

TinyBlog: Desarrolla tu Primera Aplicación web con Pharo

Olivier Auverlot, Stéphane Ducasse y Luc Fabresse

1 de abril de 2020

Copyright 2017 de Olivier Auverlot, Stéphane Ducasse y Luc Fabresse.

El contenido de este libro está protegido por la licencia Creative Commons Attribution ShareAlike 3.0 Unported.

Eres libre:

- Compartir: copiar, distribuir y transmitir la obra,
- Remezclar:
adaptar la obra,

Bajo las siguientes condiciones:

Atribución. Debe atribuir la obra en la forma especificada por el autor o licenciatario (pero no de ninguna manera que sugiera que lo respaldan a usted o su uso del trabajo).

Compartir por igual. Si altera, transforma o construye a partir de este trabajo, puede distribuir el trabajo resultante solo bajo la misma licencia, similar o compatible.

Para cualquier reutilización o distribución, debe dejar en claro a los demás los términos de la licencia de este trabajo. La mejor manera de hacerlo es con un enlace a esta página web: <http://creativecommons.org/licenses/by-sa/3.0/>

Se puede renunciar a cualquiera de las condiciones anteriores si obtiene el permiso del titular de los derechos de autor. Nada en esta licencia perjudica o restringe los derechos morales del autor.



Su trato justo y otros derechos no se ven afectados de ninguna manera por lo anterior. Este es un resumen legible por humanos del Código Legal (la licencia completa): <http://creativecommons.org/licenses/by-sa/3.0/legalcode>

Contenido

Ilustraciones	IV
1 Acerca de este libro	1
1.1 Estructura 1.2	1
Instalación de Pharo 1.3	2
Reglas de nomenclatura 1.4	2
Recursos	3
2 Aplicación TinyBlog: modelo principal	5
2.1 TBPost Clase	5
2.2 Visibilidad de la publicación	6
2.3 Inicialización 2.4	7
Métodos de creación de publicaciones	7
2.5 Creación de una publicación 2:6	8
Adición de algunas pruebas unitarias	8
2.7 Consultas posteriores	9
2.8 Conclusión	10
3 TinyBlog: Ampliación y prueba del modelo	11
3.1 Clase TBBlog 3.2 Solo un objeto de blog	11
.	12
3.3 Prueba del modelo 3.4 Una primera prueba 3.5 Incremento de la cobertura de prueba	12
3.6 Otras funcionalidades 3.7 Datos de prueba 3.8 Posibles extensiones	14
3.9 Conclusión	16
.	17
4 Persistencia de datos usando Voyage y Mongo	19
4.1 Configurar Voyage para guardar objetos TBBlog	19
4.2 Guardar un blog	21
4.3 Revisión de pruebas unitarias	21
4.4 Consulta de la base de datos 4.5	22
Si guardariamos publicaciones [Discusión]	22
4.6 Configurar una base de datos Mongo externa [Opcional]	23
4.7 Conclusión	25

Contenido

5 Primeros pasos con Seaside 5.1	27
Iniciando Seaside 5.2 Bootstrap para	28
Seaside 5.3 Definir el punto de entrada	28
de nuestra aplicación 5.4 Primer renderizado	29
simple	31
5.5 Arquitectura	32
5.6 Conclusión	33
6 componentes web para TinyBlog	35
6.1 Componentes visuales	35
6.2 Uso del componente TBScreenComponent	37
6.3 Patrón de Definición de Componente	37
6.4 Poblando el Blog	38
6.5 Definición de TBHeaderComponent	38
6.6 Uso de TBHeaderComponent	38
6.7 Relación de componentes compuestos	39
6.8 Representar un	39
encabezado 6.9 Lista de	41
publicaciones 6.10 El PostComponent	42
6.11 Mostrar	43
publicaciones 6.12	44
Depuración de errores 6.13 Mostrar la lista de publicaciones con Bootstrap	44
6.14 Creación de instancias de componentes en renderContentOn:	45
6.15 Conclusión	46
7 Gestión de categorías	47
7.1 Visualización de publicaciones por categoría	47
7.2 Representación de categorías	49
7.3 Actualización de la lista de	50
publicaciones 7.4 Aspecto y diseño	50
7.5 Código modular con métodos	52
pequeños 7.6 Conclusión	54
8 Autenticación y Sesión	55
8.1 Un componente de administración simple (v1)	56
8.2 Adición del botón 'admin'	56
8.3 Revisión del encabezado	58
8.4 Activación del botón Admin	58
8.5 Adición del botón 'disconnect'	59
8.6 Ventana modal para la autenticación	60
8.7 Representación del componente de autenticación	62
8.8 Integración del componente de autenticación	62
8.9 Administrar ingenuamente los inicios de sesión	63
8.10 Gestión de errores 8.11	64
Modelado del administrador 8.12	64
Administrador del blog	65

Contenido

8.13 Configuración de un nuevo administrador	67	68
8.14 Integración de la información del administrador	68	
8.15 Almacenamiento del administrador en la sesión actual	68	
8.16 Definición y uso de una sesión específica	68	
8.17 Almacenamiento del administrador actual	70	
Navegación simplificada	70	
8.19 Gestión de la desconexión	70	71
8.20 Navegación simplificada a la parte pública	71	
Conclusión		
9 Interfaz web de administración y generación automática de formularios		73
9.1 Descripción de los datos del dominio	73	
9.2 Descripción de la publicación	75	
9.3 Creación automática de componentes	76	
9.4 Creación de un informe posterior	76	
9.5 Integración de AdminComponent con PostsReport	77	
9.6 Filtrar columnas	78	
9.7 Mejoras de informes	79	80
9.8 Administración posterior	81	
9.9 Adición posterior		
9.10 Implementación de acciones CRUD		
9.11 Adición de publicaciones	81	81
9.12 Actualización de publicaciones	84	85
9.13 Mejor aspecto de la forma	86	
9.14 Conclusión		
10 Cargar el código del capítulo		
10.1 Capítulo 3: Extender y probar el modelo	87	87
Capítulo 4: Persistencia de datos usando Voyage y Mongo	88	
10.3 Capítulo 5: Primeros Pasos con Seaside	88	
10.4 Capítulo 6: Componentes Web para TinyBlog	88	
10.5 Capítulo 7: Gestión de categorías	88	
10.6 Capítulo 8: Autenticación y sesión	89	
10.7 Capítulo 9: Interfaz web de administración y generación automática de formularios	89	
10.8 Última versión de TinyBlog	89	
11 Guarda tu código		91

Ilustraciones

1-1 La aplicación TinyBlog	2
2-1 TBPost: una clase realmente básica que maneja principalmente datos.	5
2-2 Inspector en una instancia de TBPost.	8
3-1 TBBlog: una clase simple que contiene publicaciones.	11
5-1 Inicio del servidor Seaside.	27
5-2 Correr junto al mar.	28
5-3 Exploración de la biblioteca Bootstrap de Seaside.	29
5-4 Un elemento Bootstrap y su código.	30
5-5 TinyBlog es una aplicación Seaside registrada.	31
5-6 Una primera página web de Seaside.	31
5-7 Componentes principales de TinyBlog (vista pública).	32
5-8 Arquitectura de TinyBlog	33
6-1 Arquitectura Componente de la Vista Pública (opuesta a la vista de administración).	35 . . .
6-2 Componentes visuales de TinyBlog.	36
6-3 ApplicationRootComponent utiliza temporalmente un componente de pantalla que contiene un HeaderComponent.	37 . . .
6-4 Primera representación visual de TBScreenComponent.	38 . . .
6-5 TinyBlog con encabezado Bootstrap.	40
6-6 ApplicationRootComponent utiliza PostsListComponent.	41
6-7 TinyBlog mostrando una lista de publicaciones básicas.	42
6-8 Uso de PostComponents para mostrar cada publicación.	42
6-9 TinyBlog con una lista de publicaciones.	44
7-1 La arquitectura de los componentes de la parte pública con categorías.	47 7-2
Categorías y puestos.	51
7-3 Lista de publicaciones con un mejor diseño.	52
7-4 Interfaz de usuario pública final de TinyBlog.	54

Ilustraciones

8-1 Flujo de autenticación.	55
8-2 Enlace simple a la parte de administración.	57
8-3 Encabezado con un botón de administración.	57
8-4 Componente de administración en definición.	59 61
8-5 Componente de autenticación.	
8-6 Mensaje de error en caso de identificadores erróneos.	66
8-7 Navegación e identificación en TinyBlog.	69
9-1 Gestión de puestos.	74
9-2 Componentes de administración.	74
9-3 Informe Magritte con postes.	78
9-4 Informe de administración.	80
9-5 Publicar informe con enlaces.	82
9-6 Representación básica de un post.	83
9-7 Adición de formulario de publicación con Bootstrap.	86

CAPÍTULO 1

Sobre este libro

En este libro, lo guiaremos para desarrollar un mini proyecto: una pequeña aplicación web, llamada TinyBlog, que administra un sistema de blogs (vea su estado final en la Figura 1-1). La idea es que un visitante del sitio web pueda leer las publicaciones y que el autor de la publicación pueda conectarse al sitio web como administrador para administrar sus publicaciones (agregar, eliminar y modificar las existentes).

TinyBlog es una pequeña aplicación pedagógica que le mostrará cómo definir e implementar una aplicación web utilizando Pharo/Seaside/Mongo y marcos disponibles en Pharo como NeoJSON.

Nuestro objetivo es que pueda reutilizar y adaptar dicha infraestructura para crear sus propias aplicaciones web.

1.1 Estructura

En la primera parte, denominada "Tutorial principal", desarrollará e implementará Tiny Blog, una aplicación y su administración utilizando Pharo, el marco del servidor web de la aplicación Seaside, así como algunos otros marcos como Voyage y Magritte. La implementación con Mongo DB es opcional, pero le permite ver que Voyage es una fachada elegante para conservar sus datos dentro de Mongo.

En la segunda parte y opcional, te mostraremos algunos aspectos opcionales como la exportación de datos, el uso de Moustache o cómo exponer tu aplicación usando una API REST.

Las soluciones presentadas a veces no son las mejores. Esto se hace de esa manera para ofrecerte un margen de mejora. Nuestro objetivo no es ser exhaustivo. Presentamos una forma de desarrollar TinyBlog; sin embargo, invitamos al lector a leer más referencias, como libros o tutoriales sobre Pharo, para profundizar su experiencia y mejorar su aplicación.

Sobre este libro

Figura 1-1 La aplicación TinyBlog.

Finalmente, para ayudarlo a superar posibles errores y evitar quedarse atascado, el último capítulo describe cómo cargar el código descrito en cada capítulo.

1.2 Instalación del faro

En este tutorial, suponemos que está utilizando la imagen Pharo MOOC (actualmente es una imagen Pharo 8.0) en la que se han cargado muchos marcos y bibliotecas web: Seaside (servidor de aplicaciones web basado en componentes), Magritte (un informe de generación automática sistema basado en descripciones), Bootstrap (una biblioteca para ajustar visualmente las aplicaciones web), Voyage (un marco para guardar sus objetos en bases de datos de documentos) y algunos otros.

Puede obtener la imagen del MOOC de Pharo usando Pharo Launcher (<http://pharo.org/download>).

1.3 Reglas de nomenclatura

A continuación, prefijamos todos los nombres de clase TB (para TinyBlog). Puedes:

- elegir otro prefijo (por ejemplo TBM) para poder cargar el solución al lado del suyo. Así podrás comparar

1.4 Recursos

las dos soluciones,

- elija el mismo prefijo para fusionar las soluciones propuestas en su código. La herramienta de combinación lo ayudará a ver las diferencias y aprender de los cambios. Esta solución puede ser más compleja si implementa sus propias funcionalidades adicionales.

1.4 Recursos

Pharo tiene muchos recursos pedagógicos sólidos, así como una comunidad de usuarios súper amigable. Aquí hay una lista de recursos:

- <http://books.pharo.org> propone libros alrededor de Pharo. Pharo by Example puede ayudarte a descubrir el idioma y sus bibliotecas. Enterprise Pharo: a Web Perspective presenta otros aspectos útiles para el desarrollo web.
- <http://book.seaside.st> es uno de los libros sobre Seaside. actualmente es un der Migration como un libro de código abierto <https://github.com/SquareBracketAssociates/DynamicWebDevelopmentWithSeaside> .
- <http://mooc.pharo.org> propone un excelente Mooc con más de 90 videos que explican sintácticamente puntos así como conceptos clave de programación de objetos.
- Un canal de discordia donde muchos Pharoers intercambian información y se ayudan mutuamente está disponible aquí: <http://www.pharo.org/community>

CAPÍTULO 2

Aplicación TinyBlog: Modelo básico

En este capítulo, comenzamos a desarrollar una parte del modelo de dominio de TinyBlog.

El modelo es particularmente simple: comienza con una publicación. En el próximo capítulo agregaremos un blog que contiene una lista de publicaciones.

2.1 TBPost Clase

Comenzamos con la representación del puesto. Es súper simple como se muestra en la Figura 2-1. Está definido por la clase TBPost:

Subclase de objeto: #TBPost
nombre de variable de instancia: 'categoría de fecha de texto de título visible'
nombre de variable de clase: ''
paquete: 'TinyBlog'

Una publicación de blog se describe mediante 5 variables de instancia.

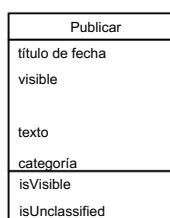


Figura 2-1 TBPost: una clase realmente básica que principalmente maneja datos.

Variable Significado título de la

título	publicación texto
texto	de la publicación
fecha	fecha de escritura
categoría	nombre de la categoría de la publicación
visible	¿la publicación es visible públicamente o no?

Todas estas variables tienen métodos de acceso correspondientes en el protocolo de "acceso". Puede utilizar una refactorización para crear automáticamente todos los métodos siguientes:

```
[ TBPost >> título título
  ^
  [ TBPost >> título: unaCadena
    título := unaCadena
  ]
  [ TBPost >> texto
    ^
    texto
  ]
  [ TBPost >> texto: unaCadena
    texto := unaCadena
  ]
  [ TBPost >> fecha
    ^
    fecha
  ]
  [ TBPost >> datos: aData
    fecha := unafecha
  ]
  [ TBPost >> visible
    ^
    visible
  ]
  [ TBPost >> visible: un booleano
    visible := aBooleano
  ]
  [ TBPost >> categoría
    ^
    categoría
  ]
  [ TBPost >> categoría: unObjeto
    categoría := unObjeto
  ]
]
```

2.2 Visibilidad de la publicación

Deberíamos agregar métodos para hacer que una publicación sea visible o no y también probar si es visible. Esos métodos se definen en el protocolo de 'acción'.

```
[ TBPost >> beVisible
  auto visible: verdadero
]
[ TBPost >> no visible
  auto visible: falso
]
```

2.3 Inicialización

El método initialize (protocolo de 'inicialización') establece la fecha en el día actual y la visibilidad en falso: el usuario debe hacer explícitamente una publicación visible. Esto le permite escribir borradores y solo publicar una publicación cuando finaliza. Por defecto, una publicación pertenece a la categoría 'Sin clasificar' que definimos a nivel de clase. Este nombre de categoría se define en el lado de la clase mediante el método unclassifiedTag .

Clase TBPost >> Etiqueta no clasificada
 ^ 'Desclasificado'

Preste atención, el método unclassifiedTag debe definirse en el lado de la clase de la clase TBPost (haga clic en el botón de clase para definirlo). Los otros métodos se definen en el lado de la instancia: significa que se aplicarán a las instancias de TBBlog .

TBPost >> inicializar súper
 inicializar. categoría
 propia: TBPost etiqueta sin clasificar. autofecha:
 fecha de hoy. yo no visible

En la solución anterior, sería mejor que el método de inicialización no codifique la referencia a la clase TBPost . Proponga una solución. La secuencia 3 de la semana 6 del Mooc puede ayudarlo a comprender por qué es mejor evitar las referencias de clase de codificación fija (consulte <http://mooc.pharo.org>).

2.4 Métodos de creación de publicaciones

En el lado de la clase, agregamos métodos de clase (es decir, métodos que se ejecutan en la clase) para facilitar la creación de publicaciones para blogs; por lo general, este tipo de métodos se agrupan en el protocolo 'creación de instancias'.

Definimos dos métodos.

Clase TBPost >> título: aTitle texto: aText
 ^ auto nuevo
 título: unTítulo;
 texto: unTexto; tú
 mismo

Clase TBPost >> título: unTítulo texto: unTexto categoría: unaCategoría
 ^ (título propio: aTitle texto: aText)
 categoría: una Categoría;
 tú mismo

Aplicación TinyBlog: Modelo básico

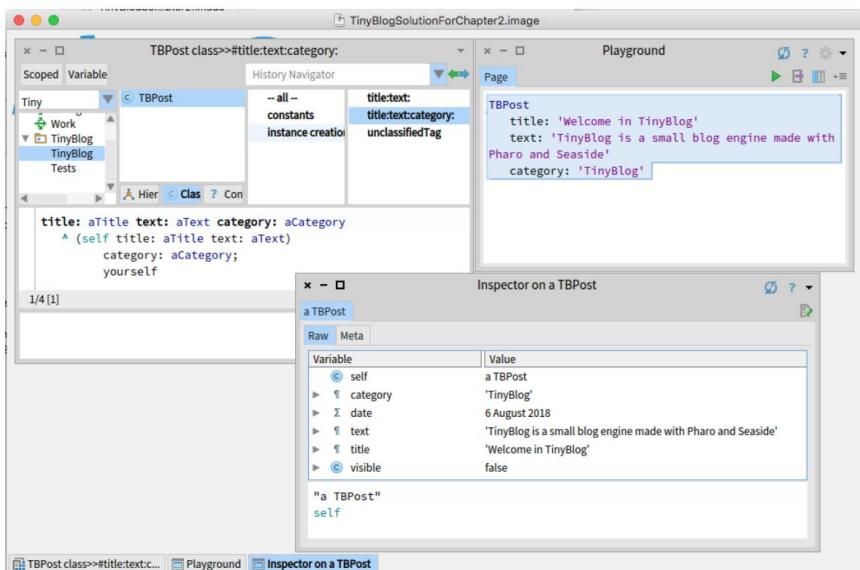


Figura 2-2 Inspector en una instancia de TBPost.

2.5 Creación de una publicación

Vamos a crear publicaciones para comprobar un poco los objetos creados. Usando las herramientas de Playgroud, ejecute la siguiente expresión:

```
TBPost
  title: 'Bienvenido a TinyBlog' text:
  'TinyBlog es un pequeño motor de blog hecho con Pharo.' categoría: 'TinyBlog'
```

Cuando inspeccione el código anterior (haga clic con el botón derecho e inspeccione), obtendrá un inspector en el objeto recién creado, como se muestra en la Figura 2-2.

2.6 Agregar algunas pruebas unitarias

Mirar manualmente los objetos no es una forma de verificar sistemáticamente que dichos objetos sigan alguna invariante esperada. Aunque el modelo es bastante simple, podemos definir algunas pruebas. En el modo de desarrollo basado en pruebas, primero escribimos prueba. Aquí preferimos dejarte definir una pequeña clase para familiarizarte con el IDE. ¡Arreglemos esto!

