CHAPTER 1

TinyBlog: A Simple Teapot Web Interface

1.1 Previous Week Solution

You can load the solution of the previous week using the following snippet:

```
Gofer new
    smalltalkhubUser: 'PharoMooc' project: 'TinyBlog';
    package: 'ConfigurationOfTinyBlog';
    load.
#ConfigurationOfTinyBlog asClass loadWeek2Correction
```

After a loading a package, you shall run the unit tests to ensure that the loaded code is correctly working. Open the TestRunner (World menu > Test Runner), find the 'TinyBlog-Tests' package and run all unit tests of the `TB-BlogTest` class by clicking on the 'Run Selected' button. All tests should be green. One alternative is to press the green icon on the side of the class `TB-BlogTest`.

Open a code browser to look at the code of both classes `TBBlog` and `TBBlogTest`. You can now complete you own implementation if needed. Before continuing, do not forget to commit a new version in your repository on Smalltalkhub or SS3 if you modified your code.

1.2 A Web Interface for TinyBlog with Teapot

This week, we will create a first simple web interface for TinyBlog with Teapot (http://smalltalkhub.com/#!/~zeroflag/Teapot). We will implement a more complete version with Seaside next week.

1.3 **The TBTeapotWebApp Class**

Create a new class named TBTeapotWebApp:

```
Object subclass: #TBTeapotWebApp
   instanceVariableNames: 'teapot'
   classVariableNames: 'Server'
   package: 'TinyBlog-Teapot'
```

The variable teapot will refer to a little Teapot HTTP server. Here we use a different implementation of the Singleton Design Pattern by using a class variable named Server. We use a Singleton to avoid to have two servers listening to the same port.

Add the instance method initialize to initialize the instance variable teapot:

```
TBTeapotWebApp >> initialize
   super initialize.
   teapot := Teapot configure: {
      #port -> 8081.
      #debugMode -> true }.
```

The Home Page

The homePage method defined inside in the 'html' protocol should return the HTML code of the home page of our web application as a String. Let's start with a simple version:

```
TBTeapotWebApp >> homePage
   ^ '<html><body><h1>TinyBlog Web App</h1></body></html>'
```

Declare Routes

Add a start method to declare to the teapot object the URLs it must answer to. So far, we only add the route / accessed via a GET Http method:
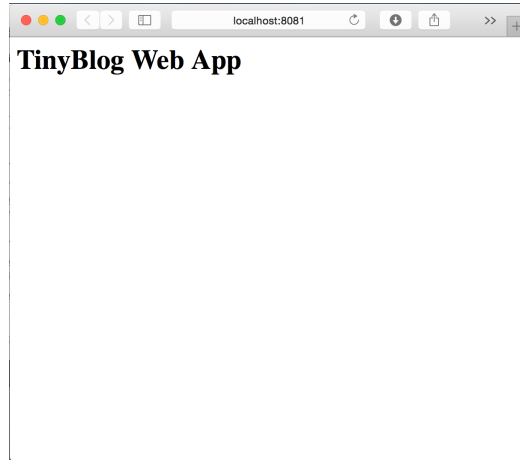
```
TBTeapotWebApp >> start
   teapot
      GET: '/' -> [ self homePage ];
      start
```

Stop the Application

Add also a method stop to stop the application.

```
TBTeapotWebApp >> stop
   teapot stop
```

Figure 1.1 A first page served by our application.

Starting the application

Add two class-side methods start and stop to start and stop the web application in the protocol 'start/stop'. These two methods use the class variable Server to implement a Singleton.

```
TBTeapotWebApp class >> start
    Server ifNil: [ Server := self new start ]

TBTeapotWebApp class >> stop
    Server ifNotNil: [ Server stop. Server := nil ]
```

1.4 Test your Application

Execute the following snippet to start your application:

```
TBTeapotWebApp start
```

In a web browser, try to access the application with this URL: http://localhost:8081/. You should see the text: "TinyBlog Web App" as in Figure 1.1.

