

Continue: Web Applications in Racket

Version 5.0

Danny Yoo <dyoo@cs.wpi.edu>
and Jay McCarthy <jay@cs.byu.edu>

June 6, 2010

How do we make dynamic web applications? This tutorial will show how we can build web applications using Racket. As our working example, we'll build a simple web journal (a "blog"). We'll cover how to start up a web server, how to generate dynamic web content, and how to interact with the user.

The target audience for this tutorial are students who've gone through the design and use of structures in *How to Design Programs*, with some higher-order functions, `local`, and a minor bit of mutation.

1 Getting Started

Everything you needed in this tutorial is provided in Racket. We will be using the DrRacket Module language. Enter the following into the Definition window.

```
#lang web-server/insta
(define (start request)
  '(html
    (head (title "My Blog"))
    (body (h1 "Under construction"))))
```

Press the Run button. If a web browser comes up with an “Under Construction” page, then clap your hands with delight: you’ve built your first web application! It doesn’t do much yet, but we will get there. Press the Stop button to shut the server down for now.

2 The Application

We want to motivate this tutorial by showing how to develop a blog. Users should be able to create posts and add comments to any posts. We'll take an iterative approach, with one or two pitfalls along the way. The game plan, roughly, will be:

- Show a static list of posts.
- Allow a user to add new posts to the system.
- Extend the model to let a user add comments to a post.
- Allow all users to share the same set of posts.
- Serialize our data structures to disk.

By the end of this tutorial, we'll have a simple blogging application.

3 Basic Blog

We start by considering our data definitions. We want to represent a list of posts. Let's say that a post is:

```
(define-struct post (title body))
```

```
(struct post (title body)
  #:extra-constructor-name make-post)
title : string?
body : string?
```

Exercise. Make a few examples of posts.

A blog, then, will be a list of posts:

```
blog : (listof post?)
```

As a very simple example of a blog:

```
(define BLOG (list (make-post "First Post!"
  "Hey, this is my first post!")))
```

Now that we have a sample blog structure, let's get our web application to show it.

4 Rendering HTML

When a web browser visits our application's URL, the browser constructs a request structure and sends it off to our web application. Our start function will consume requests and produce responses. One basic kind of response is to show an HTML page.

```
(define html-response/c
  (flat-rec-contract
    html-response
    (or/c string?
      (or/c (cons/c symbol? (listof html-response))
        (cons/c symbol?
          (cons/c (listof (list/c symbol? string?))
            (listof html-response)))))))
```

For example:

The HTML `hello` is represented as `"hello"`. Strings are automatically escaped when output. This guarantees valid HTML. Therefore, the value `"Unfinished tag"` is rendered as `Unfinished tag` not `Unfinished tag`. Similarly, `"<i>Finished tag</i>"` is rendered as `<i>Finished tag</i>`; not `<i>Finished tag</i>`.

```
<p>This is an example</p> is
```

```
'(p "This is an example").
```

```
<a href="link.html">Past</a> is
```

```
'(a ((href "link.html")) "Past").
```

```
<p>This is <div class="emph">another</div> example.</p> is
```

```
'(p "This is " (div ((class "emph")) "another") " example.").
```

We can produce these `html-responses` by using `cons` and `list` directly. Doing so, however, can be notationally heavy. Consider:

```
(list 'html (list 'head (list 'title "Some title"))
      (list 'body (list 'p "This is a simple static page.")))
```

vs:

```
'(html (head (title "Some title"))
      (body (p "This is a simple static page.")))
```

They both produce the same `html-response`, but the latter is a lot easier to type and read. We've been using the extended list abbreviation form described in Section 13 of *How to Design Programs*: by using a leading forward quote mark to concisely represent the list structure, we can construct static html responses with `aplomb`.

However, we can run into a problem when we use simple list abbreviation with dynamic content. If we have expressions to inject into the `html-response` structure, we can't use a simple list-abbreviation approach because those expressions will be treated literally as part of the list structure!

We want a notation that gives us the convenience of quoted list abbreviations, but with the option to treat a portion of the structure as a normal expression. That is, we would like to define a template whose placeholders can be easily expressed and filled in dynamically.

Racket provides this templating functionality with quasiquotation. Quasiquotation uses a leading back-quote in front of the whole structure. Like regular quoted list abbreviation, the majority of the list structure will be literally preserved in the nested list result. In places where we'd like a subexpression's value to be plugged in, we prepend an unquoting comma in front of the subexpression. As an example:

```
; render-greeting: string -> html-response
; Consumes a name, and produces a dynamic html-response.
(define (render-greeting a-name)
  `(html (head (title "Welcome"))
        (body (p ,(string-append "Hello " a-name)))))
```

Exercise. Write a function that consumes a `post` and produces an `html-response` representing that content.

```
render-post : (post? . -> . html-response/c)
```

As an example, we want:

```
(render-post (make-post "First post!" "This is a first post.))
```

to produce:

```
'(div ((class "post")) "First post!" (p "This is a first post.))
```

Exercise. Revise `render-post` to show the number of comments attached to a post.

If an expression produces a list of `html-response` fragments, we may want to splice in the elements of a list into our template, rather plug in the whole list itself. In these situations, we can use the splicing form `,@expression`.

As an example, we may want a helper function that transforms a `html-response` list into a fragment representing an unordered, itemized HTML list:

```
; render-as-itemized-list: (listof html-response) -> html-response
; Consumes a list of items, and produces a rendering
; as an unordered list.
(define (render-as-itemized-list fragments)
  '(ul ,(map render-as-item fragments)))

; render-as-item: html-response -> html-response
; Consumes an html-response, and produces a rendering
; as a list item.
(define (render-as-item a-fragment)
  '(li ,a-fragment))
```

Exercise. Write a function `render-posts` that consumes a `(listof post?)` and produces an `html-response` for that content.

```
render-posts : ((listof post?) . -> . html-response/c)
```

As examples:

```
(render-posts empty)
```

should produce:

```
'(div ((class "posts")))
```

While

```
(render-posts (list (make-post "Post 1" "Body 1")
                    (make-post "Post 2" "Body 2")))
```

should produce:

```
'(div ((class "posts"))
      (div ((class "post")) "Post 1" "Body 1")
      (div ((class "post")) "Post 2" "Body 2"))
```

Now that we have the `render-posts` function handy, let's revisit our web application and change our `start` function to return an interesting `html-response`.

```
#lang web-server/insta
```

```

; A blog is a (listof post)
; and a post is a (make-post title body)
(define-struct post (title body))

; BLOG: blog
; The static blog.
(define BLOG
  (list (make-post "First Post" "This is my first post")
        (make-post "Second Post" "This is another post")))

; start: request -> html-response
; Consumes a request, and produces a page that displays all of the
; web content.
(define (start request)
  (render-blog-page BLOG request))

; render-blog-page: blog request -> html-response
; Consumes a blog and a request, and produces an html-response page
; of the content of the blog.
(define (render-blog-page a-blog request)
  '(html (head (title "My Blog"))
         (body (h1 "My Blog")
               ,(render-posts a-blog))))

; render-post: post -> html-response
; Consumes a post, produces an html-response fragment of the post.
(define (render-post a-post)
  '(div ((class "post"))
        ,(post-title a-post)
        (p ,(post-body a-post))))

; render-posts: blog -> html-response
; Consumes a blog, produces an html-response fragment
; of all its posts.
(define (render-posts a-blog)
  '(div ((class "posts"))
        ,@(map render-post a-blog)))

```

If we press Run, we should see the blog posts in our web browser.

5 Inspecting Requests

Our application still seems a bit static: although we're building the page dynamically, we haven't yet provided a way for an external user to add new posts. Let's tackle that now. Let's provide a form that will let the user add a new blog entry. When the user presses the submit button, we want the user to see the new post at the top of the page.

Until now, we've been passing around a `request` object without doing anything with it. As we might expect, the `request` object isn't meant to be ignored so much! When a user fills out a web form and submits it, that user's browser constructs a new `request` that holds the form values in it. We can use the function `request-bindings` to grab at the values that the user has filled out. The type of `request-bindings` is:

```
request-bindings : (request? . -> . bindings?)
```

Along with `request-bindings`, there's another function called `extract-binding/single` that takes this as well as a name, and returns the value associated to that name.

```
extract-binding/single : (symbol? bindings? . -> . string?)
```

Finally, we can check to see if a name exists in a binding with `exists-binding?`:

```
exists-binding? : (symbol? bindings? . -> . boolean?)
```

With these functions, we can design functions that consume `requests` and do something useful.