Definimos la clase `TBPostTest` (como subclase de la clase `TestCase`).

2.7 Consultas posteriores

```
Subclase TestCase: #TBPostTest
  nombres de variables de instancia: "
    Nombres de variables de clase: "
      paquete: 'Pruebas de TinyBlog'
```

Definamos dos pruebas.

```
TBPostTest >> testWithoutCategoryIsUnclassified
```

```
| publicar |
publicar := TBPost
  title: 'Bienvenido a TinyBlog' text:
  'TinyBlog es un pequeño motor de blog hecho con Pharo.'.
  autoafirmación: el título de la publicación es igual a: 'Bienvenido a TinyBlog'.
  autoafirmación: categoría de publicación = TBPost etiqueta no clasificada.
```

```
TBPostTest >> testPostIsCreatedCorrectamente
```

```
| publicar |
publicar := TBPost
  title: 'Bienvenido a TinyBlog' text:
  'TinyBlog es un pequeño motor de blog hecho con Pharo.' categoría: 'TinyBlog'.

  autoafirmación: el título de la publicación es igual a: 'Bienvenido a TinyBlog'.
  autoafirmación: el texto de la publicación es igual a: 'TinyBlog es un pequeño motor de blog
hecho con Pharo.' .
```

Sus pruebas deben pasar.

2.7 Consultas posteriores

En el protocolo 'testing', defina los siguientes métodos que verifican si una publicación es visible y si está clasificada o no.

```
TBPost >> esVisible
^ auto visible

TBPost >> no está clasificado
^ autocategoría = TBPost etiqueta sin clasificar
```

Realmente no es bueno codificar una referencia a la clase TBPost en el cuerpo de un método. Proponga una solución.

Además, tomemos el tiempo para actualizar nuestra prueba para aprovechar el nuevo comportamiento.

```
TBPostTest >> testWithoutCategoryIsUnclassified
```

```
| publicar |
publicar := TBPost
  title: 'Bienvenido a TinyBlog' text:
  'TinyBlog es un pequeño motor de blog hecho con Pharo.'
```

autoafirmación: el título de la publicación es igual a: 'Bienvenido a TinyBlog'.
autoafirmación: la publicación no está clasificada. negarse a sí mismo: la publicación
es visible

2.8 Conclusión

Desarrollamos una primera parte del modelo (la clase TBPost) y algunas pruebas. Recomendamos encarecidamente escribir algunas otras pruebas unitarias para asegurarse de que su modelo funcione completamente.

CAPÍTULO 3

TinyBlog: Extendiendo y probando el modelo

En este capítulo ampliamos el modelo y agregamos más pruebas. Tenga en cuenta que cuando tenga fluidez en Pharo, tenderá a escribir primero sus pruebas y luego ejecutará pruebas para codificar en el depurador. No lo hicimos porque la codificación en el depurador requiere más explicación. Puede ver una práctica de este tipo en el video de Mooc titulado Coding a Counter in the Debugger (Ver <http://mooc.pharo.org>) y lea el libro Aprendizaje de programación orientada a objetos, diseño con TDD en Pharo (<http://books.pharo.org>).

Antes de comenzar, utilice de nuevo el código del capítulo anterior o utilice la información del Capítulo ??.

3.1 Clase TBBlog

Desarrollamos la clase TBBlog que contiene publicaciones (como se muestra en la Figura 3-1). Definimos algunas pruebas unitarias.

Aquí está su definición:

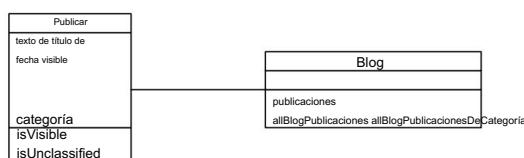


Figura 3-1 TBBlog: una clase simple que contiene publicaciones.

```

[ Subclase de objeto: #TBBlog
  nombres de variables de instancia:
  'publicaciones' nombres de variables de clase: "
  paquete: 'TinyBlog'

```

Inicializamos la variable de instancia de publicaciones en una colección vacía.

```

[ TBBlog >> inicializar super
  inicializar.
  publicaciones := OrderedCollection nuevo

```

3.2 Solo un objeto de blog

En el resto de este proyecto, asumimos que administraremos solo un blog.

Más adelante, puede agregar la posibilidad de administrar múltiples blogs, como uno por usuario de la aplicación TinyBlog. Actualmente, usamos un patrón de diseño Singleton en la clase TBBlog . Sin embargo, preste atención ya que este patrón introduce una especie de variable global en su aplicación y aporta menos modularidad a su sistema. Por lo tanto, evite hacer referencias explícitas al singleton, mejor use una variable de instancia cuyo valor primero se refiera al singleton para que luego pueda pasar otro objeto sin verse obligado a reescribir todo. No generalice lo que estamos haciendo para esta clase.

Dado que toda la gestión de un singleton es un comportamiento de clase, definimos dichos métodos en el nivel de clase de TBBlog. Definimos una variable de instancia a nivel de clase:

```

[ Instancia de
  clase TBBlogVariableNames: 'instancia única'

```

Luego definimos dos métodos para gestionar el singleton.

```

[ Clase TBBlog >> restablecer
  instancia única := nil

```

```

[ Clase TBBlog >> actual
  "responder a la instancia de TBRepository" instancia
  ^ única ifNil: [ instancia única := self new ]

```

Redefinimos el método de clase initialize para que cuando la clase se cargue en la memoria, el singleton se reinicie.

```

[ Clase TBBlog >> inicializar reinicio
  automático

```

3.3 Prueba del modelo

Ahora adoptamos un enfoque de desarrollo basado en pruebas, es decir, primero escribiremos una prueba unitaria y luego desarrollaremos la funcionalidad hasta que la prueba sea verde. Repetiremos este proceso para cada funcionalidad del modelo.

3.4 Una primera prueba

Creamos pruebas unitarias en la clase TBBlogTest que pertenece a la etiqueta TinyBlog Tests . Una etiqueta es simplemente una etiqueta para clasificar las clases dentro de un paquete (consulte el elemento de menú 'Añadir etiqueta...'). Usamos una etiqueta porque usar dos paquetes hará que este proyecto sea más complejo. Sin embargo, al implementar una aplicación real, se recomienda tener uno (o varios) paquetes de prueba separados.

Subclase de caso de prueba: #TBBlogTest

```
nombre de variable de instancia: 'publicación de blog'
primero' nombre de variable de clase: "
paquete: 'Pruebas de TinyBlog'
```

Antes de cada ejecución de prueba, el método setUp inicializa el contexto de prueba (también llamado dispositivo de prueba). Por ejemplo, borra el contenido del blog, agrega una publicación y crea otra publicación temporal que no se guarda.

Preste atención ya que tendremos que modificar dicho comportamiento en el futuro, de lo contrario, cada vez que ejecutemos la prueba, destruiremos nuestros datos. Este es un ejemplo del tipo de comportamiento insidioso que introduce un singleton.

TBBlogTest >> configurar

```
blog := TBBlog actual. blog
removeAllPosts.
```

```
primero := TB Título de la publicación: 'Un título' texto: 'Un texto' categoría: 'Primero
Categoría'.
```

```
blog escribirBlogPost: primero.
```

```
publicación := (Título de la publicación TB: 'Otro título' texto: 'Otro texto' categoría:
'Segunda categoría') beVisible
```

Como puede notar, probamos diferentes configuraciones. Las publicaciones no pertenecen a la misma categoría, una es visible y la otra no es visible.

Al final de cada prueba, se ejecuta el método tearDown y reinicia el blog.

TBBlogTest >> derribar

```
Restablecimiento de TBBlog
```

Aquí vemos uno de los límites de usar un Singleton. De hecho, si implementa un blog y luego ejecuta las pruebas, perderá todas las publicaciones que se hayan creado porque restablecimos el blog único.

Abordaremos este problema en el futuro.

Ahora desarrollaremos pruebas primero y luego implementaremos todas las funcionalidades para que sean ecológicas.

3.4 Una primera prueba

La primera prueba agrega una publicación en el blog y verifica que esta publicación se agregue de manera efectiva.

```
[ TBBlogTest >> testAddBlogPost blog
  writeBlogPost: publicación. autoafirmación:
  el tamaño del blog es igual a: 2
```

Si intenta ejecutarlo, notará que esta prueba no es verde (no pasa) porque no definimos los métodos `writeBlogPost`, `removeAll` Mensajes y tamaño. Vamos a agregarlos:

```
[ TBBlog >> removeAllPosts
  publicaciones := OrderedCollection nuevo

[ TBBlog >> escribirBlogPost: aPost
  "Agregue la publicación del blog a la lista de publicaciones". publicaciones
  agregan: una publicación

[ TBBlog >> tamaño tamaño
  ^ de las publicaciones
```

La prueba anterior ahora debería pasar (es decir, ser verde).

3.5 Aumento de la cobertura de las pruebas

También deberíamos agregar pruebas para cubrir todas las funcionalidades que introdujimos.

```
[ TBBlogTest >> testSize
  autoafirmación: el tamaño del blog es igual a: 1

[ TBBlogTest >> testRemoveAllBlogPosts blog
  removeAllPosts. autoafirmación: el tamaño del blog
  es igual a: 0
```

3.6 Otras Funcionalidades

Seguimos la forma basada en pruebas de definir métodos: Primero definimos una prueba.

Luego verificamos que esta prueba está fallando. Luego definimos el método bajo prueba y finalmente verificamos que la prueba pasa.

Todos los mensajes

Hagamos una prueba que falla:

```
[ TBBlogTest >> testAllBlogPosts blog
  writeBlogPost: publicar. autoafirmación:
  blog allBlogPosts el tamaño es igual a: 2
```

Y el código del modelo que lo hace exitoso:

```
[ TBBlog >> todas las publicaciones
  ^ de blogs
```

Su prueba debe pasar.

3.6 Otras Funcionalidades

Publicaciones visibles

Definimos una nueva prueba unitaria accediendo a blogs visibles:

```
TBBlogTest >> testAllVisibleBlogPosts blog
    writeBlogPost: publicación. autoafirmación:
        blog allVisibleBlogPosts el tamaño es igual a: 1
```

Añadimos el método correspondiente:

```
TBBlog >> publicaciones
    allVisibleBlogPosts seleccione: [ :p | p esVisible]
```

Verifique que pase la prueba.

Todas las publicaciones de una categoría

La siguiente prueba verifica que podemos acceder a todas las publicaciones de una categoría determinada. Una vez definido, debemos asegurarnos de que la prueba falló.

```
TBBlogTest >> testAllBlogPublicacionesDeCategoría
    autoafirmación: (blog allBlogPostsFromCategory: 'First Category') el tamaño es
    igual a: 1
```

Luego podemos definir la funcionalidad y asegurarnos de que pase nuestra prueba.

```
TBBlog >> allBlogPublicacionesDeCategoría: unaCategoría
    publicaciones seleccione: [ :p | p categoría = aCategoría ]
```

Verifique que pase la prueba.

Todas las publicaciones visibles de una categoría

La siguiente prueba verifica que podemos acceder a todas las publicaciones visibles de una categoría determinada. Una vez definido, debemos asegurarnos de que la prueba falló.

```
TBBlogTest >> testAllVisibleBlogPublicacionesDeCategoría
    blog escribirBlogPost: publicación.
    autoafirmación: (blog allVisibleBlogPostsFromCategory: 'Primera categoría')
    tamaño igual a: 0. autoafirmación: (blog allVisibleBlogPostsFromCategory:
    'Segunda categoría') tamaño igual a: 1
```

Luego podemos definir la funcionalidad y asegurarnos de que pase nuestra prueba.

```
TBBlog >> allVisibleBlogPostsFromCategory: aCategory posts select:
    [ :p | p categoría = una categoría
        y: [ p esVisible ] ]
```

Verifique que pase la prueba.

Revisa las publicaciones sin clasificar

La siguiente prueba verifica que no tenemos blogs sin clasificar en nuestro dispositivo de prueba.

```
[ TBBlogTest >> testUnclassifiedBlogPosts
    autoafirmación: (blog allBlogPosts select: [ :p | p isUnclassified
        ]) el tamaño es igual a: 0
```

Verifique que pase la prueba.

Recuperar todas las categorías

Nuevamente definimos una nueva prueba y verificamos que falla.

```
[ TBBlogTest >> testAllCategories blog
    writeBlogPost: publicación.
    autoafirmación: blog todas las categorías el tamaño es igual a: 2
```

Luego agregamos el nuevo comportamiento.

```
[ TBBlog >> allCategories ^
    (self allBlogPosts recopilar: [ :p | p categoría ]) activo
```

Verifique que pase la prueba.

3.7 Datos de prueba

Para ayudarlo a probar la aplicación, puede agregar el siguiente método que crea múltiples publicaciones.

```
[ Clase TBBlog >> createDemoPosts
    "TBBlog createDemoPosts" self
    actual writeBlogPost: ((Título de
        TBPost: 'Bienvenido a TinyBlog' texto: 'TinyBlog es un pequeño motor de
        blog hecho con Pharo.' categoría: 'TinyBlog') visible: verdadero);

    writeBlogPost: ((TBPost title: 'Report Pharo Sprint' text: 'El viernes 12 de
        junio hubo un Pharo sprint / Moose dojo. Fue un evento agradable con más de 15
        velocistas motivados. Con la ayuda de dulces, pasteles y chocolate, se ha hecho
        un gran trabajo' categoría: 'Pharo') visible: verdadero);

    writeBlogPost: ((TBPost title: 'Brick on top of Bloc - Preview' text: 'Nos
        complace anunciar la primera versión preliminar de Brick, un nuevo
        conjunto de widgets creado desde cero sobre Bloc. Brick está siendo
        desarrollado principalmente por Alex Syrel (junto con Alain Plantec, Andrei Chis
        y yo mismo), y el trabajo está patrocinado por ESUG.

        Brick es parte del esfuerzo Glamorous Toolkit y proporcionará
        la base para las nuevas versiones de las herramientas de desarrollo.'
        categoría: 'Pharo') visible: verdadero);
```

3.8 Posibles Extensiones

```
writeBlogPost: ((Título de TBPost: 'La triste historia de las publicaciones de blog sin clasificar' texto: 'Tan triste que puedo leer esto.') visible: verdadero); writeBlogPost: ((TBPost title: 'Trabajando con Pharo en Raspberry Pi' text: 'El hardware es cada vez más barato y muchos nuevos dispositivos pequeños como el famoso Raspberry Pi brindan una nueva potencia de cálculo que antes solo estaba disponible en las computadoras de escritorio normales' categoría : 'Pharo') visible: verdadero)
```

Si inspecciona el resultado del siguiente fragmento, verá que el blog actual contiene 5 publicaciones:

```
[ TBBlog createDemoPosts; actual
```

Tenga en cuenta que si ejecuta este método `createDemoPosts` varias veces, el objeto singleton de su blog contendrá varias copias de estas publicaciones.

3.8 Posibles Extensiones

Se pueden hacer muchas extensiones como: recuperar la lista de categorías que contiene al menos una publicación visible, eliminar una categoría y todas las publicaciones que contiene, cambiar el nombre de una categoría, mover una publicación de una categoría a otra, hacer (in)visible una categoría y todo su contenido, etc. Te animamos a desarrollar Algunos.

3.9 Conclusión

Ahora tiene el modelo completo de TinyBlog, así como algunas pruebas unitarias. Ahora está listo para implementar funciones más avanzadas, como el almacenamiento de la base de datos o un primer front-end web. No olvides guardar tu código.

CAPÍTULO 4

Persistencia de datos usando Voyage y Mongo

Hasta ahora usábamos objetos modelo almacenados en la memoria y funciona bien porque al guardar la imagen de Pharo también se guardan estos objetos. Sin embargo, sería mejor guardar estos objetos (entradas de blog) en una base de datos externa. Pharo admite múltiples serializadores de objetos como Fuel (formato binario) o STON (formato de texto). Estos serializadores son útiles y potentes. A menudo, con una sola línea de código, podemos guardar un gráfico completo de los objetos, como se explica en el libro Enterprise Pharo disponible en <http://books.pharo.org>.

En este capítulo, usaremos otra posibilidad: guardar datos en una base de datos de documentos como Mongo (<https://www.mongodb.com>) utilizando el marco de trabajo de Voyage. Voyage proporciona una API unificada para almacenar y recuperar objetos en varias bases de datos basadas en documentos, como Mongo o UnQLite. Pero primero, usaremos Voyage y su capacidad para simular una base de datos externa en la memoria. Esto es realmente útil durante el desarrollo. Luego, puede instalar una base de datos Mongo local y acceder a ella a través de Voyage. Como verá, este segundo paso tendrá muy poco impacto en nuestro código.

El último capítulo explica cómo cargar el código de los capítulos anteriores si es necesario.

4.1 Configurar Voyage para guardar objetos TBBlog

Al definir el método de clase `isVoyageRoot`, declaramos que los objetos de esta clase deben guardarse en la base de datos como objetos raíz. Significa que la base de datos contendrá tantos documentos como instancias de esta clase.

Clase TBBlog >> isVoyageRoot

"Indica que las instancias de esta clase son documentos de nivel superior en bases de datos
 ^ nosQL"
 verdadero

Deberíamos establecer conexión a una base de datos real o trabajar en la memoria.

Comencemos a trabajar en la memoria usando esta expresión:

VOMemoryRepository nuevo enableSingleton.

El mensaje enableSingleton le indica a Voyage que usaremos solo una base de datos. Esto nos liberará para especificar la base de datos cada vez. Creamos e inicializamos la base de datos en la memoria en un método del lado de la clase llamado initial izeVoyageOnMemoryDB.

Clase TBBlog >> initializeVoyageOnMemoryDB

VOMemoryRepository nuevo enableSingleton

El método reset class reinicializa la base de datos. El método de clase de inicialización garantiza que la base de datos se inicialice cuando cargamos el código de TinyBlog. No olvide ejecutar esta expresión TBBlog initialize para asegurarse de que la base de datos se inicialice.

Clase TBBlog >> restablecer

auto inicializar VoyageOnMemoryDB

Clase TBBlog >> inicializar reinicio

automático

El método actual del lado de la clase es más complicado. Antes de usar Voyage, implementamos un patrón singleton simple (actual TBBlog). Sin embargo, ya no funciona porque imagina que guardamos nuestro blog y que el servidor se detiene por accidente o que recargamos una nueva versión del código, reiniciaría la conexión y crearía una nueva instancia nueva del blog. Entonces sería posible terminar con una instancia diferente a la guardada.

Así que cambiamos la implementación del método de clase actual para realizar una solicitud de base de datos y recuperar objetos guardados. Como solo guardamos un objeto de blog, solo consiste en hacer: self selectOne: [:each | verdadero] o self selectAll anyOne. Si la base de datos no contiene ninguna instancia, creamos una nueva y la guardamos.

Clase TBBlog >> auto actual

^ selectAll

si no está vacío: [:x | x anyOne] ifEmpty:

[self new save]

También podemos eliminar la variable de instancia de clase llamada uniqueInstance que usamos anteriormente para almacenar nuestro objeto singleton.

