Shinmera/clip.