Exercise. Write a function `can-parse-post?` that consumes a `bindings?`. It should produce `#t` if there exist bindings both for the symbols `'title` and `'body`, and `#f` otherwise.

```
can-parse-post? : (bindings? . -> . boolean?)
```

Exercise. Write a function `parse-post` that consumes a `bindings`. Assume that the `bindings` structure has values for the symbols `'title` and `'body`. `parse-post` should produce a post containing those values.

```
parse-post : (bindings? . -> . post?)
```

Now that we have these helper functions, we can extend our web application to handle form input. We'll add a small form at the bottom, and adjust our program to handle the addition of new posts. Our start method, then, will first see if the request has a parsable post, extend its set of posts if it can, and finally display those blog posts.

```

#lang web-server/insta

; A blog is a (listof post)
; and a post is a (make-post title body)
(define-struct post (title body))

; BLOG: blog
; The static blog.
(define BLOG
  (list (make-post "First Post" "This is my first post")
        (make-post "Second Post" "This is another post")))

; start: request -> html-response
; Consumes a request and produces a page that displays all of the
; web content.
(define (start request)
  (local [(define a-blog
            (cond [(can-parse-post? (request-bindings request))
                   (cons (parse-post (request-bindings request))
                         BLOG)]
                  [else
                     BLOG]))])
    (render-blog-page a-blog request)))

; can-parse-post?: bindings -> boolean
; Produces true if bindings contains values for 'title and 'body.
(define (can-parse-post? bindings)
  (and (exists-binding? 'title bindings)
        (exists-binding? 'body bindings)))

; parse-post: bindings -> post
; Consumes a bindings, and produces a post out of the bindings.
(define (parse-post bindings)
  (make-post (extract-binding/single 'title bindings)
             (extract-binding/single 'body bindings)))

; render-blog-page: blog request -> html-response
; Consumes a blog and a request, and produces an html-response page
; of the content of the blog.
(define (render-blog-page a-blog request)
  '(html (head (title "My Blog"))
        (body
         (h1 "My Blog")
         ,(render-posts a-blog))))

```

```

      (form
        (input ((name "title")))
        (input ((name "body")))
        (input ((type "submit"))))))

; render-post: post -> html-response
; Consumes a post, produces an html-response fragment of the post.
(define (render-post a-post)
  '(div ((class "post"))
    ,(post-title a-post)
    (p ,(post-body a-post))))

; render-posts: blog -> html-response
; Consumes a blog, produces an html-response fragment
; of all its posts.
(define (render-posts a-blog)
  '(div ((class "posts"))
    ,@(map render-post a-blog)))

```

This appears to work... but there's an issue with this! Try to add two new posts. What happens?

6 Advanced Control Flow

For the moment, let's ignore the admittedly huge problem of having a blog that only accepts one new blog entry. Don't worry! We will fix this.

But there's a higher-level problem with our program: although we do have a function, `start`, that can respond to requests directed at our application's URL, that `start` function is starting to get overloaded with a lot of responsibility. Conceptually, `start` is now handling two different kinds of requests: it's either a request for showing a blog, or a request for adding a new blog post.

What's happening is that `start` is becoming a traffic cop — a dispatcher — for all the behavior of our web application. As far as we know, if we want to add more functionality to our application, `start` needs to know how to deal. Can we get different kinds of requests to automatically direct themselves to different functions?

The web server library provides a function, `send/suspend/dispatch`, that allows us to create URLs that direct to different parts of our application. Let's demonstrate a dizzying example. In a new file, enter the following in the definition window.

```
#lang web-server/insta
; start: request -> html-response
(define (start request)
  (phase-1 request))

; phase-1: request -> html-response
(define (phase-1 request)
  (local [(define (response-generator embed/url)
            `(html
              (body (h1 "Phase 1")
                    (a ((href ,(embed/url phase-2)))
                       "click me!"))))]
          (send/suspend/dispatch response-generator)))

; phase-2: request -> html-response
(define (phase-2 request)
  (local [(define (response-generator embed/url)
            `(html
              (body (h1 "Phase 2")
                    (a ((href ,(embed/url phase-1)))
                       "click me!"))))]
          (send/suspend/dispatch response-generator)))
```

This is a web application that goes round and round. When a user first visits the application, the user starts off in `phase-1`. The page that's generated has a hyperlink that, when clicked, continues to `phase-2`. The user can click back, and falls back to `phase-1`, and the cycle

repeats.

Let's look more closely at the `send/suspend/dispatch` mechanism. `send/suspend/dispatch` consumes a response-generating function, and it gives that response-generator a function called `embed/url` that we'll use to build special URLs. What makes these URLs special is this: when a web browser visits these URLs, our web application restarts, but not from start, but from the handler that we associate to the URL. In `phase-1`, the use of `embed/url` associates the link with `phase-2`, and vice versa.

We can be more sophisticated about the handlers associated with `embed/url`. Because the handler is just a request-consuming function, it can be defined within a `local`. Consequently, a local-defined handler knows about all the variables that are in the scope of its definition. Here's another loopy example:

```
#lang web-server/insta
; start: request -> html-response
(define (start request)
  (show-counter 0 request))

; show-counter: number request -> html-response
; Displays a number that's hyperlinked: when the link is pressed,
; returns a new page with the incremented number.
(define (show-counter n request)
  (local [(define (response-generator embed/url)
            `(html (head (title "Counting example"))
                  (body
                    (a ((href ,(embed/url next-number-handler)))
                       ,(number->string n))))))

          (define (next-number-handler request)
            (show-counter (+ n 1) request))]

    (send/suspend/dispatch response-generator)))
```

This example shows that we can accumulate the results of an interaction. Even though the user starts off by visiting and seeing zero, the handlers produced by `next-number-handler` continue the interaction, accumulating a larger and larger number.

Now that we've been going a little bit in circles, let's move forward back to the blog application. We will adjust the form's action so it directs to a URL that's associated to a separate handler.

```
#lang web-server/insta

; A blog is a (listof post)
; and a post is a (make-post title body)
(define-struct post (title body))
```

```

; BLOG: blog
; The static blog.
(define BLOG
  (list (make-post "First Post" "This is my first post")
        (make-post "Second Post" "This is another post")))

; start: request -> html-response
; Consumes a request and produces a page that displays all of the
; web content.
(define (start request)
  (render-blog-page BLOG request))

; parse-post: bindings -> post
; Extracts a post out of the bindings.
(define (parse-post bindings)
  (make-post (extract-binding/single 'title bindings)
            (extract-binding/single 'body bindings)))

; render-blog-page: blog request -> html-response
; Consumes a blog and a request, and produces an html-response page
; of the content of the blog.
(define (render-blog-page a-blog request)
  (local [(define (response-generator make-url)
            `(html (head (title "My Blog"))
                  (body
                    (h1 "My Blog")
                    ,(render-posts a-blog)
                    (form ((action
                          ,(make-url insert-post-handler)))
                          (input ((name "title")))
                          (input ((name "body")))
                          (input ((type "submit"))))))))

            (define (insert-post-handler request)
              (render-blog-page
                (cons (parse-post (request-bindings request))
                      a-blog)
                request))]
    (send/suspend/dispatch response-generator)))

; render-post: post -> html-response
; Consumes a post, produces an html-response fragment of the post.
(define (render-post a-post)
  `(div ((class "post"))

```

```
      ,(post-title a-post)
      (p ,(post-body a-post))))

; render-posts: blog -> html-response
; Consumes a blog, produces an html-response fragment
; of all its posts.
(define (render-posts a-blog)
  '(div ((class "posts"))
        ,(map render-post a-blog)))
```

Note that the structure of the `render-blog-page` function looks very similar to that of our last `show-counter` example. The user can finally add and see multiple posts to their blog.

Unfortunately, there's still a problem. To see the problem: add a few posts to the system, and then open up a new browser window. In the new browser window, visit the web application's URL. What happens?

7 Share and Share Alike

We have run into another flaw with our application: each browser window keeps track of its own distinct blog. That defeats the point a blog for most people, that is, to share with others! When we insert a new post, rather than create a new blog value, we'd like to make a structural change to the existing blog. (HTDP Chapter 41). So let's add mutation into the mix.

There's one small detail we need to touch: in the web-server language, structures are immutable by default. We'll want to override this default and get access to the structure mutators. To do so, we adjust our structure definitions with the `#:mutable` keyword.

Earlier, we had said that a `blog` was a list of `posts`, but because we want to allow the blog to be changed, let's revisit our definition so that a blog is a mutable structure:

```
(define-struct blog (posts) #:mutable)
```

```
(struct blog (posts)
  #:extra-constructor-name make-blog)
posts : (listof post?)
```

Mutable structures provide functions to change the fields of a structure; in this case, we now have a structure mutator called `set-blog-posts!`,

```
set-blog-posts! : (blog? (listof post?) . -> . void)
```

and this will allow us to change the posts of a blog.

Exercise. Write a function `blog-insert-post!`

```
blog-insert-post! : (blog? post? . -> . void)
```

The intended side effect of the function will be to extend the blog's posts.

Since we've changed the data representation of a blog, we'll need to revise our web application to use the updated representation. One other thing to note is that, within the web application, because we're sharing the same blog value, we don't need to pass it around with our handlers anymore: we can get at the current blog through our `BL0G` variable.

After doing the adjustments incorporating `insert-blog-post!`, and doing a little variable cleanup, our web application now looks like this:

```

#lang web-server/insta

; A blog is a (make-blog posts)
; where posts is a (listof post)
(define-struct blog (posts) #:mutable)

; and post is a (make-post title body)
; where title is a string, and body is a string
(define-struct post (title body))

; BLOG: blog
; The initial BLOG.
(define BLOG
  (make-blog
    (list (make-post "First Post" "This is my first post")
          (make-post "Second Post" "This is another post"))))

; blog-insert-post!: blog post -> void
; Consumes a blog and a post, adds the post at the top of the blog.
(define (blog-insert-post! a-blog a-post)
  (set-blog-posts! a-blog
    (cons a-post (blog-posts a-blog))))

; start: request -> html-response
; Consumes a request and produces a page that displays
; all of the web content.
(define (start request)
  (render-blog-page request))

; parse-post: bindings -> post
; Extracts a post out of the bindings.
(define (parse-post bindings)
  (make-post (extract-binding/single 'title bindings)
    (extract-binding/single 'body bindings)))

; render-blog-page: request -> html-response
; Produces an html-response page of the content of the BLOG.
(define (render-blog-page request)
  (local [(define (response-generator make-url)
    '(html (head (title "My Blog"))
      (body
        (h1 "My Blog")
        ,(render-posts)
        (form ((action
          ,(make-url insert-post-handler)))
          (input ((name "title"))))
        ))))
    (response-generator make-url)
    ))

```

```

        (input ((name "body")))
        (input ((type "submit"))))))))

(define (insert-post-handler request)
  (blog-insert-post!
   BLOG (parse-post (request-bindings request)))
  (render-blog-page request)])

(send/suspend/dispatch response-generator)))

; render-post: post -> html-response
; Consumes a post, produces an html-response fragment of the post.
(define (render-post a-post)
  '(div ((class "post"))
        ,(post-title a-post)
        (p ,(post-body a-post))))

; render-posts: -> html-response
; Consumes a blog, produces an html-response fragment
; of all its posts.
(define (render-posts)
  '(div ((class "posts"))
        ,@(map render-post (blog-posts BLOG))))

```

Open two windows that visit our web application, and start adding in posts from both windows. We should see that both browsers are sharing the same blog.