Nombres de

variables de instancia de la clase TBBlog: "

4.2 Guardar un blog

4.2 Guardar un blog

Cada vez que modificamos un objeto de blog, debemos propagar los cambios a la base de datos. Por ejemplo, modificamos el método writeBlogPost: para guardar el blog cuando agregamos una nueva publicación.

```
TBBlog >> escribirBlogPost: aPost
    "Escribir la publicación del blog en la base de
    datos" self allBlogPosts add: aPost. salvarse a
    sí mismo
```

También guardamos el blog cuando eliminamos (método de eliminación) una publicación de un blog.

```
TBBlog >> removeAllPosts posts :=
    OrderedCollection new. salvarse a sí mismo
```

4.3 Revisión de pruebas unitarias

Ahora guardamos los blogs en una base de datos, ya sea en memoria o en un servidor Mongo externo, a través de Voyage. Debemos tener cuidado con las pruebas unitarias que modifican la base de datos porque pueden corromper los datos de producción. Para evitar esta situación peligrosa, una prueba no debe modificar el estado del sistema.

Para resolver esta situación, antes de ejecutar una prueba mantendremos una referencia al blog actual y crearemos un nuevo contexto y lo restauraremos después de la ejecución de la prueba.

Agreguemos una variable de instancia anteriorRepository en la clase TBBLogTest .

```
Subclase TestCase: #TBBlogTest
    instanceVariableNames: 'primera entrada de blog anteriorRepository'
    classVariableNames: ''
    paquete: 'Pruebas de TinyBlog'
```

Luego, modificamos el método setUp para guardar la base de datos antes de cada ejecución de prueba. Creamos un objeto de base de datos temporal que será utilizado por la prueba.

```
TBBlogTest >> configurar
    anteriorRepository := VORespository actual.
    VORespository setRepository: VOMemoryRepository nuevo. blog := TBBlog
    actual. primero := TB Título de la publicación: 'Un título' texto: 'Un texto'
    categoría: 'Primero
        Categoría'.
    blog escribirBlogPost: primero.
    publicación := (Título de la publicación TB: 'Otro título' texto: 'Otro texto' categoría:
    'Segunda categoría') beVisible
```

En el método tearDown ejecutado después de cada prueba, restauramos el objeto de la base de datos original.

```
[ TBBlogTest >> derribar  
  VORespository setRepository: anteriorRepositorio
```

4.4 Consultando la base de datos

La base de datos está actualmente en la memoria y podemos acceder al objeto de blog usando el método del lado de la clase actual de la clase `TBBlog`. Basta con mostrar la API de Voyage ya que será lo mismo acceder a una base de datos real de Mongo.

Puedes crear publicaciones:

```
[ TBBlog createDemoPosts
```

Puede contar el número de blogs guardados. `count` es parte de la API de Voyage. En este ejemplo, obtenemos el resultado 1 porque el blog se implementa como Single ton.

```
[ TBRuento de blogs  
>1
```

De manera similar, puede recuperar todos los objetos raíz guardados de un tipo.

```
[ TBBlog seleccionar todo
```

También puede eliminar un objeto raíz utilizando el mensaje de eliminación .

Puede descubrir más sobre la API de Voyage consultando:

- la clase Clase ,
- la clase VORespository que es la raíz de la jerarquía de todas las bases de datos ya sea en memoria o externa.

Esas consultas serán más relevantes con más objetos pero serían similares.

4.5 Si guardáramos las publicaciones [Discusión]

Esta sección no debe implementarse. Solo se describe como un ejemplo (puede encontrar más información sobre Voyage en el libro de Enterprise Pharo <http://books.pharo.org>). Queremos ilustrar que declarar una clase como raíz de Voyage influye en cómo se guarda y recarga una instancia de esta clase.

Hasta ahora, una publicación (una instancia de `TBPost`) no se declara como raíz de Voyage. Por lo tanto, los objetos de publicación se guardan como subpartes en el objeto de blog al que pertenecen. Implica que no se garantiza que una publicación sea única después de guardarla y volver a cargarla desde la base de datos. De hecho, después de cargar cada objeto del blog, tendrá sus propios objetos de publicaciones, incluso si algunas publicaciones se compartieron antes de guardarlas. Los objetos compartidos antes de guardar se duplicarán para cada objeto raíz después de la carga.

4.6 Configurar una base de datos Mongo externa [Opcional]

Podemos declarar publicaciones como objetos raíz, lo que significa que una publicación se puede guardar de forma independiente desde un blog. Implica que los blogs guardados tienen una referencia a un objeto TBPost . Esto preservaría el intercambio de publicaciones entre objetos de blog.

Sin embargo, no todos los objetos deben ser objetos raíz. Si representamos comentarios de publicaciones, no los definiríamos también como objetos raíz porque manipular un comentario fuera de su contexto (una publicación) no tiene sentido.

Publicar como raíz = Exclusividad Si desea

compartir publicaciones y hacerlas únicas entre varios blogs, la clase TBPost debe declararse como raíz en la base de datos. En este caso, las publicaciones se guardan como entidades autónomas y las instancias de TBBlog harán referencia a las entidades de publicaciones en lugar de incrustarlas. La consecuencia es que una publicación es única y se puede compartir a través de una referencia desde un blog. Para lograr esto, definiríamos los siguientes métodos:

Clase TBPost >> isVoyageRoot

"Indica que las instancias de esta clase son documentos de nivel superior en bases de datos noSQL"

verdadero

Durante la adición de una publicación a un blog, sería importante guardar tanto el blog como la nueva publicación.

TBBlog >> escribirBlogPost: aPost

Las publicaciones "Escribe la publicación del blog en la base de datos" agregan: aPost.

aPublicar guardar

salvarse a sí mismo

TBBlog >> las publicaciones removeAllPosts

hacen: [:each | cadauitar]. publicaciones: =
OrderedCollection nuevo. salvarse a sí mismo

En el método removeAllPosts , primero eliminamos todas las publicaciones, luego actualizamos la colección y finalmente guardamos el blog.

4.6 Configurar una base de datos Mongo externa [Opcional]

Al usar Voyage, podemos guardar fácilmente nuestros objetos de modelo en una base de datos de Mongo. Esta sección explica cómo proceder y las pocas modificaciones a realizar en nuestro código. Esto no es obligatorio hacerlo. Incluso si lo hace, lo alentamos a que continúe trabajando con una base de datos de memoria después.

Instalación de Mongo

Independientemente de su sistema operativo (Linux, MacOS o Windows), puede instalar un servidor Mongo local en su máquina (cf. <https://www.mongodb.com>).

Esto es útil para probar su aplicación sin necesidad de una conexión a Internet. En lugar de instalar directamente Mongo, sugerimos instalar Docker (<https://www.docker.com>) en su máquina y ejecute un contenedor Mongo usando la siguiente línea de comando:

```
[ ventana acopiable ejecutar --nombre mongo -p 27017:27017 -d mongo
```

Nota El servidor Mongo en ejecución no debe usar autenticación (no es el caso con la instalación predeterminada) porque el nuevo mecanismo de autenticación SCRAM utilizado por Mongo 3.0 actualmente no es compatible con Voyage.

Algunos comandos útiles de Docker:

```
# para detener la ventana acopiable del contenedor
Mongo docker stop mongo

# para reiniciar la ventana acopiable de
su contenedor, inicie mongo

# para eliminar su contenedor (debe detenerse antes) docker rm mongo
```

Conexión de un servidor Mongo local Una vez

instalado, puede conectarse a un servidor Mongo directamente desde Pharo. Definimos el método llamado `initializelocalhostMongoDB` para establecer la conexión con el servidor Mongo local (`localhost`, puerto predeterminado) y acceder a la base de datos llamada '`tinyblog`'.

```
[ Clase TBBlog >> initializelocalhostMongoDB | repositorio |
    repositorio := VOMongoBase de datos del repositorio:
    'tinyblog'. repositorio enableSingleton.
```

Restablezca la clase para establecer una nueva conexión a la base de datos.

```
[ Clase TBBlog >> restablecer
    autoinicializaciónlocalhostMongoDB
```

Ahora, si recrea publicaciones de demostración, se guardan automáticamente en su Base de datos Mongo:

```
[ Restablecimiento de TBBlog.
TBBlog createDemoPosts
```

4.7 Conclusión

En caso de problemas

Si necesita reiniciar completamente una base de datos externa, puede usar el método `dropDatabase`.

```
(Repositorio VOMongo  
host: 'localhost' base  
de datos: 'tinyblog') dropDatabase
```

También puede hacerlo en la línea de comando cuando mongod se está ejecutando con:

```
mongo tinyblog --eval "db.dropDatabase()"
```

o conectándose al contenedor docker en el que se está ejecutando:

```
docker exec -it mongo bash -c 'mongo tinyblog --eval  
"db.dropDatabase()"'
```

Puntos de atención: Cambiar la definición de TBBlog

Cuando utiliza una base de datos Mongo externa en lugar de una de memoria, cada vez que agrega nuevos objetos raíz o modifica la definición de algunos objetos raíz, es importante restablecer la memoria caché mantenida por Voyage. Se puede hacer usando:

```
Restablecimiento actual de VORespository
```

4.7 Conclusión

Voyage propone una buena API para administrar de forma transparente el almacenamiento de objetos en la memoria o en una base de datos de documentos. Los datos de la aplicación ahora se guardan en una base de datos y estamos listos para construir la interfaz de usuario web.

CAPÍTULO 5

Primeros pasos con Seaside

En este capítulo, configuraremos Seaside y construiremos nuestro primer componente Seaside. En los próximos capítulos, desarrollaremos la parte pública de TinyBlog, luego el sistema de autenticación, seguido de la parte de administración reservada a los administradores del blog.

Todo el tiempo, definiremos los componentes de Seaside <http://www.seaside.st>. Un libro de referencia está disponible en línea <http://book.seaside.st> y los primeros capítulos pueden ayudarte y ser un gran compañero de este libro tutorial.

Todo el trabajo siguiente es independiente de Voyage y de la base de datos de Mongo. Como de costumbre, puede descargar el código de los capítulos anteriores como se explica en el último capítulo.

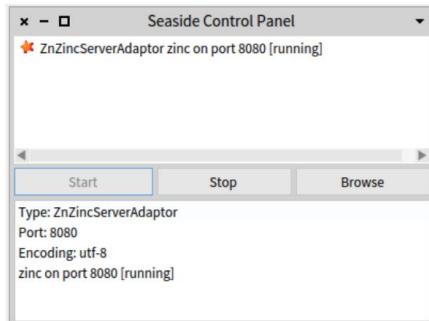


Figura 5-1 Inicio del servidor Seaside.

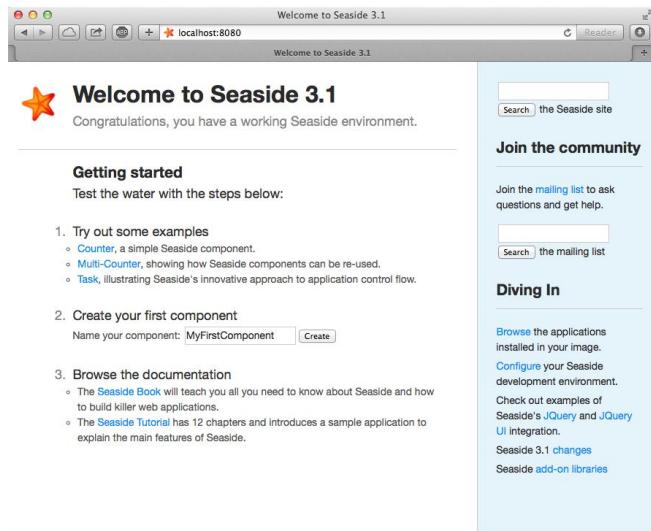


Figura 5-2 Corriendo junto al mar.

5.1 Comenzando junto al mar

Seaside ya debería estar cargado en su imagen de PharoWeb. Si no es así, consulte el capítulo de carga.

Hay dos formas de iniciar Seaside. La primera consiste en ejecutar el siguiente snippet:

```
[ ZnZincServerAdaptor startOn: 8080.]
```

El segundo utiliza la herramienta gráfica denominada "Panel de control de Seaside" (Menú Herramientas>Panel de control de Seaside). En el menú contextual (clic derecho) de esta herramienta, seleccione "agregar adaptador..." y agregue un servidor de tipo ZnZincServerAdaptor, luego defina el número de puerto (por ejemplo, 8080) en el que debe ejecutarse (ver Figura 5-1). Al abrir un navegador web en la URL <http://localhost:8080>, debería ver la página de inicio de Seaside como se muestra en la Figura 5-2.

5.2 Bootstrap para Seaside

Se puede acceder directamente a la biblioteca Bootstrap desde Pharo y Seaside. El repositorio y la documentación de Bootstrap para Pharo están disponibles aquí: <https://github.com/astares/Seaside-Bootstrap4>. Pero ya está cargado en la imagen de PharoWeb que estamos usando con este libro.

Puede buscar los ejemplos localmente en su navegador haciendo clic en el enlace de arranque en la lista de aplicaciones alojadas por Seaside o ingresando directamente

5.3 Definir nuestro punto de entrada de la aplicación

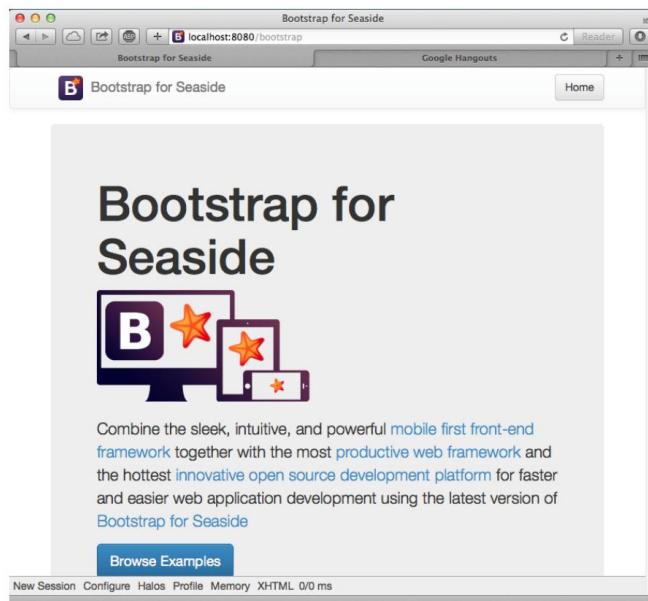


Figura 5-3 Exploración de la biblioteca Seaside Bootstrap.

esta URL <http://localhost:8080/bootstrap>. Debería ver ejemplos de Bootstrap como se muestra en la Figura 5-3.

Al hacer clic en el enlace **Ejemplos** en la parte inferior de la página, puede ver los elementos gráficos de Bootstrap y el código de Seaside necesario para obtenerlos (consulte la Figura 5-4).

5.3 Definir nuestro punto de entrada de la aplicación

Cree una clase llamada **TBApplicationRootComponent** que será el punto de entrada de la aplicación.

Subclase **WACOMPONENT**: #**TBApplicationRootComponent**
 nombres de variables de instancia:
 " nombres de variables de clase: "
 paquete: 'componentes de TinyBlog'

Registraremos la aplicación TinyBlog en el servidor de aplicaciones Seaside definiendo el método de clase de inicialización en el protocolo de 'inicialización' .

También integramos dependencias al marco Bootstrap (los archivos CSS y JS se incrustarán en la aplicación).

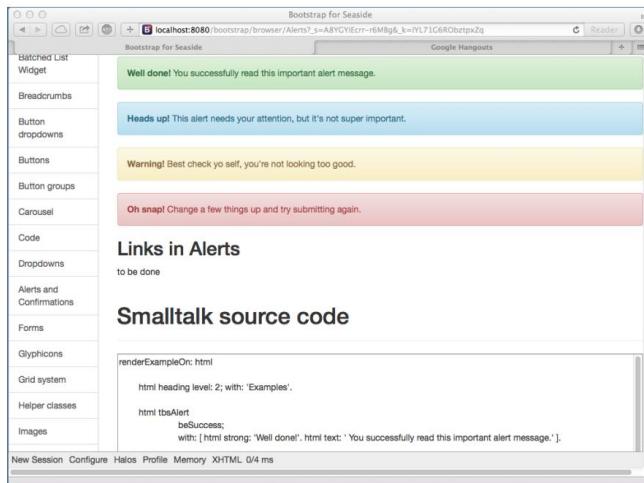


Figura 5-4 Un elemento Bootstrap y su código.

Clase TBApplicationRootComponent >> inicializar "autoinicializar" | aplicación
| app := WAAdmin registro: self asApplicationAt: 'TinyBlog'.

aplicación addLibrary: JQDeploymentLibrary; addLibrary:
JQUIDeploymentLibrary; addLibrary: TBSDeploymentLibrary

Una vez declarado, debe ejecutar este método con **TBApplicationRoot Component initialize**. De hecho, los métodos de inicialización del lado de la clase se ejecutan en el momento de la carga de una clase, pero dado que la clase ya existe, debemos ejecutarla a mano.

También agregamos un método llamado **canBeRoot** para especificar que **TBApplication RootComponent** no es un simple componente de Seaside sino una **aplicación completa**. Este componente se instanciará automáticamente cuando un usuario se conecte a la aplicación.

Clase TBApplicationRootComponent >> canBeRoot
verdadero

Puede verificar que su aplicación esté correctamente registrada en Seaside conectándose al servidor de Seaside a través de su navegador web, haga clic en "Examinar las aplicaciones instaladas en su imagen" y luego vea que TinyBlog aparece en la lista como se ilustra en la Figura 5-5 . Alternativamente, puede visitar <http://localhost:8080/TinyBlog>.

5.4 Primera representación simple

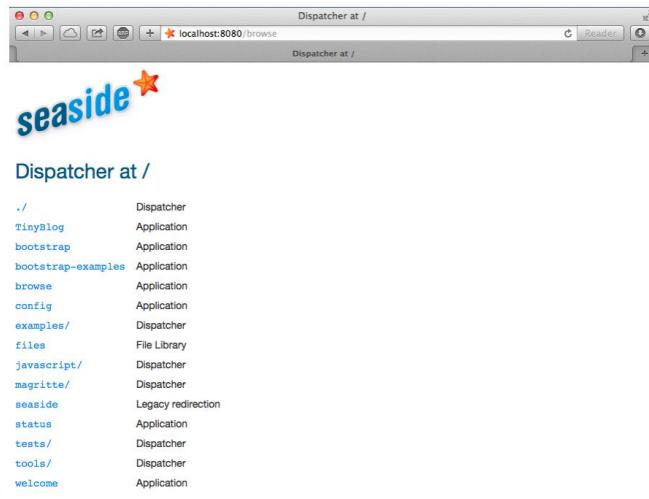


Figura 5-5 TinyBlog es una aplicación Seaside registrada.

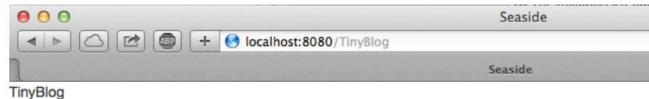


Figura 5-6 Una primera página web de Seaside.