1.5 Display the List of All Visible Posts

Modify now the code of the homePage method to display the list of all visible posts in the current blog. Remember these posts can be obtained with: TBBlog current allVisibleBlogPosts. We implement that functionality by adding three methods and modifying the homePage method.

```
TBTeapotWebApp >> allPosts
    ^ TBBlog current allVisibleBlogPosts
```

Since we need to generate a long String that contains the HTML code of the home page, we decided to use a Stream in the `homePage` method. We also factored out the HTML generation of the HTML page header and footer in two different methods: `renderPageHeaderOn:` and `renderPageFooterOn:`.

```
TBTeapotWebApp >> homePage
    ^ String streamContents: [ :s |
        self renderPageHeaderOn: s.
        s << '<h1>TinyBlog Web App</h1>'.
        s << '<ul>'.
        self allPosts do: [ :aPost |
            s << ('<li>', aPost title, '</li>') ].
        s << '</ul>'.
        self renderPageFooterOn: s.
    ]
```

Note that the message `<<` is a different name for the message `nextPutAll:` that adds a collection of elements to a stream.

```
TBTeapotWebApp >> renderPageHeaderOn: aStream
    aStream << '<html><body>'
```

```
TBTeapotWebApp >> renderPageFooterOn: aStream
    aStream << '</body></html>'
```

Test your application in a web browser, you should now see a list of post titles as in Figure 1.2. If this is not the case make sure that your blog contains some post. You can use the message `createDemoPosts` to add some generic blog posts.

```
TBBlog createDemoPosts
```

1.6 Details of a Post

Add a New Web Page

We would like that the following URL http://localhost:8081/post/1 displays the whole post number 1.

To start, let us think about the worst case and define what should happen in case of errors. We define the method `errorPage`.

```
TBTeapotWebApp >> errorPage
    ^ String streamContents: [ :s |
        self renderPageHeaderOn: s.
        s << '<p>Oops, an error occurred</p>'.
        self renderPageFooterOn: s ]
```
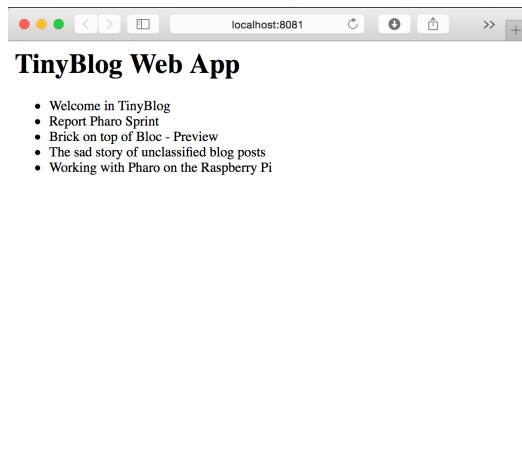
Figure 1.2   Showing post titles.

Teapot supports patterns such as '<id>' in route definitions. The corresponding value of '<id>' in the incoming URL is then accessible through the `re-quest` object passed as a block parameter. Now we modify the `start` method and introduce a new route into the application to display the content of a post.

```
TBTeapotWebApp >> start
   teapot
      GET: '/' -> [ self homePage ];
      GET: '/post/<id>' -> [ :request | self pageForPostNumber:
    (request at: #id) asNumber ];
      start
```

We now add a new method named `pageForPostNumber:` displaying the whole content of a post:

```
TBTeapotWebApp >> pageForPostNumber: aPostNumber
   | currentPost |
   currentPost := self allPosts at: aPostNumber ifAbsent: [ ^ self
    errorPage ].
   ^ String streamContents: [ :s |
       self renderPageHeaderOn: s.
       s << ('<h1>', currentPost title, '</h1>').
       s << ('<h3>', currentPost date mmddyyyy, '</h3>').
       s << ('<p> Category: ', currentPost category, '</p>').
       s << ('<p>', currentPost text, '</p>').
       self renderPageFooterOn: s ]
```

You should now restart the server before testing your application with the following URL: http://localhost:8081/post/1

The parameter of `pageForPostNumber:` is the integer passed in the URL and it is used as an index to retrieve the post to display in the collection of posts. Obviously, this is a fragile solution because if the order of the posts changes in the collection, a given URL will not display the same post as before.

Add Links to Posts

Modify the `homePage` method so that post titles in the list will be links to their own web page.

```
TBTeapotWebApp >> homePage
   ^ String streamContents: [ :s |
       self renderPageHeaderOn: s.
       s << '<h1>TinyBlog Web App</h1>'.
       s << '<ul>'.
       self allPosts withIndexDo: [ :aPost :index |
          s << '<li>';
            << ('<a href="/post/', index asString, '">');
            << aPost title ;
            << '</a></li>' ].
       s << '</ul>'.
       self renderPageFooterOn: s.
     ]
```

Now, the home page of the application displays a list of clickable post titles and if you click on a post title, you will see the content of this post.

1.7 Possible Extensions

This application is a really simple and pedagogical example through which you manipulate collections, streams, etc.

You can improve this web application and implement new functionalities such as:

- adding a return to home page link on a post page,

- adding a new page that displays the list of all post categories,

- adding a new page that displays all posts that belong to one specific category,

- adding CSS styles to make this web application more appealing.