8 Extending the Model

Next, let's extend the application so that each post can hold a list of comments. We refine the data definition of a blog to be:

```
(struct post (title body comments)
  #:constructor-name make-post
  #:mutable)
title : string?
body : string?
comments : (listof string?)
```

Exercise. Write the updated data structure definition for posts. Make sure to make the structure mutable, since we intend to add comments to posts.

Exercise. Make up a few examples of posts.

Exercise. Define a function `post-add-comment!`

```
post-add-comment! : (post? string? . -> . void)
```

whose intended side effect is to add a new comment to the end of the post's list of comments.

Exercise. Adjust `render-post` so that the produced fragment will include the comments in an itemized list.

Exercise. Because we've extended a post to include comments, other post-manipulating parts of the application may need to be adjusted, such as uses of `make-post`. Identify and fix any other part of the application that needs to accommodate the post's new structure.

Once we've changed the data structure of the posts and adjusted our functions to deal with this revised structure, the web application should be runnable. The user may even see some of the fruits of our labor: if the initial `BLOG` has a post with a comment, the user should see those comments now. But obviously, there's something missing: the user doesn't have the user interface to add comments to a post!

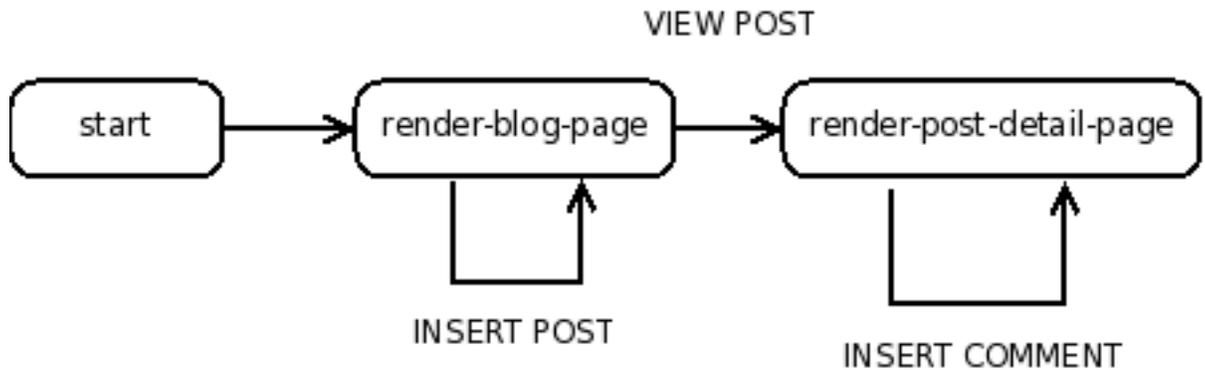
9 Breaking Up the Display

How should we incorporate comments more fully into the user's web experience? Seeing all the posts and comments all on one page may be a bit overwhelming. Perhaps we should hold off on showing the comments on the main blog page. Let's present a secondary "detail" view of a post, and present the comments there.

The top-level view of a blog then can show the blog's title and body. We can also show a count of how many comments are associated to the post.

So now we need some way to visit a post's detail page. One way to do this is to hyperlink each post's title: if the user wants to see the detail page of a post, user should be able to click the title to get there. From that post's detail page, we can even add a form to let the user add new comments.

Here's a diagram of a simple page flow of our web application that should let us add comments.



Each point in the diagram corresponds to a request-consuming handler. As we might suspect, we'll be using `send/suspend/dispatch` some more. Every arrow in the diagram will be realized as a URL that we generate with `embed/url`.

This has a slightly messy consequence: previously, we've been rendering the list of posts without any hyperlinks. But since any function that generates a special dispatching URL uses `embed/url` to do it, we'll need to adjust `render-posts` and `render-post` to consume and use `embed/url` itself when it makes those hyperlinked titles.

Our web application now looks like:

```
#lang web-server/insta

; A blog is a (make-blog posts)
; where posts is a (listof post)
```

```

(define-struct blog (posts) #:mutable)

; and post is a (make-post title body comments)
; where title is a string, body is a string,
; and comments is a (listof string)
(define-struct post (title body comments) #:mutable)

; BLOG: blog
; The initial BLOG.
(define BLOG
  (make-blog
    (list (make-post "First Post"
                    "This is my first post"
                    (list "First comment!"))
          (make-post "Second Post"
                    "This is another post"
                    (list))))))

; blog-insert-post!: blog post -> void
; Consumes a blog and a post, adds the post at the top of the blog.
(define (blog-insert-post! a-blog a-post)
  (set-blog-posts! a-blog
    (cons a-post (blog-posts a-blog))))

; post-insert-comment!: post string -> void
; Consumes a post and a comment string. As a side-effect,
; adds the comment to the bottom of the post's list of comments.
(define (post-insert-comment! a-post a-comment)
  (set-post-comments!
    a-post
    (append (post-comments a-post) (list a-comment))))

; start: request -> html-response
; Consumes a request, and produces a page that displays
; all of the web content.
(define (start request)
  (render-blog-page request))

; render-blog-page: request -> html-response
; Produces an html-response page of the content of the
; BLOG.
(define (render-blog-page request)
  (local [(define (response-generator make-url)
            '(html (head (title "My Blog"))
                  (body

```

```

        (h1 "My Blog")
        ,(render-posts make-url)
        (form ((action
                ,(make-url insert-post-handler)))
              (input ((name "title")))
              (input ((name "body")))
              (input ((type "submit"))))))))

; parse-post: bindings -> post
; Extracts a post out of the bindings.
(define (parse-post bindings)
  (make-post (extract-binding/single 'title bindings)
            (extract-binding/single 'body bindings)
            (list)))

(define (insert-post-handler request)
  (blog-insert-post!
   BLOG (parse-post (request-bindings request)))
  (render-blog-page request)])

(send/suspend/dispatch response-generator)))

; render-post-detail-page: post request -> html-response
; Consumes a post and request, and produces a detail page
; of the post. The user will be able to insert new comments.
(define (render-post-detail-page a-post request)
  (local [(define (response-generator make-url)
            '(html (head (title "Post Details"))
                  (body
                   (h1 "Post Details")
                   (h2 ,(post-title a-post))
                   (p ,(post-body a-post))
                   ,(render-as-itemized-list
                     (post-comments a-post))
                   (form ((action
                           ,(make-url insert-comment-handler)))
                         (input ((name "comment")))
                         (input ((type "submit"))))))))

          (define (parse-comment bindings)
            (extract-binding/single 'comment bindings))

          (define (insert-comment-handler a-request)
            (post-insert-comment!
             a-post (parse-comment (request-bindings a-request)))
            (render-post-detail-page a-post a-request))])

```

```

(send/suspend/dispatch response-generator)))

; render-post: post (handler -> string) -> html-response
; Consumes a post, produces an html-response fragment of the post.
; The fragment contains a link to show a detailed view of the post.
(define (render-post a-post make-url)
  (local [(define (view-post-handler request)
            (render-post-detail-page a-post request))]
    `(div ((class "post"))
          (a ((href ,(make-url view-post-handler)))
              ,(post-title a-post))
            (p ,(post-body a-post))
            (div ,(number->string (length (post-comments a-post)))
                  " comment(s)"))))

; render-posts: (handler -> string) -> html-response
; Consumes a make-url, and produces an html-response fragment
; of all its posts.
(define (render-posts make-url)
  (local [(define (render-post/make-url a-post)
            (render-post a-post make-url))]
    `(div ((class "posts"))
          ,(map render-post/make-url (blog-posts BLOG))))

; render-as-itemized-list: (listof html-response) -> html-response
; Consumes a list of items, and produces a rendering as
; an unordered list.
(define (render-as-itemized-list fragments)
  `(ul ,(map render-as-item fragments)))

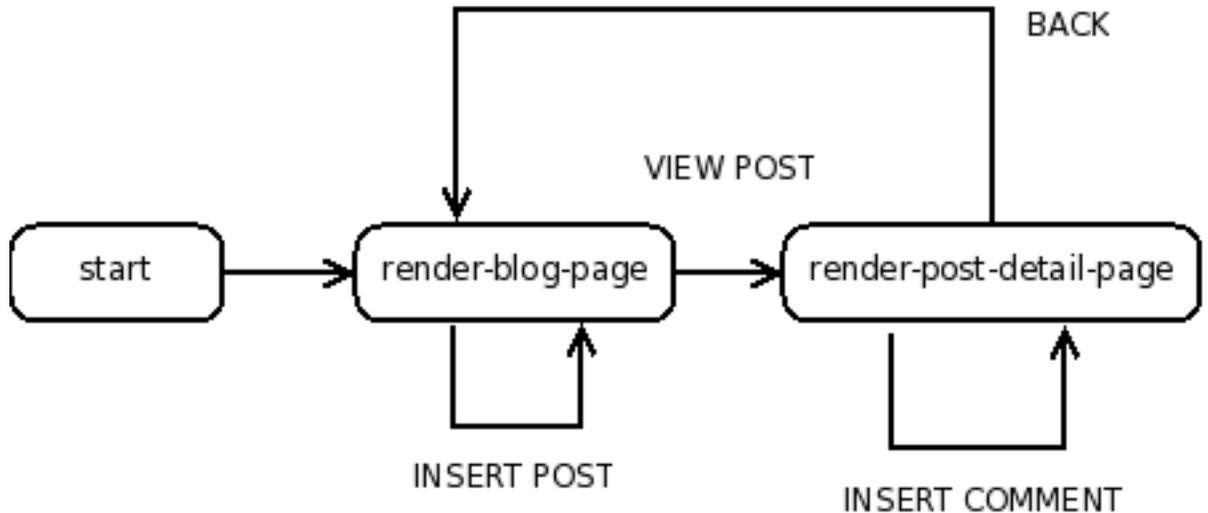
; render-as-item: html-response -> html-response
; Consumes an html-response, and produces a rendering
; as a list item.
(define (render-as-item a-fragment)
  `(li ,a-fragment))

```

We now have an application that's pretty sophisticated: we can add posts and write comments. Still, there's a problem with this: once the user's in a `post-detail-page`, they can't get back to the blog without pressing the browser's back button! That's disruptive. We should provide a page flow that lets us get back to the main blog-viewing page, to keep the user from every getting "stuck" in a dark corner of the web application.