5.4 Primera representación simple

Agreguemos un método de instancia llamado `renderContentOn:` en el protocolo de procesamiento para hacer que nuestra aplicación muestre algo.

```
[ TBAplicationRootComponent >> renderContentOn: html texto
    html: 'TinyBlog'
```

Si abre `http://localhost:8080/TinyBlog` en su navegador web, la página debería parecerse a la de la Figura 5-6.

Puede personalizar el encabezado de la página web y declararlo compatible con HTML 5 redefiniendo el método `updateRoot:`.

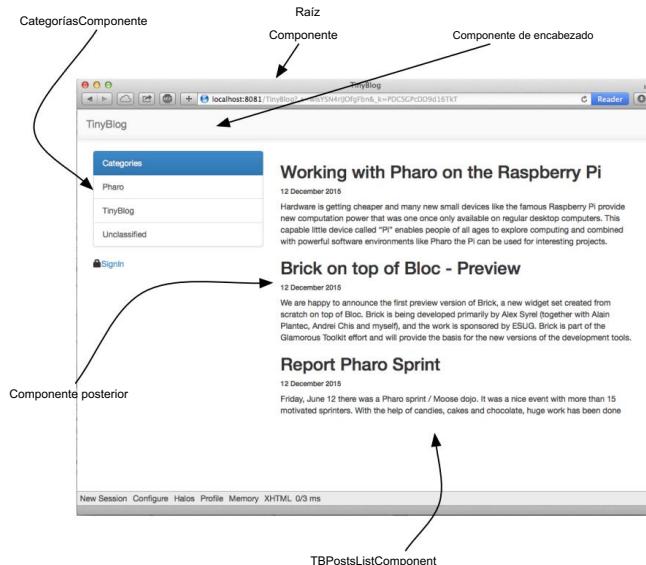


Figura 5-7 Componentes principales de TinyBlog (vista pública).

```
TBApplicationRootComponent >> updateRoot: anHtmlRoot
    super updateRoot: anHtmlRoot.
    unHtmlRoot beHtml5.
    un título HtmlRoot: 'TinyBlog'
```

El **título**: mensaje es responsable de configurar el título de la página, como se puede ver en la barra de título de su navegador web. El componente **TBApplicationRootComponent** es el componente raíz de nuestra aplicación. No mostrará muchas cosas. A continuación, contendrá y mostrará otros componentes. Por ejemplo, un componente para mostrar publicaciones a los lectores del blog, un componente para administrar el blog y sus publicaciones, ...

5.5 Arquitectura

Ahora estamos listos para definir los componentes visuales de nuestra aplicación web.

Descripción general de

TinyBlog La Figura 6-2 muestra una descripción general de ellos y sus responsabilidades, mientras que la Figura 5-8 muestra la arquitectura general de nuestra aplicación y las relaciones entre esos componentes.

5.6 Conclusión

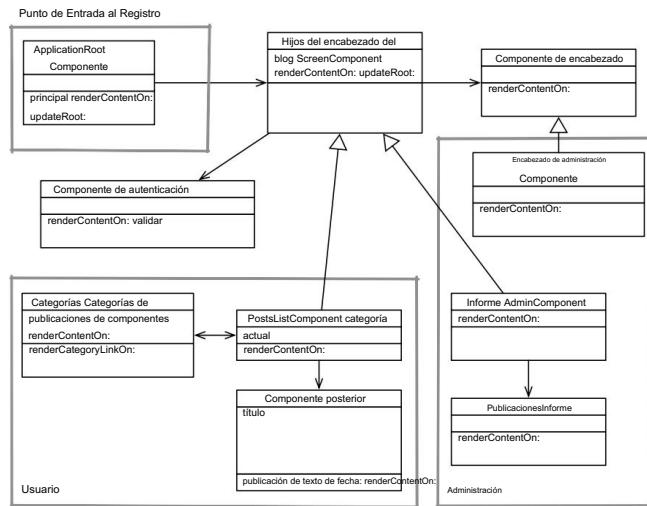


Figura 5-8 Arquitectura de TinyBlog.

Descripción de los componentes

principales Para facilitar su comprensión del desarrollo incremental de esta aplicación, la Figura 5-8 describe la arquitectura objetivo.

- **ApplicationRootComponent** es el punto de entrada registrado en Seaside. Este componente contiene componentes heredados de la clase abstracta **ScreenComponent**.
- **ScreenComponent** es la raíz de los componentes usados para construir el Vista pública y administrativa de la aplicación. Se compone de un encabezado.
- **PostsListComponent** es el componente principal que muestra las publicaciones. Está compuesto por instancias de **PostComponent**) y administra categorías.
- **AdminComponent** es el componente principal de la vista de administración. Se compone de un componente de informe (instancia de **PostsReport**) creado con Magritte.

5.6 Conclusión

Ahora estamos listos para comenzar el desarrollo de los componentes descritos.

En los próximos capítulos, lo guaremos linealmente para desarrollar esos componentes. Si se siente perdido en algún momento, lo invitamos a regresar a esta descripción general de la arquitectura para comprender mejor lo que estamos desarrollando.

CAPÍTULO 6

Componentes web para TinyBlog

En este capítulo, construimos la vista pública de TinyBlog que muestra las publicaciones del blog. La Figura 6-1 muestra los componentes en los que trabajaremos durante este capítulo. Si te sientes perdido en algún momento, consúltalo.

Antes de comenzar, puede cargar el código de los capítulos anteriores como se describe en el último capítulo de este libro.

6.1 Componentes visuales

La Figura 6-2 muestra los componentes visuales que definiremos en este capítulo y dónde se mostrarán.

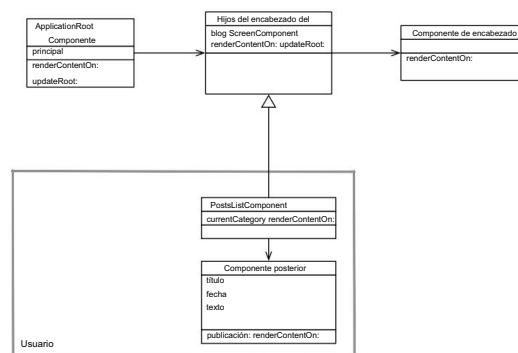


Figura 6-1 Arquitectura de componentes de la vista pública (opuesta a la vista de administración).

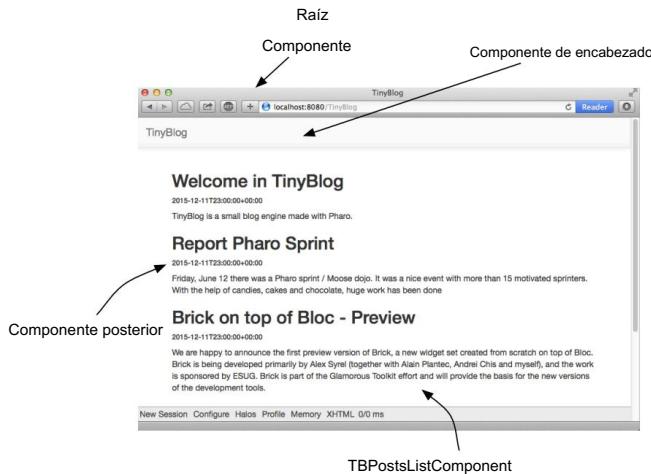


Figura 6-2 Componentes visuales de TinyBlog.

El componente `TBScreenComponent` **Todos los**

componentes contenidos en `TBApplicationRootComponent` serán subclases de la clase abstracta `TBScreenComponent`. Esta clase nos permite factorizar el comportamiento compartido entre todos nuestros componentes.

Subclase `WAComponent`: `#TBScreenComponent`
 nombres de variables de instancia: "
 Nombres de variables de clase: "
 paquete: 'Componentes TinyBlog'

Todos los componentes necesitan acceder al modelo de nuestra aplicación. Por lo tanto, en el protocolo de 'acceso', agregamos un método de blog que devuelve la instancia actual de `TBBlog` (el singleton). En el futuro, si desea administrar varios blogs, modificará este método y devolverá el objeto de blog con el que se configuró.

`TBScreenComponent >> blog`

"Devolver el blog actual. En el futuro, le pediremos a la sesión que devuelva el blog del usuario conectado actualmente".
 TBBlog actual

Definamos un método `renderContentOn`: en este nuevo componente que muestra un mensaje temporalmente. Si actualiza su navegador, no aparece nada porque este nuevo componente aún no se muestra en absoluto.

`Componente TBScreen >> renderContentOn: html`
 texto html: 'Hola desde TBScreenComponent'

6.2 Uso del componente TBScreenComponent

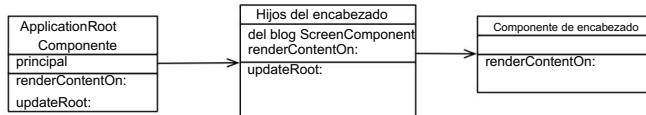


Figura 6-3 ApplicationRootComponent usa temporalmente un ScreenComponent que contiene un HeaderComponent.

6.2 Uso del componente TBScreenComponent

En la arquitectura final, TBScreenComponent es un componente abstracto y no debe usarse directamente. Sin embargo, lo usaremos temporalmente mientras desarrollamos otros componentes.

Agreguemos una variable de instancia principal en la clase TBApplicationRootComponent . Obtenemos la situación descrita en la figura 6-3.

Subclase WACOMPONENT: #TBApplicationRootComponent
nombres de variables de instancia:
'principal' nombres de variables de clase:
" paquete: 'componentes de TinyBlog'

Inicializamos esta variable de instancia en el método de inicialización con una nueva instancia de TBScreenComponent.

TBApplicationRootComponent >> inicializar super
inicializar. principal := TBScreenComponent nuevo

Hacemos el TBApplicationRootComponent para representar este subcomponente.

TBApplicationRootComponent >> renderContentOn: html
representación html: principal

No olvidemos declarar que el objeto contenido en la variable de instancia principal es un subcomponente de TBApplicationRootComponent al redefinir el método de los niños .

TBApplicationRootComponent >> niños {principal}

La Figura 6-4 muestra el resultado que debería obtener en su navegador. Actualmente, solo existe el texto: Hello from TBScreenComponent mostrado por el subcomponente TBScreenComponent . (ver figura 6-4).

6.3 Patrón de Definición de Componente

A menudo usaremos los mismos pasos siguientes:

- primero, definimos una clase y el comportamiento de un nuevo componente;



Figura 6-4 Primera representación visual de TBScreenComponent.

- luego, lo referenciamos desde un componente existente que lo usa; • y expresamos la relación compuesto/subcomponente por redefinir ing el método de los niños .

6.4 Poblando el Blog

Puede inspeccionar el objeto de blog devuelto por TBBlog actual y verificar que contiene algunas publicaciones. También puedes hacerlo simplemente como:

[TBBlog actual allBlogPosts tamaño

Si no lo hace, ejecute:

[TBBlog createDemoPosts

6.5 Definición de TBHeaderComponent

Definamos un componente llamado **TBHeaderComponent** que **represente el encabezado común de todas las páginas de TinyBlog**. Este componente se insertará en la parte superior de todos los componentes, como **TBPostsListComponent**. Usamos el patrón descrito anteriormente: defina la clase del componente, haga referencia a él desde su componente de cierre y redefina el método de los niños .

Aquí la definición de clase:

[Subclase WACOMPONENT: #TBHeaderComponent
nombres de variables de instancia: "
Nombres de variables de clase: "
paquete: 'Componentes TinyBlog'

6.6 Uso de TBHeaderComponent

Recuerde que **TBScreenComponent es la raíz (abstracta) de todos los componentes de nuestra arquitectura final. Por tanto, introduciremos nuestra cabecera en TBScreenComponent para que todas sus subclases la hereden**. Dado que no es deseable crear una instancia de TBHeaderComponent cada vez que se llama a un componente, almacenamos el encabezado en una variable de instancia denominada **encabezado**.

6.7 Relación compuesto-componente

```

[ Subclase WAComponent: #TBScreenComponent nombres de
variables de instancia: 'encabezado'
Nombres de variables de clase: "
paquete: 'Componentes TinyBlog'
```

Lo inicializamos en el método de inicialización categorizado en el protocolo de 'inicialización':

```

[ TBScreenComponent >> inicializar super
  inicializar. encabezado := self
  createHeaderComponent

[ Componente de pantalla TBS >> crear componente de encabezado
  ^ TBHeaderComponent nuevo
```

Tenga en cuenta que usamos un método específico llamado `createHeaderComponent` para crear la instancia del componente de encabezado. La redefinición de este método hace posible cambiar completamente el componente de encabezado que se utiliza. Usaremos eso para mostrar un componente de encabezado diferente para la administración vista.

6.7 Relación compuesto-componente

En Seaside, los subcomponentes de un componente deben ser devueltos por el compuesto al enviarle el mensaje secundario . Por lo tanto, debemos definir que la instancia de TBHeaderComponent es un elemento secundario del componente TBScreenComponent en la jerarquía de componentes de Seaside (y no en la jerarquía de clases de Pharo). Lo hacemos especializando el método niños. En este ejemplo, devuelve una colección de un elemento que es la instancia de TBHeaderComponent al que hace referencia la variable de instancia del encabezado .

```

[ TBScreenComponent >> niños { encabezado }
```

6.8 Renderizar un encabezado

En el método `renderContentOn:` (protocolo 'rendering'), ahora podemos mostrar el subcomponente (el encabezado):

```

[ Componente TBScreen >> renderContentOn: html
  representación html: encabezado
```

Si actualiza su navegador, no aparece nada porque TBHeaderCompo nent no tiene representación. Agreguemos un método `renderContentOn:` que muestre un encabezado de navegación de Bootstrap:

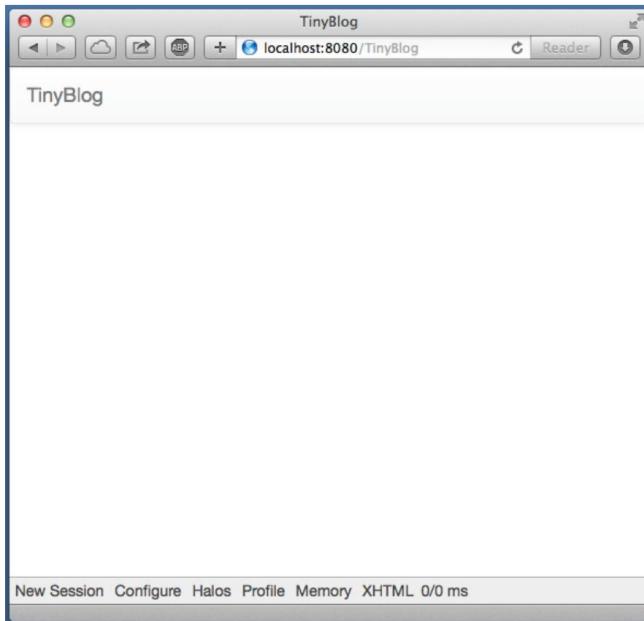


Figura 6-5 TinyBlog con encabezado Bootstrap.

```

[ TBHeaderComponent >> renderContentOn: html html
  tbsNavbar beDefault; con: [ html tbsContainer: [
    self renderBrandOn: html
  ]]

[ TBHeaderComponent >> renderBrandOn: html html
  tbsNavbarHeader: [ html tbsNavbarBrand

    url: url de autoaplicación; con:
    'TinyBlog' ]
  
```

Su navegador ahora debería mostrar lo que se muestra en la Figura 6-5. Como es habitual en la barra de navegación de Bootstrap, el enlace en el título de la aplicación (`tbsNavbarBrand`) permite a los usuarios volver a la página de inicio de la aplicación.

Posibles mejoras

El nombre del blog debe poder personalizarse mediante una variable de instancia en la clase Blog de TB y el componente de encabezado de la aplicación debe mostrar este título.

6.9 Lista de publicaciones

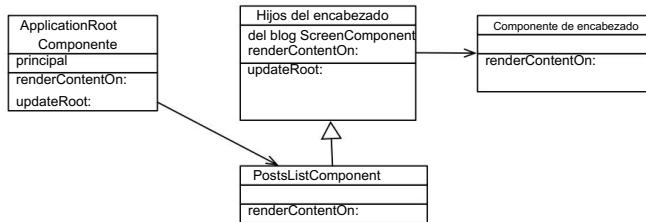


Figura 6-6 ApplicationRootComponent utiliza PostsListComponent .

6.9 Lista de publicaciones

Vamos a crear un TBPostsListComponent heredado de TBScreenComponent para mostrar la lista de todas las publicaciones. Recuerde que aquí hablamos del acceso público al blog y no de la interfaz de administración que se desarrollará más adelante.

Subclase TBScreenComponent: #TBPostsListComponent
nombres de variables de instancia:
" nombres de variables de clase: "
paquete: 'componentes de TinyBlog'

Ahora podemos modificar TBApplicationRootComponent, el componente principal de la aplicación, para que muestre este nuevo componente como se muestra en la figura 6-6. Para lograr esto, modificamos su método de inicialización :

```
TBApplicationRootComponent >> inicializar super
  inicializar. principal := TBPostsListComponent nuevo
```

Agregamos un método setter llamado `main`: para cambiar dinámicamente el subcomponente para mostrar, pero por defecto es una instancia de TBPostsListComponent.

```
TBApplicationRootComponent >> principal: unComponente
  principal := unComponente
```

Ahora agregamos un método temporal `renderContentOn`: (en el protocolo 'rendering') en TBPostsListComponent para probar durante el desarrollo (cf. Figura 6-7). En este método, llamamos al `renderContentOn`: de la superclase que representa el componente de encabezado.

```
TBPostsListComponent >> renderContentOn: html
  super renderContentOn: html. texto
  html: '¡Publicaciones de blog aquí!'
```

Si actualiza TinyBlog en su navegador, ahora debería ver lo que se muestra en la figura 6-7.

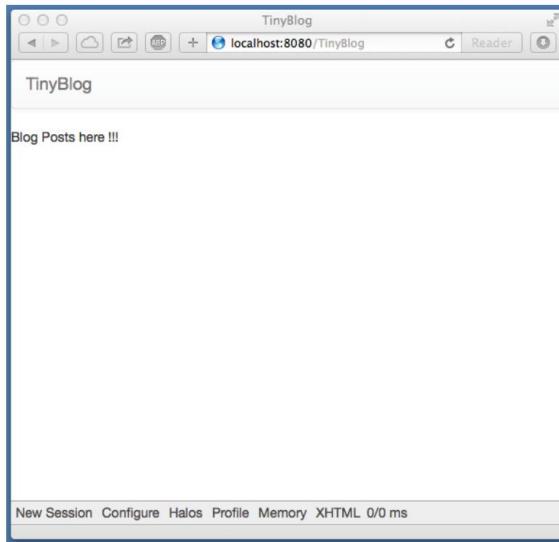


Figura 6-7 TinyBlog mostrando una lista de publicaciones básicas.

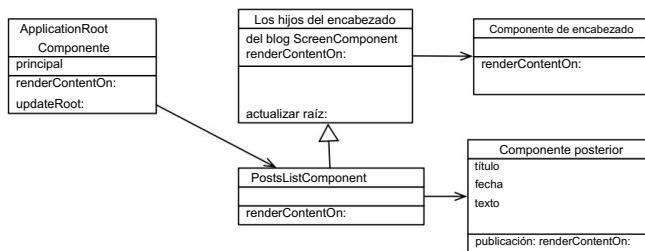


Figura 6-8 Uso de PostComponents para mostrar cada publicación.

6.10 El Componente Post

Ahora definiremos TBPostComponent para mostrar los detalles de una publicación. Cada publicación se mostrará gráficamente mediante una instancia de TBPostComponent que mostrará el título de la publicación, su fecha y su contenido, como se muestra en la figura 6-8.

```

[ Subclase WACOMPONENT: #TBPostComponent
  nombres de variables de instancia: 'post'
  nombres de variables de clase: "
  paquete: 'Componentes TinyBlog'

  TBPostComponent >> inicializar super
  inicializar. publicar := TBPUBLICAR
  nuevo
  ]

```

6.11 Mostrar publicaciones

```
[ TBPostComponent >> título de la
    ^ publicación
```

```
[ TBPostComponent >> texto de
    ^ publicación de texto
```

```
[ TBPostComponent >> fecha posterior
    ^ a la fecha
```

El método renderContentOn: define la representación HTML de una publicación.

```
[ TBPostComponent >> renderContentOn: html
    nivel de encabezado html: 2; con: título propio. nivel de
    encabezado html: 6; con: fecha propia. texto html: texto propio
```

Acerca de los formularios HTML

En un capítulo futuro sobre la vista de administración, mostraremos cómo usar Magritte para agregar descripciones a los objetos del modelo y luego usarlos para generar automáticamente componentes de Seaside. Este es un desarrollador poderoso y gratuito para describir formularios manualmente en Seaside.

Para darle una idea de eso, aquí el código equivalente al anterior usando Magritte:

```
[ TBPostComponent >> renderContentOn: html
    "NO ESCRIBIR ESTO TODAVÍA"
    representación html: publicar como componente
```

6.11 Mostrar publicaciones

Antes de mostrar las publicaciones disponibles en la base de datos, debe verificar que su blog contenga algunas publicaciones:

```
[ TBBlog actual allBlogPosts tamaño
```

Si no contiene publicaciones, puede recrear algunas:

```
[ TBBlog createDemoPosts
```

Ahora, solo necesitamos modificar el método TBPostsListComponent >> renderContentOn: para mostrar todas las publicaciones visibles en la base de datos:

```
[ TBPostsListComponent >> renderContentOn: html super renderContentOn:
    html. autoblog allVisibleBlogPosts hacer: [ :p |
        representación html: (TBPostComponent nueva publicación: p)]
```

Actualice su navegador web y debería obtener un error.

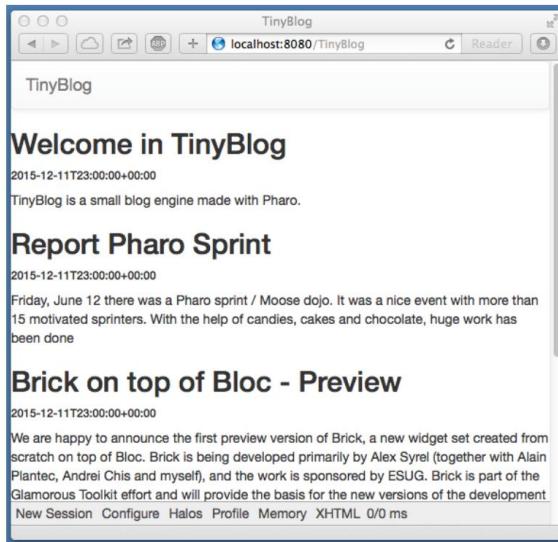


Figura 6-9 TinyBlog con una lista de publicaciones.

6.12 Errores de depuración

De forma predeterminada, cuando se produce un error en una aplicación web, Seaside devuelve una página HTML con el mensaje de error. Puede cambiar este mensaje o, durante el desarrollo, puede configurar Seaside para abrir un depurador directamente en Pharo IDE. Para configurar Seaside, simplemente ejecute el siguiente fragmento:

```
(WAAdmin defaultDispatcher handlerAt: 'TinyBlog')
    manejador de excepciones: WADebugErrorHandler
```

Ahora, si actualiza la página web en su navegador, se debería abrir un depurador en el lado de Pharo. Si analiza la pila, debería ver que olvidamos definir el siguiente método:

```
TBPostComponent >> publicación: aPost
    publicación := una publicación
```

Puede definir este método en el depurador mediante el botón Crear . Despues de eso, presione el botón Continuar . La aplicación web ahora debería representar correctamente lo que se muestra en la Figura 6-9.

6.13 Visualización de la lista de publicaciones con Bootstrap

Usemos Bootstrap para hacer que la lista de publicaciones sea más hermosa usando un contenedor Bootstrap gracias al mensaje tbsContainer::

6.14 Creación de instancias de componentes en renderContentOn:

```
TBPostsListComponent >> renderContentOn: html super
renderContentOn: html. html tbsContainer: [
    autoblog allVisibleBlogPosts hacer: [ :p | representación
        html: (TBPostComponent nueva publicación: p) ]]
```

Su aplicación web debería parecerse a la Figura 6-2.

6.14 Creación de instancias de componentes en renderContentOn:

Explicamos que el método child de un componente debe devolver sus subcomponentes. De hecho, antes de ejecutar el método renderContentOn: de un compuesto, Seaside necesita recuperar todos sus subcomponentes y su estado. Sin embargo, si se crean instancias de subcomponentes en el método renderContentOn: del compuesto (como en TBPostsListComponent>>renderContentOn:), no es necesario que los elementos secundarios devuelvan esos subcomponentes.

Tenga en cuenta que crear instancias de subcomponentes en el método de representación no es una buena práctica, ya que aumenta el tiempo de carga de la página web.

Si almacenaríamos todos los subcomponentes que muestran publicaciones, deberíamos agregar una variable de instancia postComponents.

```
TBPostsListComponent >> inicializar
súper inicializar.
postComponents := OrderedCollection nuevo
```

Inicialízalo con publicaciones.

```
TBPostsListComponent >> postComponents
postComponents := self readSelectedPosts recopilar:
    [ :each | Nueva publicación de TBPostComponent: cada uno].
postComponents
```

Redefina el método de los niños y, por supuesto, represente estos subcomponentes en renderContentOn::

```
TBPostsListComponent >> niños
self postComponents, super niños

TBPostsListComponent >> renderContentOn: html super
renderContentOn: html. html tbsContainer: [ self
postComponents do: [ :p | representación html: p ]]
```

No hacemos esto en TinyBlog porque hace que el código sea más complejo.

6.15 Conclusión

En este capítulo, desarrollamos un componente Seaside que genera una lista de publicaciones. En el próximo capítulo, mejoraremos esto al mostrar las categorías de las publicaciones.

Tenga en cuenta que no nos importaron las solicitudes web o el estado de la aplicación. Un programador de Seaside solo define componentes y los compone como lo haríamos en aplicaciones de escritorio.

Un componente Seaside es responsable de renderizarse redefiniendo su método `renderContentOn:`. También debería devolver sus subcomponentes (si no se instancian durante cada representación) redefiniendo el método de los niños .

Gestión de categorías

En este capítulo, agregamos la posibilidad de ordenar las publicaciones en una categoría. La Figura 7-1 le muestra en qué componentes trabajaremos en este capítulo.

Puede encontrar instrucciones para cargar el código del capítulo anterior en el Capítulo 10.

7.1 Mostrar publicaciones por categoría

Las publicaciones se ordenan por categoría. Si no se especifica ninguna categoría, las publicaciones se ordenan en una categoría especial llamada "Sin clasificar". Para administrar una lista de categorías, definiremos un componente llamado TBCategoriesComponent.

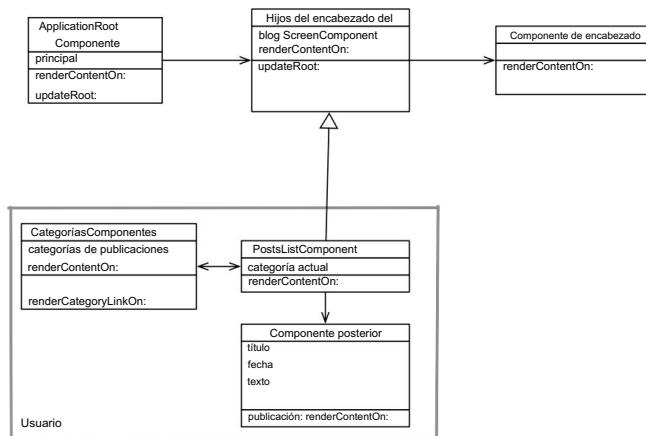


Figura 7-1 La arquitectura de componentes de la parte pública con categorías.

Visualización de categorías

Necesitamos un componente para mostrar una lista de categorías definidas en el blog. Este componente debe apoyar la selección de una categoría. Este componente debería poder comunicarse con el componente `TBPostsListComponent` para darle la categoría seleccionada actualmente. La figura 7-1 describe la situación.

Recuerde que una categoría se expresa simplemente como una cadena en el modelo que definimos en el Capítulo 2 y cómo lo ilustra la siguiente prueba:

```
testAllBlogPublicacionesDeCategoría
autoafirmación: (blog allBlogPostsFromCategory: 'First Category') el tamaño es igual a: 1
```

Definición de componente

Definamos un nuevo componente denominado `TBCategoriesComponent`. Mantiene una colección ordenada de cadenas que representan cada categoría, así como una referencia al componente que administra la lista de publicaciones.

```
Subclase WACOMPONENT: #TBCategoriesComponent
    instanciaVariableNames: 'categorías publicacionesLista'
    claseVariableNames: ''
    paquete: 'Componentes TinyBlog'
```

Definimos los accesores asociados.

```
TBCategoriesComponent >> categorías categorías
TBCategoriesComponent >> categorías: aCollection categorías := aCollection
    asSortedCollection
TBCategoriesComponent >> postsList: aComponent postsList := aComponent
TBCategoriesComponent >> publicacionesLista
    Lista de publicaciones
```

Definimos un método de creación como un método de clase.

```
TBCategoriesComponent class >> categorías: categorías postsList:
    ^ aTBSPantalla
        self nuevas categorías: categorías; Lista de publicaciones: aTBScreen
```

De la lista de publicaciones

En la clase `TBPostsListComponent`, necesitamos agregar una variable de instancia para almacenar la categoría actual.

7.2 Representación de categorías

```

[ Subclase TBScreenComponent: #TBPostsListComponent
  nombres de variables de instancia: 'categoría actual'
  nombres de variables de clase: "
  paquete: 'Componentes TinyBlog'
```

Definimos sus accesores asociados.

```

[ TBPostsListComponent >> categoría actual
  ^ categoría actual

[ TBPostsListComponent >> categoría actual: unObjeto
  ^ categoría actual := unObjeto
```

El método selectCategory: Definimos

el método selectCategory: (protocolo 'acciones') para comunicar la categoría actual al componente TBPostsListComponent .

```

[ TBCategoriesComponent >> selectCategory: aCategory postsList
  ^ currentCategory: aCategory
```

7.2 Representación de categorías

Ahora podemos definir el método para la representación del componente de categoría en la página. Llamémoslo renderCategoryLinkOn:with:, definimos en particular que hacer clic en una categoría la seleccionará como la actual. Usamos una devolución de llamada (mensaje de devolución de llamada:). El argumento de este mensaje es un bloque que puede contener cualquier expresión de Pharo. Esto ilustra lo simple que es llamar a la función para reaccionar al evento.

```

[ TBCategoriesComponent >> renderCategoryLinkOn: html with: aCategory html
  ^ tbsLinkifyListGroupItem callback: [self selectCategory: aCategory]; con: una categoría
```

El método renderContentOn: de TBCategoriesComponent es simple: iteramos en todas las categorías y las mostramos usando Bootstrap.

```

[ TBCategoriesComponent >> renderContentOn: html html
  ^ tbsListGroup: [ html tbsListGroupItem con: [ html strong:
    ^ 'Categorías']. categorías hacer: [ :gato |
      self renderCategoryLinkOn: html con: cat ] ]
```

Estamos casi allí. Necesitamos mostrar la lista de categorías y actualizar las publicaciones según la categoría actual.

7.3 Actualización de la lista de publicaciones

Ahora debemos actualizar la lista de publicaciones. Modificamos el método de renderizado del componente `TBPostsListComponent`.

El método `readSelectedPosts` recopila las publicaciones que se mostrarán. Los filtra según la categoría actual. Cuando la categoría actual es nula, significa que el usuario aún no seleccionó una categoría. Por lo tanto, mostramos todas las publicaciones. Cuando la categoría actual es diferente a cero, el usuario selecciona una categoría y la aplicación muestra las publicaciones correspondientes.

```
TBPostsListComponent >> readSelectedPosts self
  ^ currentCategory ifNil: [self blog
    allVisibleBlogPosts] ifNotNil: [self blog
    allVisibleBlogPostsFromCategory: self currentCategory]
```

Modificamos ahora el método responsable del renderizado de la lista de publicaciones:

```
TBPostsListComponent >> renderContentOn: html
  super renderContentOn: html.
  representación html: (TBCategoriesComponent
    categorías: (self blog allCategories) postsList:
    self). html tbsContainer: [ self readSelectedPosts
  do: [ :p |
    representación html: (TBPostComponent nueva publicación: p) ]]
```

Se agrega una instancia del componente `TBCategoriesComponent` a la página y permite seleccionar la categoría actual (consulte la Figura 7-2).

Como se explicó anteriormente, se crea una nueva instancia de `TBCategoriesComponent` cada vez que se procesa el componente `TBPostsListComponent`, por lo que no es obligatorio agregarlo a la sublista secundaria del componente.

Posibles mejoras

Codificar el nombre de la clase y la lógica de creación de categorías y publicaciones no es realmente óptimo. Proponga alguna solución.

7.4 Aspecto y diseño

No ubicaremos mejor el componente `TBPostsListComponent` usando un diseño más 'receptivo' (como se muestra en la Figura 7-3). Significa que el estilo CSS debe adaptar el componente al espacio disponible.

Los componentes se colocan en un contenedor Bootstrap y luego se colocan en una línea con dos columnas. La dimensión de la columna se determina según el puerto de visualización y la resolución del dispositivo utilizado. Las 12 columnas de Bootstrap se distribuyen en las listas de categorías y publicaciones.

7.4 Aspecto y diseño

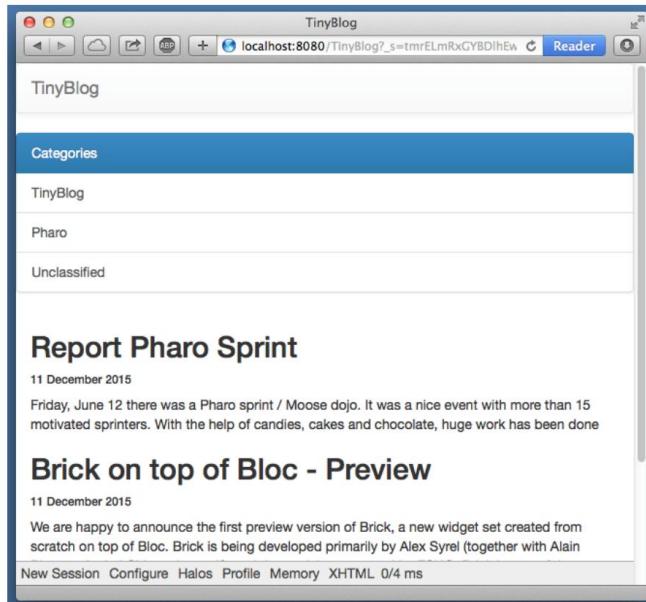


Figura 7-2 Categorías y publicaciones.

En el caso de una resolución baja, la lista de categorías se coloca encima de la lista de publicaciones (cada elemento ocupa el 100 % del ancho del contenedor).

```
TBPostsListComponent >> renderContentOn: html
    súper renderContentOn: html. html
    tbsContainer: [
        html tbsRow showGrid; con:
        [ html tbsColumn

            tamaño extrapequeño:
            12; pequeñoTamaño: 2;
            tamaño mediano: 4; con:
            [ representación html:
                (TBCategoriesComponent
                    categorías: (self blog allCategories) postsList: self)].
                html tbsColumna

            tamaño extrapequeño:
            12; pequeñoTamaño: 10;
            tamaño mediano: 8; con:
            [ self readSelectedPosts
                do: [ :p | representación html:
                    (TBPostComponent nueva publicación: p) ]]]]
```

Debería obtener una situación cercana a la presentada en la Figura 7-3.

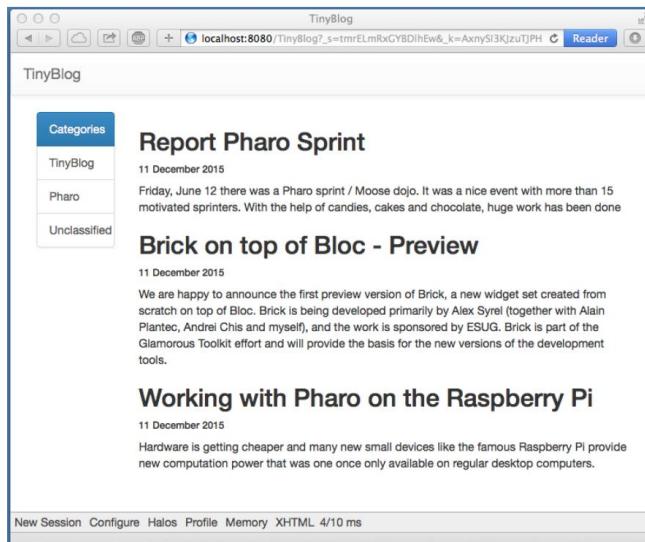


Figura 7-3 Lista de publicaciones con un mejor diseño.

Cuando uno selecciona una categoría, la lista de publicaciones se actualiza. Sin embargo, la categoría seleccionada no está seleccionada. Modificamos el siguiente método para abordar este punto.

```
TBCategoriesComponent >> renderCategoryLinkOn: html con: aCategory html
    tbsLinkifyListGroupItem
        class: 'active' if: aCategory = self postsList currentCategory; devolución de llamada: [self
            selectCategory: aCategory]; con: una categoría
```

Incluso si el código funciona, no podemos mantener el método `renderContentOn:` en tal estado. Es demasiado largo y no reutilizable. Proponga una solución.