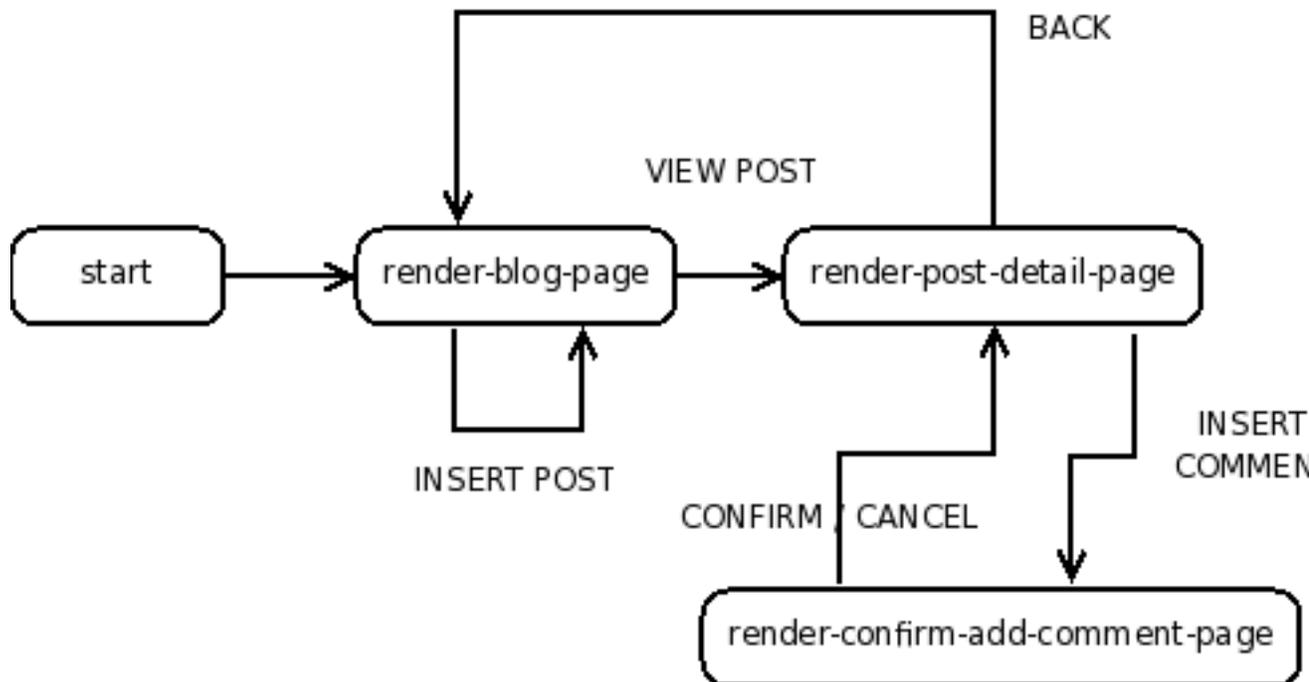
10 Adding a Back Button

Here's a diagram of our revised page flow of our web application. Maybe we can just add a BACK link from the `render-post-detail-page` that gets us back to viewing the top-level blog.



Exercise. Adjust `render-post-detail-page` to include another link that goes back to `render-blog-page`.

To make this more interesting, maybe we should enrich the flow a bit more. We can give the user a choice right before committing to their comment. Who knows? They may have a change of heart.



Although this seems complicated, the shape of our handlers will look more-or-less like what we had before. After we've added all the handlers, our web application is fairly functional.

```

#lang web-server/insta

; A blog is a (make-blog posts)
; where posts is a (listof post)
(define-struct blog (posts) #:mutable)

; and post is a (make-post title body comments)
; where title is a string, body is a string,
; and comments is a (listof string)
(define-struct post (title body comments) #:mutable)

; BLOG: blog
; The initial BLOG.
(define BLOG
  (make-blog
    (list (make-post "First Post"
                    "This is my first post"
                    (list "First comment!"))
          (make-post "Second Post"
                    "This is another post"
                    (list "Second comment!")))))
  
```

```

        (list))))))

; blog-insert-post!: blog post -> void
; Consumes a blog and a post, adds the post at the top of the blog.
(define (blog-insert-post! a-blog a-post)
  (set-blog-posts! a-blog
    (cons a-post (blog-posts a-blog))))

; post-insert-comment!: post string -> void
; Consumes a post and a comment string. As a side-effect,
; adds the comment to the bottom of the post's list of comments.
(define (post-insert-comment! a-post a-comment)
  (set-post-comments!
    a-post
    (append (post-comments a-post) (list a-comment))))

; start: request -> html-response
; Consumes a request and produces a page that displays
; all of the web content.
(define (start request)
  (render-blog-page request))

; render-blog-page: request -> html-response
; Produces an html-response page of the content of the
; BLOG.
(define (render-blog-page request)
  (local [(define (response-generator make-url)
    '(html (head (title "My Blog"))
      (body
        (h1 "My Blog")
        ,(render-posts make-url)
        (form ((action
          ,(make-url insert-post-handler)))
          (input ((name "title")))
          (input ((name "body")))
          (input ((type "submit"))))))))
    (define (parse-post: bindings -> post)
      ; Extracts a post out of the bindings.
      (define (parse-post bindings)
        (make-post (extract-binding/single 'title bindings)
          (extract-binding/single 'body bindings)
          (list)))
      (define (insert-post-handler request)

```

```

        (blog-insert-post!
         BLOG (parse-post (request-bindings request)))
        (render-blog-page request))]

(send/suspend/dispatch response-generator)))

; render-post-detail-page: post request -> html-response
; Consumes a post and produces a detail page of the post.
; The user will be able to either insert new comments
; or go back to render-blog-page.
(define (render-post-detail-page a-post request)
  (local [(define (response-generator make-url)
            `(html (head (title "Post Details"))
                   (body
                     (h1 "Post Details")
                     (h2 ,(post-title a-post))
                     (p ,(post-body a-post))
                     ,(render-as-itemized-list
                       (post-comments a-post))
                     (form ((action
                            ,(make-url insert-comment-handler)))
                           (input ((name "comment")))
                           (input ((type "submit"))))
                     (a ((href ,(make-url back-handler)))
                        "Back to the blog"))))
          (define (parse-comment bindings)
            (extract-binding/single 'comment bindings))
          (define (insert-comment-handler request)
            (render-confirm-add-comment-page
             (parse-comment (request-bindings request))
             a-post
             request))
          (define (back-handler request)
            (render-blog-page request))]
    (send/suspend/dispatch response-generator)))

; render-confirm-add-comment-page :
; comment post request -> html-response
; Consumes a comment that we intend to add to a post, as well
; as the request. If the user follows through, adds a comment
; and goes back to the display page. Otherwise, goes back to
; the detail page of the post.

```

```

(define (render-confirm-add-comment-page a-comment a-post request)
  (local [(define (response-generator make-url)
            `(html (head (title "Add a Comment"))
                  (body
                     (h1 "Add a Comment")
                     "The comment: " (div (p ,a-comment))
                     "will be added to "
                     (div ,(post-title a-post))

                     (p (a ((href ,(make-url yes-handler)))
                          "Yes, add the comment.))
                     (p (a ((href ,(make-url cancel-handler)))
                          "No, I changed my mind!"))))))

            (define (yes-handler request)
              (post-insert-comment! a-post a-comment)
              (render-post-detail-page a-post request))

            (define (cancel-handler request)
              (render-post-detail-page a-post request))])

    (send/suspend/dispatch response-generator)))

; render-post: (post (handler -> string) -> html-response
; Consumes a post, produces an html-response fragment of the post.
; The fragment contains a link to show a detailed view of the post.
(define (render-post a-post make-url)
  (local [(define (view-post-handler request)
            (render-post-detail-page a-post request))]
    `(div ((class "post"))
          (a ((href ,(make-url view-post-handler)))
              ,(post-title a-post))
          (p ,(post-body a-post))
          (div ,(number->string (length (post-comments a-post))
                                " comment(s))))))

; render-posts: (handler -> string) -> html-response
; Consumes a make-url, produces an html-response fragment
; of all its posts.
(define (render-posts make-url)
  (local [(define (render-post/make-url a-post)
            (render-post a-post make-url))]
    `(div ((class "posts"))
          ,@(map render-post/make-url (blog-posts BLOG))))))

; render-as-itemized-list: (listof html-response) -> html-response

```

```
; Consumes a list of items, and produces a rendering as
; an unordered list.
(define (render-as-itemized-list fragments)
  '(ul ,@(map render-as-item fragments)))

; render-as-item: html-response -> html-response
; Consumes an html-response, and produces a rendering
; as a list item.
(define (render-as-item a-fragment)
  '(li ,a-fragment))
```

11 Decorating With Style!

We have an application that's functionally complete, but is visual lacking. Let's try to improve its appearance. One way we can do this is to use a cascading style sheet. A style sheet can visual panache to our web pages. For example, if we'd like to turn all of our paragraphs green, we might add the following style declaration within our response.

```
'(style ((type "text/css")) "p { color: green }")
```

It's tempting to directly embed this style information into our `html-responses`. However, our source file is already quite busy. We often want to separate the logical representation of our application from its presentation. Rather than directly embed the `.css` in the HTML response, let's instead add a link reference to an separate `.css` file.