7.5 Código modular con métodos pequeños

Aquí está nuestra solución al problema anterior. Para facilitar la lectura y la futura reutilización, comenzamos a definir métodos de creación de componentes.

```
TBPostsListComponent >> categoríasComponente
    TBCategoriesCategorías de
        componentes: self blog allCategories postsList:
        self

TBPostsListComponent >> postComponentFor: aPost
    ^ TBPostComponent nueva publicación: aPost
```

7.5 Código modular con métodos pequeños

```

TBPostsListComponent >> renderContentOn: html super
    renderContentOn: html. html

        tbsContainer: [ html tbsRow
            showGrid; con: [ html tbsColumn

                tamaño extrapequeño:
                12; pequeñoTamaño: 2;
                tamaño mediano: 4; con:
                [ html render: selfcategoriesComponent ]. html tbsColumnna

                tamaño extrapequeño:
                12; pequeñoTamaño:
                10; tamaño mediano: 8;
                con: [ self readSelectedPosts do: [ :p |
                    representación html: (autopostComponentFor: p)]]
            ]]]
```

otro pase

Seguimos cortando este método en otros métodos más pequeños. Creamos un método para cada una de las tareas elementales.

```

TBPostsListComponent >> basicRenderCategoriesOn: html html
    render: selfcategoriesComponent

TBPostsListComponent >> basicRenderPostsOn: html self
    readSelectedPosts do: [ :p | representación html:
        (autopostComponentFor: p)]
```

Luego usamos dichas tareas para simplificar el método renderContentOn::

```

TBPostsListComponent >> renderContentOn: html
    super renderContentOn: html. html
    tbsContainer: [ html tbsRow

        Mostrar
        cuadrícula; con: [ self renderCategoryColumnOn: html.
            self renderPostColumnOn: html ]]

TBPostsListComponent >> renderCategoryColumnOn: html
    html tbsColumnna
        tamaño extrapequeño:
        12; pequeñoTamaño: 2;
        tamaño mediano: 4; con:
        [ self basicRenderCategoriesOn: html ]
```

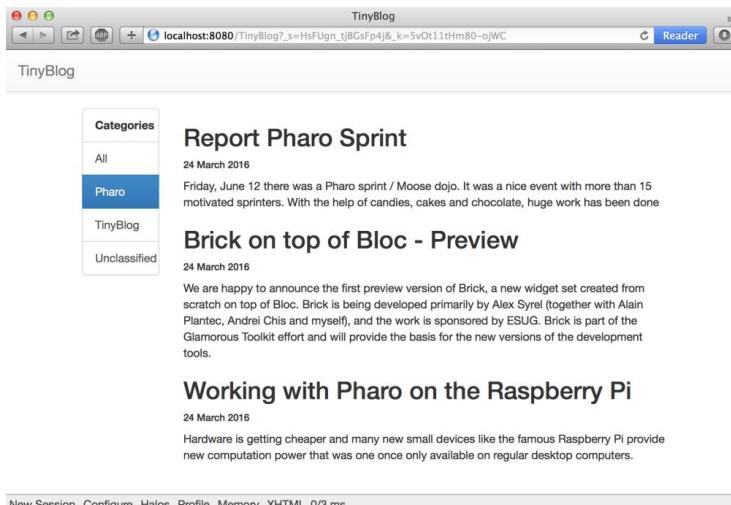


Figura 7-4 Interfaz de usuario pública final de TinyBlog.

```
TBPostsListComponent >> renderPostColumnOn: html
    html tbsColumna
        tamaño extrapequeño:
        12; pequeñoTamaño:
        10; tamaño mediano: 8;
        con: [ self basicRenderPostsOn: html ]
```

La aplicación final se muestra en la Figura 7-4.

7.6 Conclusión

Definimos una interfaz para nuestro blog utilizando un conjunto de componentes, cada uno especificando su propio estado y responsabilidad. Muchas aplicaciones web se construyen de la misma manera reutilizando componentes. Tiene la base para crear una aplicación web más avanzada.

En el próximo capítulo, mostramos cómo administrar la autenticación para acceder a la parte de administración posterior de nuestra aplicación.

Posibles mejoras

Como ejercicio puedes:

- ordenar la categoría alfabéticamente, o •

agregar un enlace llamado 'Todos' en la lista de categorías para mostrar todas las publicaciones.

CAPÍTULO 8

Autenticación y Sesión

En este capítulo desarrollaremos un escenario tradicional: el usuario debe iniciar sesión para acceder a la parte de administración de la aplicación. Lo hace usando un nombre de usuario y una contraseña.

La figura 8-1 muestra la arquitectura a la que llegaremos en este capítulo.

Comencemos a implementar una primera versión que permita navegar entre la parte de TinyBlog presentada por el componente TBPostsListComponent y un primer borrador del componente de administración como se muestra en la Figura 8-2.

Esto ilustra cómo invocar un componente.

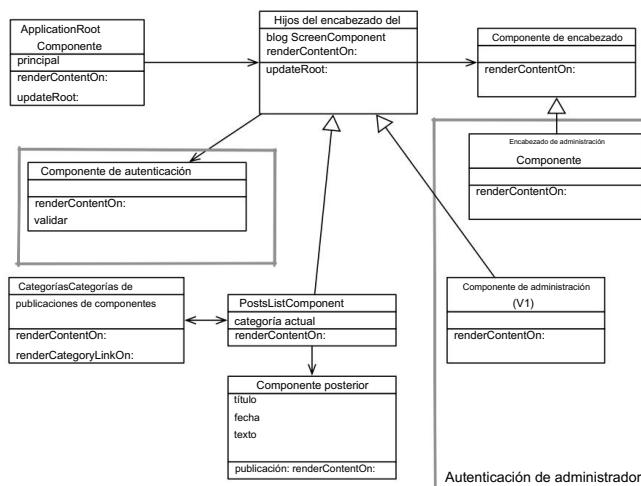


Figura 8-1 Flujo de autenticación.

A continuación, construiremos e integraremos un componente que gestione el inicio de sesión en función de la interacción modal. Esto ilustrará cómo podemos mapear elegantemente entradas archivadas a variables de instancia de un componente.

Finalmente, mostraremos cómo se almacena la información del usuario en la sesión actual.

8.1 Un componente de administración simple (v1)

Definamos un componente de administración realmente súper simple. Este componente hereda de la clase TBScreenComponent como se mencionó en capítulos anteriores y se ilustra en la Figura 8-1.

```
Subclase TBScreenComponent: #TBAdminComponent
    nombres de variables de instancia: "
        Nombres de variables de clase: "
            paquete: 'Componentes TinyBlog'
```

Definimos una primera versión del método de renderizado para poder probar nuestro enfoque.

```
TBAdminComponent >> renderContentOn: html super
    renderContentOn: html. html tbsContainer: [título
        html: 'Administrador del blog'. html reglahorizontal]
```

8.2 Añadir el botón 'admin'

Agregamos ahora un botón en el encabezado del sitio (componente TBHeaderComponent) para que el usuario pueda acceder al administrador como se muestra en la Figura 8-2. Para ello, modificamos los componentes existentes: TBHeaderComponent (encabezado) y TBPostsListComponent (parte pública).

Agreguemos un botón en el encabezado:

```
TBHeaderComponent >> renderContentOn: html html
    tbsNavbar beDefault; con: [ html tbsContainer:
        [ self renderBrandOn: html.

        self renderButtonsOn: html
    ]]
```

```
TBHeaderComponent >> renderButtonsOn: html self
    renderSimpleAdminButtonOn: html
```

```
TBHeaderComponent >> renderSimpleAdminButtonOn: html
    formulario html: [ html tbsNavbarButton
        tbsPullDerecha;
```

8.2 Añadir el botón 'admin'

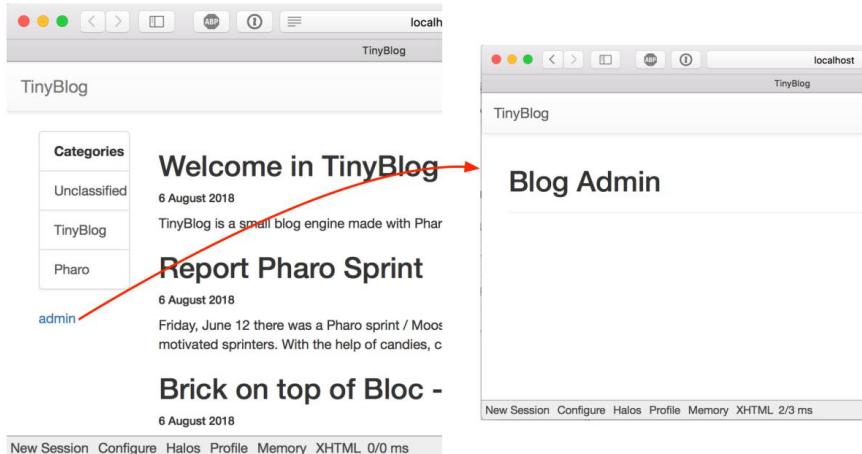


Figura 8-2 Enlace simple a la parte de administración.

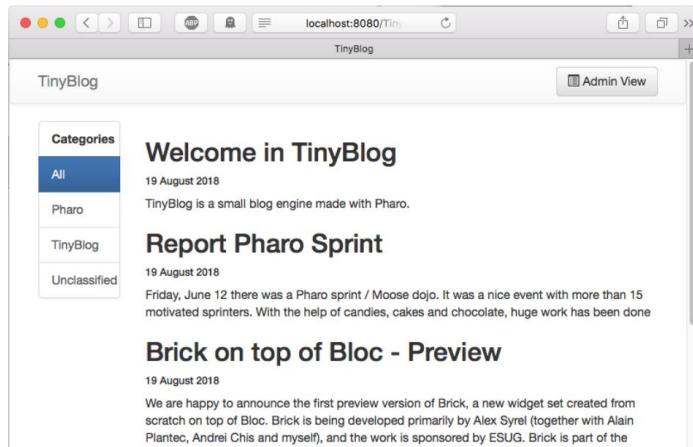


Figura 8-3 Encabezado con un botón de administración.

```
con: [ html
      tbsGlyphIcon iconListAlt.
      administrador' ] ] 'tarea.html'
```

Cuando actualiza el navegador web, el botón de administración está presente pero no tiene ningún efecto (consulte la Figura 8-3).

Deberíamos definir una devolución de llamada en este botón (mensaje de devolución de llamada:) para reemplazar el componente actual (TBPostsListComponent) por el componente de administración (TBAdminComponent).

8.3 Revisión de encabezado

Revisemos la definición de TBHeaderComponent agregando una nueva variable de instancia llamada componente para almacenar y acceder al componente actual (ya sea una lista de publicaciones o un componente de administración). Esto nos permitirá acceder al componente desde la cabecera.

```
Subclase WAComponent: #TBHeaderComponent nombres de
variables de instancia: 'componente' nombres de variables
de clase: "
paquete: 'Componentes TinyBlog'
```

```
TBHeaderComponent >> componente: unObjeto
componente := unObjeto
```

```
TBHeaderComponent >> componente
^ componente
```

Agregamos un nuevo método de clase.

```
Clase TBHeaderComponent >> de: aComponent self nuevo
^
componente: unComponente; tú
mismo
```

8.4 Activación del botón de administración

Modificamos la creación de instancias del componente en el método TBScreenComponent para pasar el componente que estará debajo del encabezado.

```
Componente de pantalla TBS >> crear componente de encabezado
^ TBHeaderComponent de: self
```

Tenga en cuenta que el método createHeaderComponent está definido en la superclase TBScreenComponent, y es aplicable a todas las subclases.

Podemos agregar ahora la devolución de llamada en el botón:

```
TBHeaderComponent >> renderSimpleAdminButtonOn: html formulario html:
[ html tbsNavbarButton

    tbsPullDerecha;
    devolución de llamada: [componente goToAdministrationView]; con: [
        html tbsGlyphIcon iconListAlt.
        texto html:      'Vista de administrador' ]]
```

Solo necesitamos definir el método goToAdministrationView en el componente TBPostsListComponent:

```
TBPostsListComponent >> ir a Vista de administración
llamada automática: TBAdminComponent nuevo
```

8.5 Adición del botón 'desconectar'

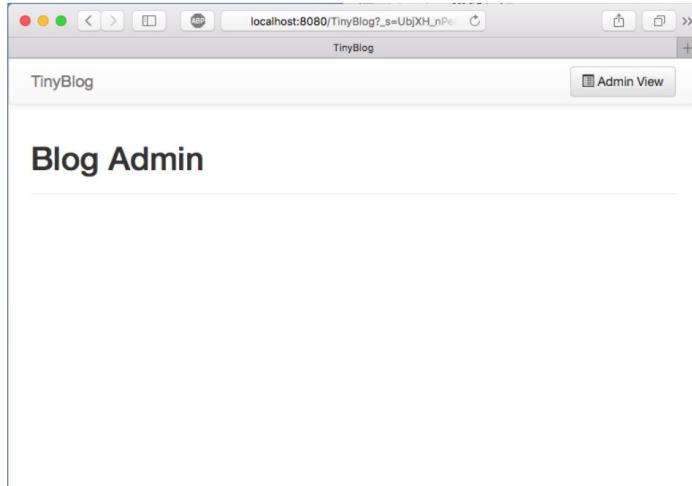


Figura 8-4 Componente de administración bajo definición.

Antes de hacer clic en el botón de administración, debe renovar la sesión actual haciendo clic en 'Nueva sesión': volverá a crear el componente TBHeaderComponent.

Debería obtener la situación presentada en la Figura 8-4. El botón 'Administrador' permite acceder a la parte de administración v1.

Preste atención a no hacer clic dos veces en el botón de administración porque aún no lo manejamos para la parte de administración. Lo reemplazaremos por un botón Desconectar.

8.5 Adición del botón 'desconectar'

Cuando mostramos la parte de administración, reemplazaremos el componente de encabezado por uno nuevo. Este nuevo encabezado mostrará un botón de desconexión.

Definamos este nuevo componente de encabezado:

```

Subclase TBHeaderComponent: #TBAdminHeaderComponent
  nombres de variables de instancia: "
    Nombres de variables de clase: "
  paquete: 'Componentes TinyBlog'

TBAdminHeaderComponent >> renderButtonsOn: html
  formulario html: [self renderDisconnectButtonOn: html]

```

El componente TBAdminComponent debe usar este encabezado:

```

TBAdminComponent >> createHeaderComponent
  ^ TBAdminHeaderComponent de: self

```

Ahora podemos especializar nuestro nuevo encabezado de administración para mostrar un botón de desconexión.

```
TBAdminHeaderComponent >> renderDisconnectButtonOn: html
    html tbsNavbarButton
        tbsPullDerecha;
        devolución de llamada: [componente goToPostListView]; con:
        [texto html: 'Desconectar'. html tbsGlyphIcon iconCerrar sesión]
```

```
Componente TBAdmin >> ir a la vista de la lista de publicaciones
    auto respuesta
```

Lo que se ve es que el mensaje respuesta devuelve el control al componente que lo llama. Así que volvemos a la lista de publicaciones.

Restablezca la sesión actual haciendo clic en 'Nueva sesión'. Luego puede hacer clic en el botón 'Administrador', debería ver ahora que el administrador v1 se muestra con un botón 'Desconectar'. Este botón permite volver a la parte pública como se muestra en la Figura 8-2.

llamada:/respuesta: Noción

Cuando estudias el código anterior, ves que usamos el mecanismo call:/answer: de Seaside para navegar entre los componentes TBPostsListComponent y TBAdminComponent.

La llamada de mensaje: reemplaza el componente actual con el pasado en el argumento y le da el flujo de control. La respuesta del mensaje: devuelve un valor a esta llamada y devuelve el flujo de control al argumento de la llamada.

Este mecanismo es realmente potente y elegante. Está explicado en el video 1 de la semana 5 del Pharo Mooc (http://rmod-pharo-mooc.lille.inria.fr/MOOC/WebPortal/co/content_5.html).

8.6 Ventana Modal para Autenticación

Desarrollaremos ahora un componente de autenticación que, cuando se invoque, abrirá un cuadro de diálogo para solicitar el nombre de usuario y la contraseña. El resultado que queremos obtener se muestra en la Figura 8-5.

Hay algunas bibliotecas de componentes listas para ser utilizadas. Por ejemplo, el proyecto Heimdal disponible en <http://www.github.com/DuneSt/> ofrece un componente de autenticación o el proyecto Steam <https://github.com/guilip/steam> ofrece formas de interrogar cuentas de google o twitter.

Definición del componente de autenticación

Definimos una nueva subclase de WAComponent y sus accesores. Este componente contiene un nombre de usuario, una contraseña y un componente que lo invocó para acceder a

8.6 Ventana Modal para Autenticación

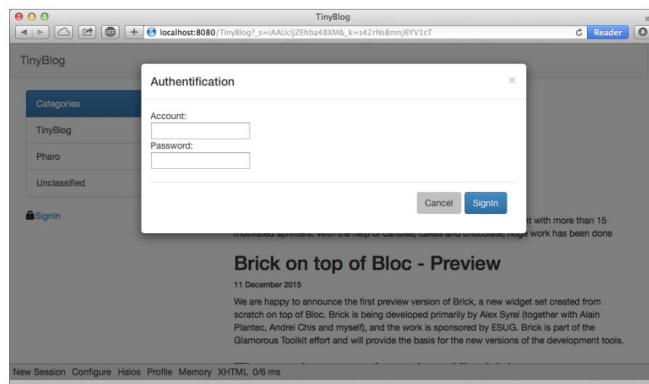
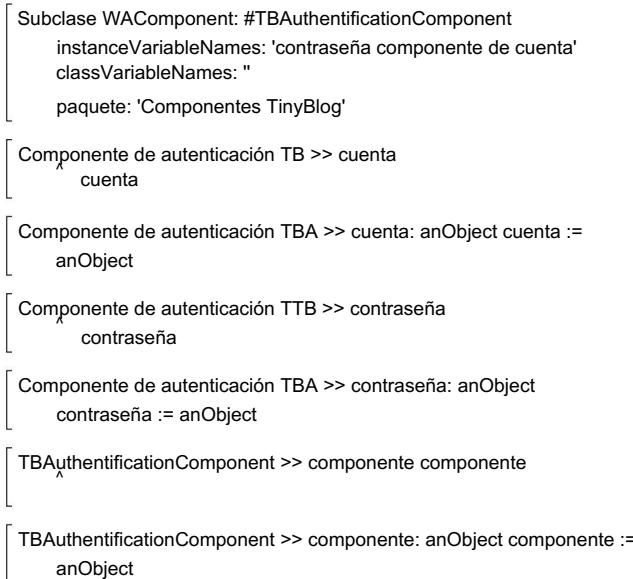
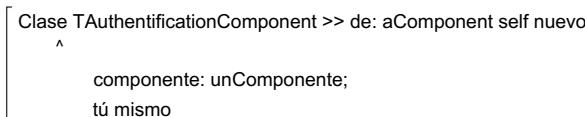


Figura 8-5 Componente de autenticación.

la parte de administración.



El componente de variable de instancia se inicializará mediante el siguiente método de clase: classe suivante :



8.7 Representación del componente de autenticación

El siguiente método renderContentOn: define el contenido de un cuadro de diálogo con el ID myAuthDialog. Este ID se usará para seleccionar el componente que debe hacerse visible en el modo modal.

Este cuadro de diálogo tiene un encabezado y un cuerpo. Tenga en cuenta el uso de los mensajes tb sModal, tbsModalBody: y tbsModalContent: que admite una interacción modal con el componente.

```
Componente de autenticación TBA >> renderContentOn: html html
  tbsModal id: 'myAuthDialog'; con: [ html tbsModalDialog: [
    ...
    html tbsModalContent: [
      self renderHeaderOn: html.
      self renderBodyOn: html ]]]
```

El encabezado muestra un botón para cerrar el cuadro de diálogo y un título con fuentes grandes. Tenga en cuenta que también puede usar la tecla ESC para cerrar el cuadro de ventana modal.

```
Componente de autenticación TBA >> renderHeaderOn: html
  html
    tbsModalHeader: [
      html tbsModalCloseIcon.
      html tbsModalTitle
        nivel 4;
        con: 'Autenticación' ]
```

El cuerpo del componente muestra el campo de entrada para el identificador de inicio de sesión, la contraseña y algunos botones.

```
Componente de autenticación TBA >> renderBodyOn: html
  html
    tbsModalBody:
      [ html tbsForm:
        [ self renderAccountFieldOn: html.
        self renderPasswordFieldOn: html.
        html tbsModalFooter: [self renderButtonsOn: html]
      ]]
```

El método renderAccountFieldOn: muestra cómo se pasa y almacena el valor de un campo de entrada en una variable de instancia de un componente cuando el usuario finaliza su entrada.

El parámetro de la devolución de llamada: mensaje es un bloque que toma como argumento el valor del campo de entrada.

8.8 Integración del componente de autenticación

```
Componente de autenticación TBA >> renderAccountFieldOn: html
html
  tbsFormGroup: [etiqueta html con: 'Cuenta'. html
    entrada de texto tbsFormControl; atributoAt:
      'enfoque automático' put: 'verdadero'; devolución
      de llamada: [ :valor | cuenta := valor ]; valor:
      cuenta]
```

El mismo proceso se utiliza para la contraseña.

```
Componente de autenticación TBA >> renderPasswordFieldOn: html
html tbsFormGroup:
  [etiqueta html con: 'Contraseña'.
  html contraseñaInput
    tbsFormControl;
    devolución de llamada: [ :valor | contraseña := valor ];
    valor: contraseña]
```

Finalmente, en el siguiente método renderContentOn:, se agregan dos botones en la parte inferior de la ventana modal. El botón 'Cancelar' que permite cerrar la ventana usando el atributo 'descartar datos' y el botón 'Iniciar sesión' que envía la validación usando una devolución de llamada.

La tecla Intro está vinculada a la activación del botón 'Iniciar sesión' cuando se usa el método tbsSubmitButton. Este método establece el atributo 'tipo' en 'enviar'.

```
Componente de autenticación TBA >> renderButtonsOn: html html
tbsButton
  atributoEn: 'tipo' poner: 'botón'; atributoAt:
  'descartar datos' put: 'modal'; serPredeterminado;
  valor: 'Cancelar'.

html tbsSubmitButton
  serPrimario;
  devolución de llamada: [validación
  automática]; valor: 'Iniciar sesión'
```

En el método de validación , simplemente enviamos un mensaje al componente principal brindándole la información ingresada por el usuario.

```
TBAuthenticationComponent >> validar
  ^ componente tryConnectionWithLogin: autocuenta andPassword: autocontraseña
```

8.8 Integración del componente de autenticación

Para integrar nuestro componente de autenticación, modificamos el botón Admin del componente de encabezado (TBHeaderComponent) de la siguiente manera:

```

TBHeaderComponent >> renderButtonsOn: html self
    renderModalLoginButtonOn: html

TBHeaderComponent >> renderModalLoginButtonOn: html
    representación html: (TBAuthenticationComponent de: componente). html
    tbsNavbarButton tbsPullRight; atributoAt: 'objetivo de datos' put:
        '#myAuthDialog'; atributoAt: 'cambio de datos' put: 'modal'; con: [ html
        tbsGlyphIcon iconLock. texto html: 'Iniciar sesión']

```

El método `renderModalLoginButtonOn:` comienza renderizando el componente `TBAuthenticationComponent` dentro de esta página web. Este componente se crea durante cada visualización y no tiene que ser devuelto por el método de niños . Además, añadimos el botón 'Iniciar sesión' con un ícono de candado.

Cuando el usuario haga clic en este botón, se mostrará el diálogo modal identificado con el ID `myAuthDialog` .

Al volver a cargar la página de TinyBlog, debería ver ahora un botón 'Iniciar sesión' en el encabezado (botón que mostrará la autenticación que acabamos de desarrollar) como se ilustra en la Figura 8-5.

8.9 Administrar ingenuamente los inicios de sesión

Al hacer clic en el botón 'Iniciar sesión' aparece un error. Usando el Pharo de bugger, puede ver que debemos definir el método `tryConnectionWithLogin:andPassword:` en el componente `TBPostsListComponent` ya que es el que envía la devolución de llamada del botón.

```

TBPostsListComponent >> tryConnectionWithLogin: inicio de sesión y contraseña:
    contraseña
    (login = 'admin' and: [ password = 'topsecret' ]) ifTrue: [self
        goToAdministrationView] ifFalse: [self loginErrorOccurred]

```

Por el momento almacenamos directamente el nombre de usuario y la contraseña en el método y esto no es realmente una buena práctica.

8.10 Gestión de errores

Definimos el método `goToAdministrationView`. Agreguemos el método `loginErrorOccurred` y un mecanismo para mostrar un mensaje de error cuando el usuario no usa los identificadores correctos como se muestra en la Figura 8-6.

Para ello añadiremos una nueva variable de instancia `showLoginError` que representa el hecho de que deberíamos mostrar un error.

8.11 Modelado del administrador

```

[ Subclase TBScreenComponent: #TBPostsListComponent
  instanciaVariableNames: 'categoría actual showLoginError'
  classVariableNames: ''
  paquete: 'Componentes TinyBlog'
]

```

El método loginErrorOccurred especifica que se debe mostrar un error.

```

[ TBPostsListComponent >> loginErrorOccurred
  showLoginError := verdadero
]

```

Agregamos un método para probar este estado.

```

[ TBPostsListComponent >> hasLoginError
  ^ showLoginError ifNil: [falso]
]

```

Definimos también un mensaje de error.

```

[ TBPostsListComponent >> loginErrorMessage
  'Inicio de sesión y/o contraseña incorrectos'
]

```

Modificamos el método renderPostColumnOn: para realizar una tarea específica para manejar los errores.

```

[ TBPostsListComponent >> renderPostColumnOn: html
  html tbsColumn
    extraSmallSize: 12;
    pequeñoTamaño: 10;
    tamaño mediano: 8; con:
    [ self
      renderLoginErrorMessageIfAnyOn: html. self
      basicRenderPostsOn: html ]
]

```

El método renderLoginErrorMessageIfAnyOn: muestra si es necesario un mensaje de error. Establece la variable de instancia showLoginError para que no mostremos el error indefinidamente.

```

[ TBPostsListComponent >> renderLoginErrorMessageIfAnyOn: html self
  hasLoginError ifTrue: [ showLoginError := false. html tbsAlerta
    bePeligro ; con:
    mensaje de error de inicio de sesión automático
  ]
]

```

8.11 Modelado del administrador

No queremos almacenar los identificadores de administrador en el código como lo hicimos anteriormente.

Revisamos esto ahora y almacenaremos los identificadores en un modelo: un administrador de clase.

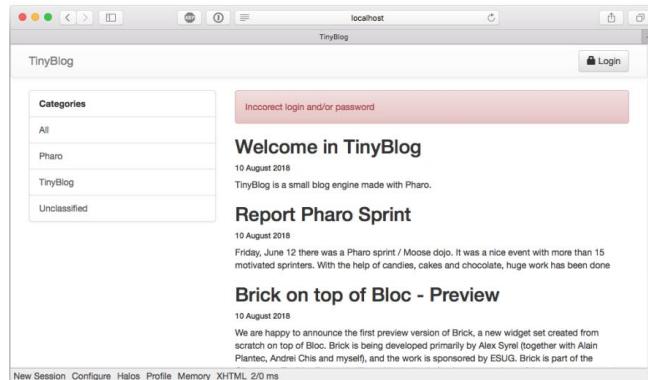


Figura 8-6 Mensaje de error en caso de identificadores incorrectos.

Comencemos a enriquecer nuestro modelo TinyBlog con la noción de administrador.

Definimos una clase denominada TBAdministrator caracterizada por su pseudo, login y contraseña.

Subclase de objeto: #TBAdministrator
 instanciaVariableNames: 'contraseña de inicio de sesión'
 classVariableNames: ""
 paquete: 'TinyBlog'

TBAdministrator >> iniciar sesión iniciar
 sesión

TBAdministrator >> inicio de sesión: anObject
 iniciar sesión := unObjeto

TBAdministrator >> contraseña contraseña

Tenga en cuenta que no almacenamos la contraseña de administrador en la contraseña de la variable de instancia , sino su hash codificado en SHA256.

TBAdministrator >> contraseña: anObject
 contraseña := SHA256 hashMensaje: unObjeto

Definimos también un nuevo método de creación de instancias.

Clase TBAdministrator >> inicio de sesión: contraseña de inicio de sesión: contraseña
 auto nuevo
 iniciar sesión: iniciar
 sesión; contraseña:
 contraseña; tú mismo

Puedes verificar que el modelo funciona ejecutando la siguiente expresión:

[luc := TBIinicio de sesión del administrador: 'luc' contraseña: 'topsecret'.

8.12 Administrador de blogs

8.12 Administrador de blogs

Decidimos por simplicidad que un blog tiene un administrador. Agregamos la variable de instancia adminUser y un acceso en la clase TBBlog para almacenar el blog admin.

```

Subclase de objeto: #TBBlog
  instanceVariableNames: 'adminUser posts'
  classVariableNames: ''
  paquete: 'TinyBlog'

TBBlog >> administrador adminUser

```

Definimos un nombre de usuario y una contraseña predeterminados que usamos por defecto. Como veremos más adelante, modificaremos dichos atributos y estos atributos modificados se guardarán al mismo tiempo que el blog en una base de datos.

```

Clase TBBlog >> defaultAdminPassword 'topsecret'

Clase TBBlog >> defaultAdminLogin 'admin'

```

Ahora creamos un administrador predeterminado.

```

TBBlog >> crear Administrador
  ^ Administrador de TB
    inicio de sesión: autoclase defaultAdminLogin
    contraseña: autoclase defaultAdminPassword

```

E inicializamos el blog para establecer un administrador predeterminado.

```

TBBlog >> inicializar super
  inicializar. publicaciones:
    = OrderedCollection nuevo. adminUser := self
      createAdministrator

```

8.13 Configuración de un nuevo administrador

No debemos recrear el blog:

```

  Restablecimiento de TBBlog: crearDemoPublicaciones

```

Ahora podemos modificar la información del administrador de la siguiente manera:

```

  |
  administrador| admin := Administrador actual de TBBlog.
  inicio de sesión de administrador: 'lucas'. contraseña de
  administrador: 'thebrightside'.
  TBBlog guardado actual

```

Tenga en cuenta que, sin hacer nada, Voyage ha guardado la información del administrador del blog en la base de datos. De hecho, la clase TBBlog es una raíz de Voyage,

todos sus atributos se almacenan automáticamente en la base de datos cuando recibe el mensaje de guardar.

Posibles mejoras

Defina algunas pruebas para las extensiones escribiendo nuevas pruebas unitarias.

8.14 Integración de la información de administración

Modifiquemos el método `tryConnectionWithLogin:andPassword:` para que use los identificadores de administrador del blog actuales. Tenga en cuenta que estamos comparando el hash SHA256 de la contraseña ya que no almacenamos la contraseña.

```
TBPostsListComponent >> tryConnectionWithLogin: inicio de sesión y contraseña:  
contraseña  
(inicio de sesión = inicio de sesión del administrador del blog y: [  
 (SHA256 hashMessage: contraseña) = contraseña del administrador del blog  
 propio]) ifTrue: [self goToAdministrationView] ifFalse: [self loginErrorOccurred]
```

8.15 Almacenamiento del administrador en la sesión actual

Con la configuración actual, cuando el administrador del blog quiere navegar entre la parte privada y la pública, debe volver a conectarse cada vez. Simplificaremos esta situación pero almacenaremos la información de administración actual en la sesión cuando la conexión sea exitosa.

Se asigna un objeto de sesión a cada instancia de la aplicación. Dicha sesión permite mantener información compartida y accesible entre los componentes.

Luego almacenaremos el administrador actual en una sesión y modificaremos los componentes para mostrar botones que admitan una navegación simplificada cuando el administrador esté registrado.

Cuando se desconecta explícitamente o cuando la sesión expira, eliminamos la sesión actual.

La Figura 8-7 muestra la navegación entre las páginas de TinyBlog.

8.16 Definición y uso de sesión específica

Comencemos a definir una subclase de `WASession` y asígnele el nombre `TBSession`.

Agregamos en esta nueva clase una variable de instancia que almacena el administrador actual.

8.16 Definición y uso de sesión específica

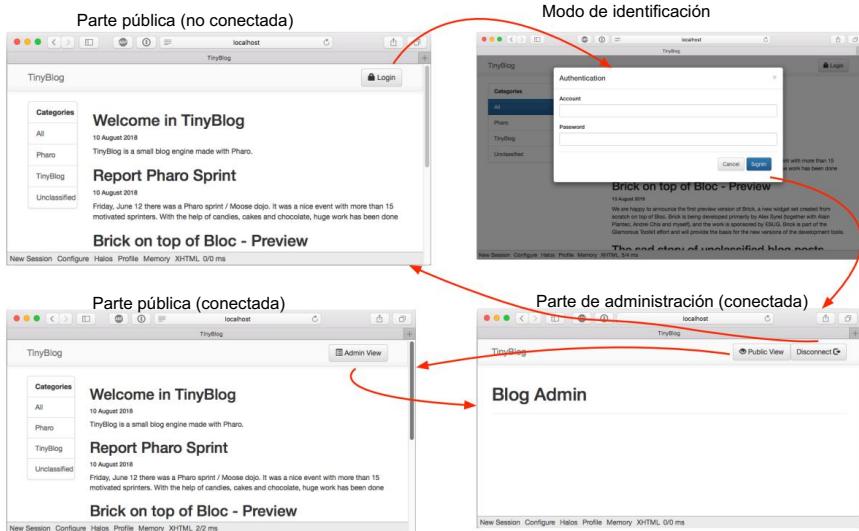


Figura 8-7 Navegación e identificación en TinyBlog.

```

Subclase WASESession: #TBSession
    nombres de variables de instancia: 'administrador actual'
    Nombres de variables de clase: ''
    paquete: 'Componentes TinyBlog'

```

```

TBSession >> administrador actual
    ^ administrador actual

```

```

TBSession >> administrador actual: anObject
    administrador actual := unObjeto

```

Definimos un método `isLogged` que permite saber si la administración está logueada.

```

TBSession >> isLogged
    ^ self currentAdmin notNil

```

Ahora debemos indicarle a Seaside que use `TBSession` como la clase de la sesión actual para nuestra aplicación. Esta inicialización se realiza en el método de clase `initialize` en la clase `TBApplicationRootComponent` de la siguiente manera:

```

Clase TBApplicationRootComponent >> inicializar "autoinicializar"
|
    aplicación | app := WAAdmin registro: self asApplicationAt: 'TinyBlog'.
|
    preferencia de aplicación en: #sessionClass put: TBSession.
|
    aplicación addLibrary: JQDeploymentLibrary;

```

```
| addLibrary: JQUiDeploymentLibrary; addLibrary:  
TBSDeploymentLibrary
```

No olvide ejecutar esta expresión TBApplicationRootComponent initialize antes de probar la aplicación.

8.17 Almacenamiento del administrador actual

Cuando una conexión es exitosa, agregamos el objeto de administración a la sesión actual usando el mensaje currentAdmin:. Tenga en cuenta que la sesión actual está disponible para todos los componentes de Seaside a través de la sesión propia.

```
TBPostsListComponent >> tryConnectionWithLogin: inicio de sesión y contraseña:  
contraseña  
(inicio de sesión = inicio de sesión del administrador del blog y: [  
(SHA256 hashMessage: contraseña) = contraseña del administrador del blog  
propio]) ifTrue: [  
  
sesión propia currentAdmin: administrador del blog propio. self  
goToAdministrationView ] ifFalse: [ self loginErrorOccurred ]
```

8.18 Navegación simplificada

Para implementar la navegación simplificada que discutimos anteriormente, modificamos el encabezado para mostrar un botón de inicio de sesión o un botón de navegación simple a la parte de administración sin forzar ninguna reconexión. Para ello usamos la sesión y el hecho de que podemos saber si un usuario está logueado.

```
TBHeaderComponent >> renderButtonsOn: html  
sesión propia isLoggedIn ifTrue:  
[ self renderSimpleAdminButtonOn: html ]  
ifFalse: [self renderModalLoginButtonOn: html]
```

Puede probar esta nueva navegación, pero primero cree una nueva sesión (botón 'Nueva sesión'). Uno reconectado, el administrador se agrega en la sesión. Tenga en cuenta que el botón de desconexión no funciona correctamente ya que invalida la sesión.

8.19 Gestión de la desconexión

Agregamos un restablecimiento de método en nuestro objeto de sesión para eliminar el administrador actual, invalidar la sesión actual y redirigir al punto de entrada de la aplicación.

8.20 Navegación simplificada a la parte pública

```
TBSession >> restablecer
    administrador actual := nil.
    self requestContext redirectTo: autoaplicación url. darse de baja por sí mismo.
```

Ahora modificamos el botón de desconexión del encabezado para enviar el mensaje de reinicio a la sesión correcta.

```
TBAdminHeaderComponent >> renderDisconnectButtonOn: html
    html tbsNavbarButton
        tbsPullDerecha;
        devolución de llamada: [restablecimiento de
        sesión automática]; con: [texto html: 'Desconectar'.
        html tbsGlyphIcon iconCerrar sesión]
```

Ahora el botón 'Desconectar' funciona como debería.

8.20 Navegación simplificada a la parte pública

Ahora podemos agregar un botón en el encabezado de la parte de administración para volver a la parte pública sin tener que desconectarnos.

```
TBAdminHeaderComponent >> renderButtonsOn: html
    formulario html: [
        self renderDisconnectButtonOn: html.
        self renderPublicViewButtonOn: html ]
```



```
TBAdminHeaderComponent >> renderPublicViewButtonOn: html
    sesión propia isLoggedIn ifTrue: [ html
        tbsNavbarButton
            tbsPullDerecha;
            devolución de llamada: [componente goToPostListView];
            con: [ html tbsGlyphIcon iconEyeOpen.
            html: 'Vista pública' ]] texto
```

Ahora puedes probar la navegación. Debería corresponder a la situación representada en la figura 8-7.

8.21 Conclusión

Implementamos una autenticación para TinyBlog. Creamos un componente modal reutilizable. Hicimos la distinción entre el componente que se muestra cuando un usuario está conectado o no y facilitamos la navegación de un usuario conectado usando la sesión.

Ahora estamos listos para la parte de administración de la aplicación y trabajaremos en esto en el próximo capítulo. Lo aprovecharemos para mostrar un aspecto avanzado: la generación automática de formularios.

Posibles mejoras

Puede:

- Agregar el inicio de sesión de administrador en el encabezado • Administrar cuentas de administración multipel.