Until now, all the content that our web application has produced has come from a response generating handler. Of course, we know that not everything needs to be dynamically generated: it's common to have files that won't be changing. We should be able to serve these static resources (images, documents, `.css` files) alongside our web applications.

To do this, we set aside a path to store these files, and then tell the web server where that directory is. The function `static-files-path`,

```
static-files-path : (path-string? -> void)
```

tells the web server to look in the given path when it receives a URL that looks like a static resource request.

Exercise. Create a simple web application called `"test-static.rkt"` with the following content:

```
#lang web-server/insta
(define (start request)
  '(html (head (title "Testing"))
        (link ((rel "stylesheet")
              (href "/test-static.css")
              (type "text/css"))))
        (body (h1 "Testing")
              (h2 "This is a header")
              (p "This is " (span ((class "hot")) "hot") "."))))

(static-files-path "htdocs")
```

Make a subdirectory called `"htdocs"` rooted in the same directory as the `"test-static.rkt"` source. Finally, just to see that we can serve this `.css` page, create a very simple `.css` file `"test-static.css"` file in `"htdocs/"` with the following content:

```
body {
  margin-left: 10%;
  margin-right: 10%;
}
p { font-family: sans-serif }
h1 { color: green }
h2 { font-size: small }
span.hot { color: red }
```

At this point, run the application and look at the browser's output. We should see a Spartan web page, but it should still have some color in its cheeks.



Exercise. Improve the presentation of the blog web application by writing your an external style sheet to your tastes. Adjust all of the HTML response handlers to include a link to the style sheet.

12 The Double Submit Bug

There's yet another a subtle problem in our application. To see it, bring our blog application up again, and add a post. Then reload the page. Reload the page again.

What's happening is a well-known problem: it's an instance of the "double-submit" problem. If a user presses reload, a request is sent over to our application. This wouldn't be such a bad thing, if not for the fact that we're handling certain requests by mutating our application's data structures.

A common pattern that web developers use to dodge the double submission problem is to handle state-mutating request in a peculiar way: once the user sends over a request that affects change to the system, we then redirect them off to a different URL that's safe to reload. To make this happen, we will use the function `redirect/get`.

```
redirect/get : (-> request?)
```

This `redirect/get` function has an immediate side effect: it forces the user's browser to follow a redirection to a safe URL, and gives us back that fresh new request.

For example, let's look at a toy application that lets the users add names to a roster:

```
#lang web-server/insta

; A roster is a (make-roster names)
; where names is a list of string.
(define-struct roster (names) #:mutable)

; roster-add-name!: roster string -> void
; Given a roster and a name, adds the name
; to the end of the roster.
(define (roster-add-name! a-roster a-name)
  (set-roster-names! a-roster
    (append (roster-names a-roster)
            (list a-name))))

(define ROSTER (make-roster '("kathi" "shriram" "dan")))

; start: request -> html-response
(define (start request)
  (show-roster request))

; show-roster: request -> html-response
(define (show-roster request)
  (local [(define (response-generator make-url)
```

```

      '(html (head (title "Roster"))
            (body (h1 "Roster")
                  ,(render-as-itemized-list
                    (roster-names ROSTER))
                  (form ((action
                          ,(make-url add-name-handler)))
                        (input ((name "a-name")))
                        (input ((type "submit"))))))))
      (define (parse-name bindings)
        (extract-binding/single 'a-name bindings))

      (define (add-name-handler request)
        (roster-add-name!
         ROSTER (parse-name (request-bindings request)))
        (show-roster request))]
      (send/suspend/dispatch response-generator)))

; render-as-itemized-list: (listof html-response) -> html-response
(define (render-as-itemized-list fragments)
  '(ul ,@(map render-as-item fragments)))

; render-as-item: html-response -> html-response
(define (render-as-item a-fragment)
  '(li ,a-fragment))

```

This application suffers the same problem as our blog: if the user adds a name, and then presses reload, then the same name will be added twice.

We can fix this by changing a single expression. Can you see what changed?

```

#lang web-server/insta

; A roster is a (make-roster names)
; where names is a list of string.
(define-struct roster (names) #:mutable)

; roster-add-name!: roster string -> void
; Given a roster and a name, adds the name
; to the end of the roster.
(define (roster-add-name! a-roster a-name)
  (set-roster-names! a-roster
                    (append (roster-names a-roster)
                            (list a-name))))

(define ROSTER (make-roster '("kathi" "shriram" "dan")))

```

```

; start: request -> html-response
(define (start request)
  (show-roster request))

; show-roster: request -> html-response
(define (show-roster request)
  (local [(define (response-generator make-url)
            `(html (head (title "Roster"))
                  (body (h1 "Roster")
                        ,(render-as-itemized-list
                          (roster-names ROSTER))
                        (form ((action
                              ,(make-url add-name-handler)))
                              (input ((name "a-name")))
                              (input ((type "submit"))))))))
          (define (parse-name bindings)
            (extract-binding/single 'a-name bindings))

          (define (add-name-handler request)
            (roster-add-name!
             ROSTER (parse-name (request-bindings request)))
            (show-roster (redirect/get)))]
    (send/suspend/dispatch response-generator)))

; render-as-itemized-list: (listof html-response) -> html-response
(define (render-as-itemized-list fragments)
  `(ul ,@(map render-as-item fragments)))

; render-as-item: html-response -> html-response
(define (render-as-item a-fragment)
  `(li ,a-fragment))

```

Double-submit, then, is painlessly easy to mitigate. Whenever we have handlers that mutate the state of our system, we use `redirect/get` when we send our response back.

Exercise. Revise the blog application with `redirect/get` to address the double-submit problem.

With these minor fixes, our blog application now looks like this:

```

#lang web-server/insta

; A blog is a (make-blog posts)
; where posts is a (listof post)
(define-struct blog (posts) #:mutable)

```

```

; and post is a (make-post title body comments)
; where title is a string, body is a string,
; and comments is a (listof string)
(define-struct post (title body comments) #:mutable)

; BLOG: blog
; The initial BLOG.
(define BLOG
  (make-blog
    (list (make-post "First Post"
                    "This is my first post"
                    (list "First comment!"))
          (make-post "Second Post"
                    "This is another post"
                    (list))))))

; blog-insert-post!: blog post -> void
; Consumes a blog and a post, adds the post at the top of the blog.
(define (blog-insert-post! a-blog a-post)
  (set-blog-posts! a-blog
    (cons a-post (blog-posts a-blog))))

; post-insert-comment!: post string -> void
; Consumes a post and a comment string. As a side-effect,
; adds the comment to the bottom of the post's list of comments.
(define (post-insert-comment! a-post a-comment)
  (set-post-comments!
    a-post
    (append (post-comments a-post) (list a-comment))))

; start: request -> html-response
; Consumes a request and produces a page that displays
; all of the web content.
(define (start request)
  (render-blog-page request))

; render-blog-page: request -> html-response
; Produces an html-response page of the content of the
; BLOG.
(define (render-blog-page request)
  (local [(define (response-generator make-url)
            '(html (head (title "My Blog"))
                  (body
                    (h1 "My Blog")
                    ,(render-posts make-url)

```

```

        (form ((action
                ,(make-url insert-post-handler)))
              (input ((name "title")))
              (input ((name "body")))
              (input ((type "submit")))))

; parse-post: bindings -> post
; Extracts a post out of the bindings.
(define (parse-post bindings)
  (make-post (extract-binding/single 'title bindings)
            (extract-binding/single 'body bindings)
            (list)))

(define (insert-post-handler request)
  (blog-insert-post!
   BLOG (parse-post (request-bindings request)))
  (render-blog-page (redirect/get)))

(send/suspend/dispatch response-generator))

; render-post-detail-page: post request -> html-response
; Consumes a post and produces a detail page of the post.
; The user will be able to either insert new comments
; or go back to render-blog-page.
(define (render-post-detail-page a-post request)
  (local [(define (response-generator make-url)
            '(html (head (title "Post Details"))
                  (body
                     (h1 "Post Details")
                     (h2 ,(post-title a-post))
                     (p ,(post-body a-post))
                     ,(render-as-itemized-list
                       (post-comments a-post))
                     (form ((action
                             ,(make-url insert-comment-handler)))
                           (input ((name "comment")))
                           (input ((type "submit"))))
                     (a ((href ,(make-url back-handler)))
                        "Back to the blog"))))
          ])
    (define (parse-comment bindings)
      (extract-binding/single 'comment bindings))

    (define (insert-comment-handler request)
      (render-confirm-add-comment-page
       (parse-comment (request-bindings request)))

```

```

        a-post
        request))

(define (back-handler request)
  (render-blog-page request))

(send/suspend/dispatch response-generator))

; render-confirm-add-comment-page :
; comment post request -> html-response
; Consumes a comment that we intend to add to a post, as well
; as the request. If the user follows through, adds a comment
; and goes back to the display page. Otherwise, goes back to
; the detail page of the post.
(define (render-confirm-add-comment-page a-comment a-post request)
  (local [(define (response-generator make-url)
            `(html (head (title "Add a Comment"))
                  (body
                    (h1 "Add a Comment")
                    "The comment: " (div (p ,a-comment))
                    "will be added to "
                    (div ,(post-title a-post))

                    (p (a ((href ,(make-url yes-handler)))
                        "Yes, add the comment.))
                    (p (a ((href ,(make-url cancel-handler)))
                        "No, I changed my mind!"))))))

          (define (yes-handler request)
            (post-insert-comment! a-post a-comment)
            (render-post-detail-page a-post (redirect/get)))

          (define (cancel-handler request)
            (render-post-detail-page a-post request))])

  (send/suspend/dispatch response-generator))

; render-post: post (handler -> string) -> html-response
; Consumes a post, produces an html-response fragment of the post.
; The fragment contains a link to show a detailed view of the post.
(define (render-post a-post make-url)
  (local [(define (view-post-handler request)
            (render-post-detail-page a-post request))]

          `(div ((class "post"))
                (a ((href ,(make-url view-post-handler)))
                    ,(post-title a-post))
                ))))

```

```

      (p ,(post-body a-post))
      (div ,(number->string (length (post-comments a-post)))
           " comment(s)"))))

; render-posts: (handler -> string) -> html-response
; Consumes a make-url, produces an html-response fragment
; of all its posts.
(define (render-posts make-url)
  (local [(define (render-post/make-url a-post)
            (render-post a-post make-url))]
    `(div ((class "posts"))
          ,(map render-post/make-url (blog-posts BLOG))))))

; render-as-itemized-list: (listof html-response) -> html-response
; Consumes a list of items, and produces a rendering as
; an unordered list.
(define (render-as-itemized-list fragments)
  `(ul ,(map render-as-item fragments)))

; render-as-item: html-response -> html-response
; Consumes an html-response, and produces a rendering
; as a list item.
(define (render-as-item a-fragment)
  `(li ,a-fragment))

```

13 Abstracting the Model

If we "turn off the lights" by closing the program, then the state of our application disappears into the ether. How do we get our ephemeral state to stick around? Before we tackle that question, we should consider: what do we want to save? There's some state that we probably don't have a lingering interest in, like requests. What we care about saving is our model of the blog.

If we look closely at our web application program, we see a seam between the model of our blog, and the web application that uses that model. Let's isolate the model: it's all the stuff near the top:

```
(define-struct blog (posts) #:mutable)
(define-struct post (title body comments) #:mutable)
(define BLOG ...)
(define (blog-insert-post! ...) ...)
(define (post-insert-comment! ...) ...)
```

In realistic web applications, the model and the web application are separated by some wall of abstraction. The theory is that, if we do this separation, it should be easier to then make isolated changes without breaking the entire system. Let's do this: we will first rip the model out into a separate file. Once we've done that, then we'll look into making the model persist.

Create a new file called "model.rkt" with the following content.

```
#lang racket

; A blog is a (make-blog posts)
; where posts is a (listof post)
(define-struct blog (posts) #:mutable)

; and post is a (make-post title body comments)
; where title is a string, body is a string,
; and comments is a (listof string)
(define-struct post (title body comments) #:mutable)

; BLOG: blog
; The initial BLOG.
(define BLOG
  (make-blog
    (list (make-post "First Post"
                    "This is my first post"
                    (list "First comment!"))
          (make-post "Second Post"
                    "This is another post"
                    (list))))))
```

```

; blog-insert-post!: blog post -> void
; Consumes a blog and a post, adds the post at the top of the blog.
(define (blog-insert-post! a-blog a-post)
  (set-blog-posts!
   a-blog
   (cons a-post (blog-posts a-blog))))

; post-insert-comment!: post string -> void
; Consumes a post and a comment string. As a side-effect,
; adds the comment to the bottom of the post's list of comments.
(define (post-insert-comment! a-post a-comment)
  (set-post-comments!
   a-post
   (append (post-comments a-post) (list a-comment))))

(provide (all-defined-out))

```

This is essentially a cut-and-paste of the lines we identified as our model. It's written in the `racket` language because the model shouldn't need to worry about web-server stuff. There's one additional expression that looks a little odd at first:

```
(provide (all-defined-out))
```

which tells Racket to allow other files to have access to everything that's defined in the `"model.rkt"` file.

We change our web application to use this model. Going back to our web application, we rip out the old model code, and replace it with an expression that let's use use the new model.

```
(require "model.rkt")
```

which hooks up our web application module to the `"model.rkt"` module.

```

#lang web-server/insta

(require "model.rkt")

; start: request -> html-response
; Consumes a request and produces a page that displays
; all of the web content.
(define (start request)
  (render-blog-page request))

; render-blog-page: request -> html-response
; Produces an html-response page of the content of the
; BLOG.

```



```

(define (parse-comment bindings)
  (extract-binding/single 'comment bindings))

(define (insert-comment-handler request)
  (render-confirm-add-comment-page
   (parse-comment (request-bindings request))
   a-post
   request))

(define (back-handler request)
  (render-blog-page request))

(send/suspend/dispatch response-generator))

; render-confirm-add-comment-page :
; comment post request -> html-response
; Consumes a comment that we intend to add to a post, as well
; as the request. If the user follows through, adds a comment
; and goes back to the display page. Otherwise, goes back to
; the detail page of the post.
(define (render-confirm-add-comment-page a-comment a-post request)
  (local [(define (response-generator make-url)
            `(html (head (title "Add a Comment"))
                  (body
                   (h1 "Add a Comment")
                   "The comment: " (div (p ,a-comment))
                   "will be added to "
                   (div ,(post-title a-post))

                   (p (a ((href ,(make-url yes-handler)))
                        "Yes, add the comment.))
                   (p (a ((href ,(make-url cancel-handler)))
                        "No, I changed my mind!"))))))
          (define (yes-handler request)
            (post-insert-comment! a-post a-comment)
            (render-post-detail-page a-post (redirect/get)))
          (define (cancel-handler request)
            (render-post-detail-page a-post request))]
    (send/suspend/dispatch response-generator))

; render-post: post (handler -> string) -> html-response
; Consumes a post, produces an html-response fragment of the post.
; The fragment contains a link to show a detailed view of the post.

```

```

(define (render-post a-post make-url)
  (local [(define (view-post-handler request)
            (render-post-detail-page a-post request))]
    `(div ((class "post"))
          (a ((href ,(make-url view-post-handler)))
              ,(post-title a-post))
            (p ,(post-body a-post))
            (div ,(number->string (length (post-comments a-post)))
                  " comment(s))))))

; render-posts: (handler -> string) -> html-response
; Consumes a make-url, produces an html-response fragment
; of all its posts.
(define (render-posts make-url)
  (local [(define (render-post/make-url a-post)
            (render-post a-post make-url))]
    `(div ((class "posts"))
          ,@(map render-post/make-url (blog-posts BLOG))))))

; render-as-itemized-list: (listof html-response) -> html-response
; Consumes a list of items, and produces a rendering as
; an unordered list.
(define (render-as-itemized-list fragments)
  `(ul ,@(map render-as-item fragments)))

; render-as-item: html-response -> html-response
; Consumes an html-response, and produces a rendering
; as a list item.
(define (render-as-item a-fragment)
  `(li ,a-fragment))

```

14 A Persistent Model

Now that the model is separated into a separate module, we can more easily modify its functionality, and in particular, make it persistent.

The first step is to make the model structures serializable. Earlier, we made the structures mutable by adding `#:mutable` to their definitions. We can make the structures serializable by adding `#:prefab`. This tells Racket that these structures can be "previously fabricated", that is, created before the program started running—which is exactly what we want when restoring the blog data from disk. Our blog structure definition now looks like:

```
(define-struct blog (posts) #:mutable #:prefab)
```

Now `blog` structures can be read from the outside world with `read` and written with `write`. However, we need to make sure everything inside a `blog` structure is also marked as `#:prefab`. If we had a more complicated structure, we would need to ensure that everything (transitively) in the structure was `#:prefab`'d.

Exercise. Write the new structure definition for posts.

At this point, we *can* read and write the blog to disk. Now let's actually do it.

First, we'll make a place to record in the model where the blog lives on disk. So, we need to change the blog structure again. Now it will be:

```
(struct blog (home posts)
  #:constructor-name make-blog
  #:mutable)
home : string?
posts : (listof post?)
```

Exercise. Write the new structure definition for blogs.

Then, we'll make a function that allows our application to initialize the blog:

```
; initialize-blog! : path? -> blog
; Reads a blog from a path, if not present, returns default
(define (initialize-blog! home)
  (local [(define (log-missing-exn-handler exn)
            (make-blog
              (path->string home)
              (list (make-post "First Post"
                              "This is my first post"
                              (list "First comment!"))
                    (make-post "Second Post"
                              "This is another post")))]
```

```

                                (list))))))
      (define the-blog
        (with-handlers ([exn? log-missing-exn-handler])
          (with-input-from-file home read))))
      (set-blog-home! the-blog (path->string home))
      the-blog))

```

`initialize-blog!` takes a path and tries to `read` from it. If the path contains a `blog` structure, then `read` will parse it, because `blogs` are `#:prefab`. If there is no file at the path, or if the file has some spurious data, then `read` or `with-input-from-file` will throw an exception. `with-handlers` provides an exception handler that will return the default `blog` structure for all kinds of errors.

After `the-blog` is bound to the newly read (or default) structure, we set the `home` to the correct path. (Notice that we need to convert the path into a string. Why didn't we just make the `blog` structure contain paths? Answer: They can't be used with `read` and `write`.)

Next, we will need to write a function to save the model to the disk.

```

; save-blog! : blog -> void
; Saves the contents of a blog to its home
(define (save-blog! a-blog)
  (local [(define (write-to-blog)
            (write a-blog))]
    (without-output-to-file (blog-home a-blog)
      write-to-blog
      #:exists 'replace)))

```

`save-blog!` writes the model into its `home`. It provides `without-output-to-file` with an `#:exists` flag that tells it to replace the file contents if the file at `blog-home` exists.

This function can now be used to save the `blog` structure whenever we modify it. Since we only ever modify the `blog` structure in the model, we only need to update `blog-insert-post!` and `post-insert-comment!`.

Exercise. Change `blog-insert-post!` and `post-insert-comment!` to call `save-blog!`.

You may have had a problem when trying to update `post-insert-comment!`. It needs to call `save-blog!` with the `blog` structure. But, it wasn't passed the `blog` as an argument. We'll need to add that argument and change the application appropriately. While we're at it, let's change `blog-insert-post!` to accept the contents of the `post` structure, rather the structure itself, to better abstract the model interface:

```
blog-insert-post! : (blog? string? string? . -> . void)
```

```
post-insert-comment! : (blog? post? string? . -> . void)
```

Exercise. Write the new definitions of `blog-insert-post!` and `post-insert-comment!`. (Remember to call `save-blog!`.)

In our last iteration of our model, we used `(provide (all-defined-out))` to expose all of the model's definitions. But we often want to hide things like private functions and internal data structures from others. We'll do that here by using a form of `provide` that explicitly names the exposed definitions.

For example, if we wanted to limit the exposed functions to `blog-insert-post!` and `post-insert-comment!`, we can do this:

```
(provide blog-insert-post!
        post-insert-comment!)
```

Of course, this set of functions is too minimal! Let's change the `provide` line in the model to:

```
(provide blog? blog-posts
        post? post-title post-body post-comments
        initialize-blog!
        blog-insert-post! post-insert-comment!)
```

which captures the essential interactions we do with a blog.

The last step is to change the application. We need to call `initialize-blog!` to read in the blog structure, and we need to pass the blog value that is returned around the application, because there is no longer a `BLOG` export.

First, change `start` to call `initialize-blog!` with a path in our home directory:

```
(define (start request)
  (render-blog-page
   (initialize-blog!
    (build-path (current-directory)
                "the-blog-data.db")))
  request))
```

Exercise. Thread the `blog` structure through the application appropriately to give `blog-insert-post!` and `post-insert-comment!` the correct values. (You'll also need to

change how `render-blog-page` adds new posts.)

Our model is now:

```
#lang racket

; A blog is a (make-blog home posts)
; where home is a string, posts is a (listof post)
(define-struct blog (home posts) #:mutable #:prefab)

; and post is a (make-post blog title body comments)
; where title is a string, body is a string,
; and comments is a (listof string)
(define-struct post (title body comments) #:mutable #:prefab)

; initialize-blog! : path? -> blog
; Reads a blog from a path, if not present, returns default
(define (initialize-blog! home)
  (local [(define (log-missing-exn-handler exn)
            (make-blog
             (path->string home)
             (list (make-post "First Post"
                              "This is my first post"
                              (list "First comment!"))
                   (make-post "Second Post"
                              "This is another post"
                              (list))))))
          (define the-blog
            (with-handlers ([exn? log-missing-exn-handler])
              (with-input-from-file home read))))]
    (set-blog-home! the-blog (path->string home))
    the-blog))

; save-blog! : blog -> void
; Saves the contents of a blog to its home
(define (save-blog! a-blog)
  (local [(define (write-to-blog)
            (write a-blog))]
          (with-output-to-file (blog-home a-blog)
            write-to-blog
            #:exists 'replace)))

; blog-insert-post!: blog string string -> void
```

```

; Consumes a blog and a post, adds the post at the top of the blog.
(define (blog-insert-post! a-blog title body)
  (set-blog-posts!
   a-blog
   (cons (make-post title body empty) (blog-posts a-blog)))
  (save-blog! a-blog))

; post-insert-comment!: blog post string -> void
; Consumes a blog, a post and a comment string. As a side-effect,
; adds the comment to the bottom of the post's list of comments.
(define (post-insert-comment! a-blog a-post a-comment)
  (set-post-comments!
   a-post
   (append (post-comments a-post) (list a-comment))))
  (save-blog! a-blog))

(provide blog? blog-posts
         post? post-title post-body post-comments
         initialize-blog!
         blog-insert-post! post-insert-comment!)

```

And our application is:

```

#lang web-server/insta

(require "model-2.rkt")

; start: request -> html-response
; Consumes a request and produces a page that displays
; all of the web content.
(define (start request)
  (render-blog-page
   (initialize-blog!
    (build-path (current-directory)
                 "the-blog-data.db"))
   request))

; render-blog-page: blog request -> html-response
; Produces an html-response page of the content of the
; blog.
(define (render-blog-page a-blog request)
  (local [(define (response-generator make-url)
            `(html (head (title "My Blog"))
                  (body
                   (h1 "My Blog")
                   ,(render-posts a-blog make-url))

```

```

        (form ((action
                ,(make-url insert-post-handler)))
              (input ((name "title")))
              (input ((name "body")))
              (input ((type "submit")))))

(define (insert-post-handler request)
  (define bindings (request-bindings request))
  (blog-insert-post!
   a-blog
   (extract-binding/single 'title bindings)
   (extract-binding/single 'body bindings))
  (render-blog-page a-blog (redirect/get)))

(send/suspend/dispatch response-generator))

; render-post-detail-page: post request -> html-response
; Consumes a post and produces a detail page of the post.
; The user will be able to either insert new comments
; or go back to render-blog-page.
(define (render-post-detail-page a-blog a-post request)
  (local [(define (response-generator make-url)
            '(html (head (title "Post Details"))
                   (body
                     (h1 "Post Details")
                     (h2 ,(post-title a-post))
                     (p ,(post-body a-post))
                     ,(render-as-itemized-list
                       (post-comments a-post))
                     (form ((action
                             ,(make-url insert-comment-handler)))
                           (input ((name "comment")))
                           (input ((type "submit"))))
                     (a ((href ,(make-url back-handler)))
                        "Back to the blog"))))])

(define (parse-comment bindings)
  (extract-binding/single 'comment bindings))

(define (insert-comment-handler request)
  (render-confirm-add-comment-page
   a-blog
   (parse-comment (request-bindings request))
   a-post
   request))

```

```

        (define (back-handler request)
          (render-blog-page a-blog request))]

      (send/suspend/dispatch response-generator)))

; render-confirm-add-comment-page :
; blog comment post request -> html-response
; Consumes a comment that we intend to add to a post, as well
; as the request. If the user follows through, adds a comment
; and goes back to the display page. Otherwise, goes back to
; the detail page of the post.
(define (render-confirm-add-comment-page a-blog a-comment
                                          a-post request)
  (local [(define (response-generator make-url)
            `(html (head (title "Add a Comment"))
                  (body
                     (h1 "Add a Comment")
                     "The comment: " (div (p ,a-comment))
                     "will be added to "
                     (div ,(post-title a-post))

                     (p (a ((href ,(make-url yes-handler)))
                          "Yes, add the comment.))
                     (p (a ((href ,(make-url cancel-handler)))
                          "No, I changed my mind!"))))))

          (define (yes-handler request)
            (post-insert-comment! a-blog a-post a-comment)
            (render-post-detail-page a-blog a-post (redirect/get)))

          (define (cancel-handler request)
            (render-post-detail-page a-blog a-post request))]

    (send/suspend/dispatch response-generator)))

; render-post: post (handler -> string) -> html-response
; Consumes a post, produces an html-response fragment of the post.
; The fragment contains a link to show a detailed view of the post.
(define (render-post a-blog a-post make-url)
  (local [(define (view-post-handler request)
            (render-post-detail-page a-blog a-post request))]
    `(div ((class "post"))
          (a ((href ,(make-url view-post-handler)))
              ,(post-title a-post))
          (p ,(post-body a-post))
          (div ,(number->string (length (post-comments a-post))))))

```

```

        " comment(s)"))))

; render-posts: blog (handler -> string) -> html-response
; Consumes a make-url, produces an html-response fragment
; of all its posts.
(define (render-posts a-blog make-url)
  (local [(define (render-post/make-url a-post)
            (render-post a-blog a-post make-url))]
    `(div ((class "posts")
           ,(map render-post/make-url (blog-posts a-blog))))))

; render-as-itemized-list: (listof html-response) -> html-response
; Consumes a list of items, and produces a rendering as
; an unordered list.
(define (render-as-itemized-list fragments)
  `(ul ,(map render-as-item fragments)))

; render-as-item: html-response -> html-response
; Consumes an html-response, and produces a rendering
; as a list item.
(define (render-as-item a-fragment)
  `(li ,a-fragment))

```

This approach to persistence can work surprisingly well for simple applications. But as our application's needs grow, we will have to deal with concurrency issues, the lack of a simply query language over our data model, etc. So, in the next section, we'll talk about how to use an SQL database to store our blog model.

15 Using an SQL database

Our next task is to employ an SQL database for the blog model. We'll be using SQLite with the `(planet jaymccarthy/sqlite:4)` PLaneT package. We add the following to the top of our model:

```
(require (prefix-in sqlite: (planet jaymccarthy/sqlite:4)))
```

We now have the following bindings:

```
sqlite:db? : (any/c . -> . boolean?)
```

```
sqlite:open : (path? . -> . sqlite:db?)
```

```
sqlite:exec/ignore : (sqlite:db? string? . -> . void)
```

```
sqlite:select : (sqlite:db? string? . -> . (listof (vectorof (or/c integer? number? string? by
```

```
sqlite:insert : (sqlite:db? string? . -> . integer?)
```

The first thing we should do is decide on the relational structure of our model. We will use the following tables:

```
CREATE TABLE posts (id INTEGER PRIMARY KEY, title TEXT, body TEXT)
CREATE TABLE comments (pid INTEGER, content TEXT)
```

Each post will have an identifier, a title, and a body. This is the same as our old Racket structure, except we've added the identifier. (Actually, there was always an identifier—the memory pointer—but now we have to make it explicit in the database.)

Each comment is tied to a post by the post's identifier and has textual content. We could have chosen to serialize comments with `write` and add a new TEXT column to the posts table to store the value. By adding a new comments table, we are more in accord with the relational style.

A `blog` structure will simply be a container for the database handle:

```
(struct blog (db)
  #:extra-constructor-name make-blog)
  db : sqlite:db?
```

Exercise. Write the `blog` structure definition. (It does not need to be mutable or serializable.)

We can now write the code to initialize a `blog` structure:

```
; initialize-blog! : path? -> blog?
; Sets up a blog database (if it doesn't exist)
(define (initialize-blog! home)
  (define db (sqlite:open home))
  (define the-blog (make-blog db))
  (with-handlers ([exn? void])
    (sqlite:exec/ignore db
      (string-append
        "CREATE TABLE posts "
        "(id INTEGER PRIMARY KEY,"
        "title TEXT, body TEXT)"))
    (blog-insert-post!
      the-blog "First Post" "This is my first post")
    (blog-insert-post!
      the-blog "Second Post" "This is another post")
    (sqlite:exec/ignore
      db "CREATE TABLE comments (pid INTEGER, content TEXT)")
    (post-insert-comment!
      the-blog (first (blog-posts the-blog))
      "First comment!"))
  the-blog)
```

`sqlite:open` will create a database if one does not already exist at the `home` path. But, we still need to initialize the database with the table definitions and initial data.

We used `blog-insert-post!` and `post-insert-comment!` to initialize the database. Let's see their implementation:

```
; blog-insert-post!: blog? string? string? -> void
; Consumes a blog and a post, adds the post at the top of the blog.
(define (blog-insert-post! a-blog title body)
  (sqlite:insert
    (blog-db a-blog)
    (format "INSERT INTO posts (title, body) VALUES ('~a', '~a')"
      title body)))

; post-insert-comment!: blog? post string -> void
; Consumes a blog, a post and a comment string. As a side-effect,
; adds the comment to the bottom of the post's list of comments.
(define (post-insert-comment! a-blog p a-comment)
  (sqlite:insert
```

```
(blog-db a-blog)
(format
 "INSERT INTO comments (pid, content) VALUES ('~a', '~a')"
 (post-id p) a-comment)))
```

Exercise. Find the security hole common to these two functions.

A user could submit a post with a title like, "null', 'null') and INSERT INTO accounts (username, password) VALUES ('ur', 'hacked" and get our simple `sqlite:insert` to make two INSERTs instead of one.

This is called an SQL injection attack. It can be resolved by using prepared statements that let SQLite do the proper quoting for us. Refer to the SQLite package documentation for usage.

In `post-insert-comment!`, we used `post-id`, but we have not yet defined the new `post` structure. It *seems* like a `post` should be represented by an integer id, because the post table contains an integer as the identifying value.

However, we cannot tell from this structure what blog this posts belongs to, and therefore, what database; so, we could not extract the title or body values, since we do not know what to query. Therefore, we should associate the blog with each post:

```
(struct post (blog id)
 #:extra-constructor-name make-post)
blog : blog?
id : integer?
```

Exercise. Write the structure definition for posts.

The only function that creates posts is `blog-posts`:

```
; blog-posts : blog -> (listof post?)
; Queries for the post ids
(define (blog-posts a-blog)
  (local [(define (row->post a-row)
            (make-post
             a-blog
             (vector-ref a-row 0)))]
    (define rows (sqlite:select
```

```

                (blog-db a-blog)
                "SELECT id FROM posts"))]
(cond [(empty? rows)
      empty]
      [else
       (map row->post (rest rows))]))))

```

`sqlite:select` returns a list of vectors. The first element of the list is the name of the columns. Each vector has one element for each column. Each element is a string representation of the value.

At this point we can write the functions that operate on posts:

```

; post-title : post -> string?
; Queries for the title
(define (post-title a-post)
  (vector-ref
   (second
    (sqlite:select
     (blog-db (post-blog a-post))
     (format
      "SELECT title FROM posts WHERE id = '~a'"
      (post-id a-post)))))
  0))

```

Exercise. Write the definition of `post-body`.

Exercise. Write the definition of `post-comments`. (Hint: Use `blog-posts` as a template, not `post-title`.)

The only change that we need to make to the application is to require the new model. The interface is exactly the same!

Our model is now:

```

#lang racket
(require (prefix-in sqlite: (planet jaymccarthy/sqlite:4)))

; A blog is a (make-blog db)
; where db is an sqlite database handle
(define-struct blog (db))

```

```

; A post is a (make-post blog id)
; where blog is a blog and id is an integer?
(define-struct post (blog id))

; initialize-blog! : path? -> blog?
; Sets up a blog database (if it doesn't exist)
(define (initialize-blog! home)
  (define db (sqlite:open home))
  (define the-blog (make-blog db))
  (with-handlers ([exn? void])
    (sqlite:exec/ignore db
      (string-append
        "CREATE TABLE posts "
        "(id INTEGER PRIMARY KEY,"
        "title TEXT, body TEXT)"))
    (blog-insert-post!
      the-blog "First Post" "This is my first post")
    (blog-insert-post!
      the-blog "Second Post" "This is another post")
    (sqlite:exec/ignore
      db "CREATE TABLE comments (pid INTEGER, content TEXT)")
    (post-insert-comment!
      the-blog (first (blog-posts the-blog))
      "First comment!"))
  the-blog)

; blog-posts : blog -> (listof post?)
; Queries for the post ids
(define (blog-posts a-blog)
  (local [(define (row->post a-row)
            (make-post
              a-blog
              (vector-ref a-row 0)))
          (define rows (sqlite:select
                        (blog-db a-blog)
                        "SELECT id FROM posts"))]
    (cond [(empty? rows)
           empty]
          [else
           (map row->post (rest rows))])))

; post-title : post -> string?
; Queries for the title
(define (post-title a-post)
  (vector-ref

```

```

(second
  (sqlite:select
    (blog-db (post-blog a-post))
    (format "SELECT title FROM posts WHERE id = '~a'"
            (post-id a-post))))
0))

; post-body : post -> string?
; Queries for the body
(define (post-body p)
  (vector-ref
    (second
      (sqlite:select
        (blog-db (post-blog p))
        (format "SELECT body FROM posts WHERE id = '~a'"
                (post-id p))))
    0))

; post-comments : post -> (listof string?)
; Queries for the comments
(define (post-comments p)
  (local [(define (row->comment a-row)
            (vector-ref a-row 0))
          (define rows
            (sqlite:select
              (blog-db (post-blog p))
              (format
                "SELECT content FROM comments WHERE pid = '~a'"
                (post-id p))))])
    (cond
      [(empty? rows) empty]
      [else (map row->comment (rest rows))])))

; blog-insert-post!: blog? string? string? -> void
; Consumes a blog and a post, adds the post at the top of the blog.
(define (blog-insert-post! a-blog title body)
  (sqlite:insert
    (blog-db a-blog)
    (format "INSERT INTO posts (title, body) VALUES (~a, '~a'"
            title body)))

; post-insert-comment!: blog? post string -> void
; Consumes a blog, a post and a comment string. As a side-effect,
; adds the comment to the bottom of the post's list of comments.
(define (post-insert-comment! a-blog p a-comment)
  (sqlite:insert

```

```
(blog-db a-blog)
(format
 "INSERT INTO comments (pid, content) VALUES ('~a', '~a')"
 (post-id p) a-comment)))

(provide blog? blog-posts
 post? post-title post-body post-comments
 initialize-blog!
 blog-insert-post! post-insert-comment!)
```

And our application is:

```
#lang web-server/insta

(require "model-3.rkt")

....
```

16 Leaving DrRacket

So far, to run our application, we've been pressing Run in DrRacket. If we were to actually deploy an application, we'd need to do this differently.

The simplest way to do this is to use `web-server/servlet-env`.

First, change the first lines in your application from

```
#lang web-server/insta
```

to

```
#lang racket

(require web-server/servlet)
(provide/contract (start (request? . -> . response/c)))
```

Second, add the following at the bottom of your application:

```
(require web-server/servlet-env)
(serve/servlet start
  #:launch-browser? #f
  #:quit? #f
  #:listen-ip #f
  #:port 8000
  #:extra-files-paths
  (list (build-path your-path-here "htdocs"))
  #:servlet-path
  "/servlets/APPLICATION.rkt")
```

You can change the value of the `#:port` parameter to use a different port.

`#:listen-ip` is set to `#f` so that the server will listen on *all* available IPs.

You should change `your-path-here` to be the path to the parent of your `htdocs` directory.

You should change `"APPLICATION.rkt"` to be the name of your application.

Third, to run your server, you can either press Run in DrRacket, or type

```
racket -t <file.rkt>
```

(With your own file name, of course.) Both of these will start a Web server for your application.

`serve/servlet` takes other options and there are more advanced ways of starting the Web Server, but you'll have to refer to the Racket Web Server Reference Manual for details.

17 Using HTTPS

This final task that we'll cover is using the server in HTTPS mode. This requires an SSL certificate and private key. This is very platform specific, but we will provide the details for using OpenSSL on UNIX:

```
openssl genrsa -des3 -out private-key.pem 1024
```

This will generate a new private key, but it will have a passphrase on it. You can remove this via:

```
openssl rsa -in private-key.pem -out private-key.pem
```

```
chmod 400 private-key.pem
```

Now, we generate a self-signed certificate:

```
openssl req -new -x509 -nodes -sha1 -days 365 -key private-key.pem  
> server-cert.pem
```

(Each certificate authority has different instructions for generating certificate signing requests.)

We can now start the server with:

```
plt-web-server -ssl
```

The Web Server will start on port 443 (which can be overridden with the `-p` option) using the "private-key.pem" and "server-cert.pem" we've created.

18 Moving Forward

As you move forward on your own applications, you may find many useful packages on PLaneT. There are interfaces to other databases. Many tools for generating HTML, XML, and Javascript output. Etc. There is also an active community of users on the Racket mailing list. We welcome new users!