CAPÍTULO 9

Interfaz web de administración y Generación Automática de Formularios

Ahora desarrollaremos la parte de administración de TinyBlog. En el capítulo anterior, definimos los componentes de Seaside que interactúan entre sí y donde cada componente es responsable de su estado interno, comportamiento y representación gráfica.

En este capítulo, queremos mostrar que podemos ir un paso más allá y generar componentes Seaside a partir de descripciones de objetos utilizando el marco Magritte.

La Figura 9-1 muestra una parte del resultado que obtendremos, siendo la otra parte la post edición.

La figura 9-2 muestra un resumen de la arquitectura que desarrollaremos en este capítulo.

9.1 Descripción de datos de dominio

Magritte es una biblioteca que permite generar varias representaciones una vez descritos los objetos. Junto con Seaside, Magritte genera formularios e informes. El Quuve de la editorial Debris es un ejemplo brillante del poder de Magritte: todas las tablas e informes se generan automáticamente (ver <http://www.pharo.org/success>).

La validación de datos también se realiza a nivel de Magritte en lugar de estar dispersos en el código de la interfaz de usuario. Este capítulo no cubrirá tales aspectos. Los recursos sobre Magritte son un capítulo del libro Seaside (<http://book.seaside.st>) así como un tutorial bajo escritura disponible en <https://github.com/SquareBracketAssociates/Magritte>.

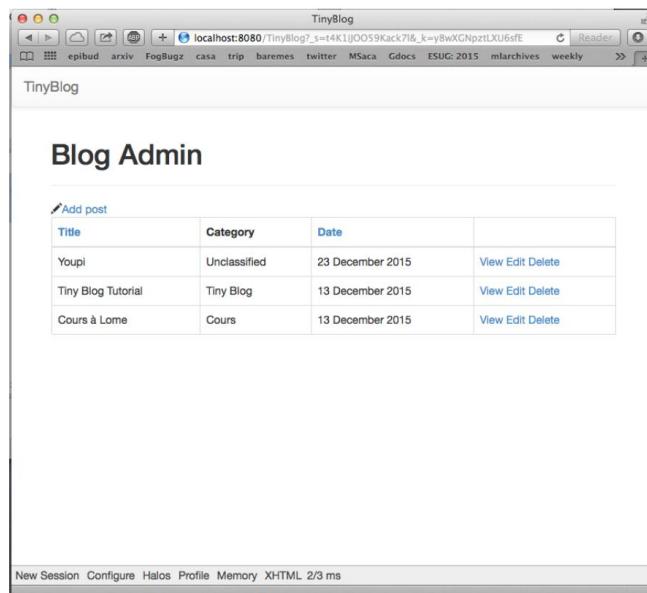


Figura 9-1 Gestión de publicaciones.

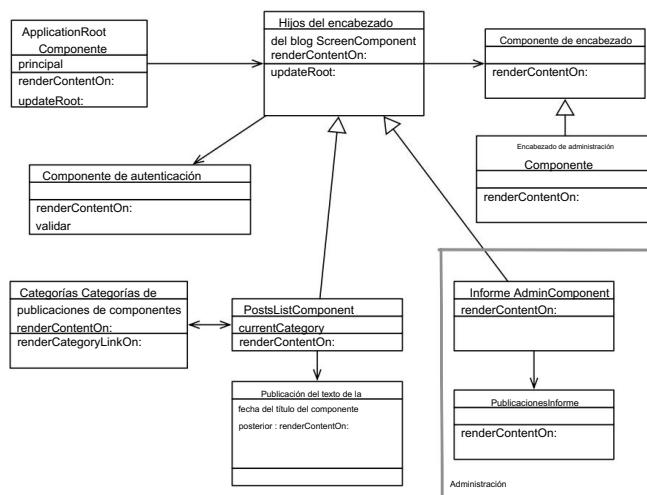


Figura 9-2 Componentes de administración.

9.2 Descripción de la publicación

Una descripción es un objeto que especifica información sobre los datos de nuestro modelo así como su tipo, si la información es obligatoria, si debe ordenarse y cuál es el valor por defecto.

9.2 Descripción de la publicación

Empecemos a describir la variable de cinco instancias de TBPost con Magritte.

Luego mostraremos cómo podemos generar un formulario para nosotros.

Definiremos los cinco métodos siguientes en el protocolo 'magritte-descriptions' de la clase TBPost. Tenga en cuenta que los nombres de los métodos no son importantes, pero seguimos una convención. Este es el pragma <magritteDescription> (anotación de método) que permite a Magritte identificar descripciones.

El título de la publicación es una cadena de caracteres que es obligatoria.

```
TBPost >> descriptionTitle
<magritteDescription>
^
MAStringDescription nuevo descriptor
de acceso: #título; ser requerido; tú
mismo
```

Una prueba posterior es una línea múltiple que es obligatoria.

```
TBPost >> descripciónTexto
<magritteDescripción>
^
MAMemoDescription nuevo
acceso: #texto; ser requerido; tú
mismo
```

La categoría se representa como una cadena y no es necesario proporcionarla. En tal caso, la publicación se ordenará en la categoría 'Sin clasificar'.

```
TBPost >> descripciónCategoría
<magritteDescripción>
^
MAStringDescription nuevo descriptor
de acceso: #categoría; tú mismo
```

El tiempo de creación de la publicación es importante ya que se usa para ordenar las publicaciones. Entonces se requiere.

```
TBPost >> descripciónFecha
<magritteDescripción>
^
MADateDescription nuevo acceso:
#fecha; ser requerido; tú mismo
```

La variable de instancia visible debe ser un valor booleano y es obligatoria.

```
[TBPost >> descripciónVisible
  <magritteDescripción>
    ^ MABooleanDescription nuevo
      acceso: #visible; ser requerido; tú
      mismo
```

Podríamos enriquecer las descripciones para que no sea posible publicar un post con fecha anterior al día actual. Podríamos cambiar la descripción de una categoría para asegurarnos de que una categoría sea parte de una lista predefinida de categorías.

No lo hacemos para mantenerlo en el punto principal.

9.3 Creación automática de componentes

Una vez que se describe una publicación, podemos generar un componente Seaside enviando un mensaje `asComponent` a una instancia de publicación.

```
[ aTBPost como componente
```

Veamos cómo podemos usar esto a continuación.

9.4 Elaboración de un informe de publicación

Desarrollaremos un nuevo componente que será utilizado por el componente TBAd minComponent. El componente TBPostReport es un informe que contendrá todas las publicaciones. Como veremos a continuación, el componente Seaside del informe se generará automáticamente desde Magritte. Podríamos haber desarrollado solo un componente, pero preferimos distinguirlo del componente de administración para la evolución futura.

El componente PostsReport

La lista de publicaciones se muestra mediante un informe generado dinámicamente por Magritte. Usaremos Magritte para implementar los diferentes comportamientos de la actividad de administración (lista de publicaciones, creación de publicaciones, edición, eliminación de una publicación).

El componente TBPostsReport es una subclase de TBSMagritteReport que gestiona informes con Bootstrap.

```
[ Subclase TBSMagritteReport: #TBPostsReport nombre de
  variable de instancia: "
    Nombres de variables de clase: "
    paquete: 'Componentes TinyBlog'
```

Agregamos un método de creación que toma un blog como argumento.

9.5 Integración de AdminComponent con PostsReport

```

Clase TBPostsReport >> de: aBlog |
  todosBlogs | allBlogs := aBlog
  allBlogPosts.
  ^
    autofilas: allBlogs descripción: allBlogs anyOne
    magritteDescription

```

9.5 Integración de AdminComponent con PostsReport

Ahora revisemos nuestro TBAdminComponent para mostrar este informe.

Agregamos un informe de variable de instancia y sus accesores en la clase Componente TBAdmin.

```

Subclase TBScreenComponent: #TBAdminComponent
  nombres de variables de instancia: 'informe'
  nombres de variables de clase: ''
  paquete: 'Componentes TinyBlog'

TBAdminComponent >> informe
  ^
    informe

TBAdminComponent >> informe: un informe
  ^
    informe := un informe

```

Dado que el informe es un componente hijo del componente de administración, no debemos olvidarnos de redefinir el método hijos. Tenga en cuenta que la colección contiene los subcomponentes definidos en la superclase (componente de encabezado) y los de la clase actual (componente de informe).

```

TBAdminComponent >> niños super
  ^
    niños copyWith: autoinforme

```

En el método de inicialización instanciamos un informe dándole una instancia de blog.

```

TBAdminComponent >> inicializar
  super inicializar.
  autoinforme:
    (TBPostsReport de: autoblog)

```

Modifiquemos la representación de la parte de administración para mostrar el informe.

```

TBAdminComponent >> renderContentOn: html
  super renderContentOn: html.
  html tbsContainer:
    [
      título html: 'Administrador del blog'.
      html
        reglahorizontal.
        representación html:
        autoinforme]

```

Puede probar este cambio actualizando su navegador web.

Welcome in TinyBlog	13 August 2018	TinyBlog
Report Pharo Sprint	13 August 2018	Pharo
Brick on top of Bloc - Preview	13 August 2018	Pharo
The sad story of unclassified blog posts	13 August 2018	Unclassified
Working with Pharo on the Raspberry Pi	13 August 2018	Pharo

New Session Configure Halos Profile Memory XHTML 3/4 ms

Figura 9-3 Informe de Magritte con publicaciones.

9.6 Columnas de filtro

De forma predeterminada, un informe muestra los datos completos de cada publicación. Sin embargo, algunas columnas no son útiles. Deberíamos filtrar las columnas. Aquí sólo conservamos el título, la categoría y la fecha de publicación.

Agregamos un método de clase para la selección de columnas y modificamos el método de: para usar esto.

```
TBPostsReport class >> filteredDescriptionsFrom: aBlogPost "Filtrar solo
algunas descripciones para las columnas del informe".
```

```
^ aBlogPost magritteDescription select:
 [ :each | #(fecha de categoría de título) incluye: cada selector de acceso]
```

```
Clase TBPostsReport >> de: aBlog |
 todosBlogs | allBlogs := aBlog
 allBlogPosts. filas propias: descripción
 de todos los blogs: (descripciones autofiltradas
 de: todos los blogs cualquiera)
```

La figura 9-3 muestra la situación que debería obtener.

9.7 Mejoras en los informes

9.7 Mejoras en los informes

El informe anterior es bastante crudo. No hay título en las columnas y el orden de las columnas de visualización no es fijo. Esto puede cambiar de una instancia a otra. Para manejar esto, modificamos la descripción de cada variable de instancia.

Especificamos una prioridad y un título (etiqueta del mensaje:) de la siguiente manera:

```
TBPost >> descriptionTitle
  ^magritteDescription>
    MAStringDescription nueva etiqueta:
      'Título'; prioridad: 100; accesos:
        #título; ser requerido; tú mismo

TBPost >> descripciónTexto
  ^magritteDescripción>
    MAMemoDescription nueva
    etiqueta: 'Texto'; prioridad: 200;
    accesos: #texto; ser requerido; tú
    mismo

TBPost >> descripciónCategoría
  ^MagritteDescripción>
    MAStringDescription nueva etiqueta:
      'Categoría'; prioridad: 300; accesos:
        #categoría; tú mismo

TBPost >> descripciónFecha
  ^magritteDescripción>
    MADateDescripción nuevo
    etiqueta: 'Fecha';
    prioridad: 400;
    descriptor de acceso:
      #fecha; ser requerido; tú
    mismo

TBPost >> descripciónVisible
  ^magritteDescripción>
    MABooleanDescription nueva etiqueta:
      'Visible'; prioridad: 500; accesos:
        #visible; ser requerido; tú mismo
```

Debería obtener la situación tal como se representa en la Figura 9-4.

Interfaz web de administración y generación automática de formularios

Title	Category	Date
Welcome in TinyBlog	TinyBlog	13 August 2018
Report Pharo Sprint	Pharo	13 August 2018
Brick on top of Bloc - Preview	Pharo	13 August 2018
The sad story of unclassified blog posts	Unclassified	13 August 2018
Working with Pharo on the Raspberry Pi	Pharo	13 August 2018

Figura 9-4 Informe de administración.

9.8 Administración de correos

Ahora podemos implementar un CRUD (Crear Leer Actualizar Eliminar) que permite generar publicaciones. Para ello, añadiremos una nueva columna (instancia de MACCommand Column) al informe. Esta columna agrupará las diferentes operaciones usando el mensaje addCommandOn:. Este método permite definir un enlace que ejecutará un método del objeto actual. Damos acceso al blog para el que se crea el informe.

```
[ Subclase TBSMagritteReport: #TBPostsReport
  nombres de variables de instancia: 'blog'
  nombres de variables de clase: "
  paquete: 'Componentes TinyBlog'
```

```
[ TBSMagritteReport >> blog blog
```

```
[ TBSMagritteReport >> blog: aTBBlog
  blog := aTBBlog
```

El método from: agrega una nueva columna al informe. Agrupa las diferentes operaciones.

```
[ Clase TBPostsReport >> de: aBlog | reportar
  blogPublicaciones | BlogPublicaciones :=
  unBlog todas lasPublicacionesBlog.
  report := self rows: blogPosts description: (self
  filteredDescriptionsFrom: blogPosts anyOne). reportar blog:
  aBlog. informe addColumn: (MACCommandColumn nuevo
```

9.9 Adición posterior

```

    addCommandOn: selector de informe: #viewPost: texto: 'Ver'; tú mismo;
    addCommandOn: selector de informe: #editPost: texto: 'Editar'; tú mismo;
    addCommandOn: selector de informes: #deletePost: texto: 'Eliminar'; tú
    mismo). informe
  
```

^

Tendremos que definir los métodos vinculados a cada operación en el siguiente sección.

Además, este método es un poco extenso y no separa la definición del informe de la definición de la operación. Una posible solución es crear un método de instancia llamado `addCommands` y llamarlo explícitamente. Intenta hacerlo para practicar.

9.9 Adición posterior

Además, una publicación no está asociada con una publicación y la colocamos justo antes del informe principal. Dado que este comportamiento es entonces parte del puerto del componente `TBPostsRe`, debemos redefinir el método `renderContentOn:` del componente `TBPostsReport` para insertar un complemento de enlace.

```

TBPostsReport >> renderContentOn: html html
  tbsGlyphIcon iconPencil. ancla html

  devolución de llamada: [self
    addPost]; con: 'Aregar publicación'.
  súper renderContentOn: html
  
```

Inicie sesión en otro momento y debería obtener la situación tal como se representa en la Figura 9-5.

9.10 Implementación de la acción CRUD

Cada acción (Crear/Leer/Actualizar/Eliminar) debe invocar métodos de la instancia de `TBPostsReport`. Los implementamos ahora. Se construye un formulario personalizado en base a la operación solicitada (no es necesario tener un botón de guardar cuando el usuario solo está viendo una publicación).

9.11 Adición posterior

Comencemos con la adición de puestos. El siguiente método `renderAddPostForm:` ilustra el poder de Magritte para generar formularios:

Interfaz web de administración y generación automática de formularios

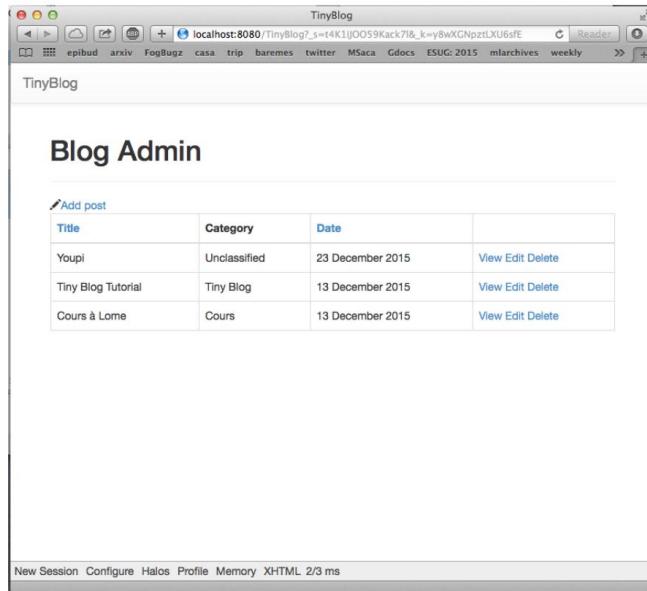


Figura 9-5 Publicar informe con enlaces.

```
[TBPostsReport >> renderAddPostForm: aPost
  ^ aPost asComponent addDecoration:
    (TBSMagritteFormDecoration botones: { #save -> 'Add post' . #cancel
-> 'Cancel'}); tú mismo
```

Aquí el mensaje `asComponent`, enviado al objeto de la clase `TBPost`, crea directamente un componente. Agregamos una decoración a este componente para administrar ok/cancelar.

El método `addPost` muestra el componente devuelto por el método `renderAddPostForm:` y cuando se crea una nueva publicación, se agrega al blog. El método `writeBlogPost:` guarda los cambios que el usuario pueda hacer.

```
[TBMensajesReporte >>
  addPost | publicar |
  post := self call: (self renderAddPostForm: TBPost new). publicar
  ifNotNil: [blog escribirBlogPost: publicar]
```

En este método vemos otro uso de la llamada de mensaje: dar el control a un componente. El enlace para agregar una publicación permite mostrar un formulario de creación que mejoraremos más adelante (consulte la Figura 9-6).

9.11 Adición posterior

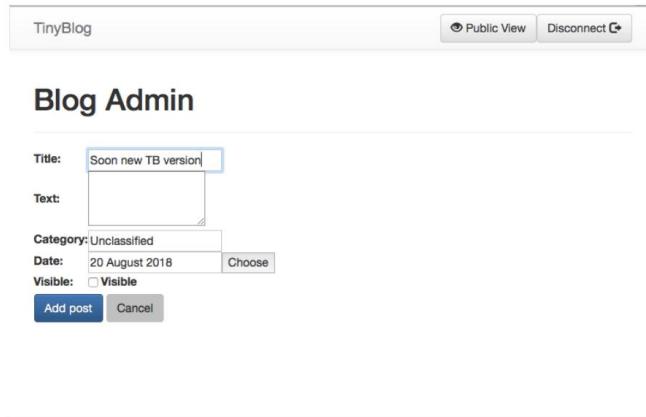


Figura 9-6 Representación básica de una publicación.

Visualización de publicaciones

Para mostrar una publicación en modo de solo lectura, definimos dos métodos similares al anterior. Tenga en cuenta que usamos `readonly: true` para indicar que el formulario no es editable.

```
TBPostsReport >> renderViewPostForm: aPost
  ^ aPost asComponent
    addDecoration: (botones TBSMagritteFormDecoration: { #cancelar
      -> 'Atrás' });
    solo
    lectura: verdadero;
    tú mismo
```

Mirar una publicación no requiere ninguna acción adicional además de renderizarla.

```
TBPostsReport >> viewPost: aPost
  autollamada: (self renderViewPostForm: aPost)
```

Edición posterior

Para editar una publicación, usamos el mismo enfoque.

```
TBPostsReport >> renderEditPostForm: aPost aPost
  ^ aPost asComponent addDecoration:
    (TBSMagritteFormDecoration botones: { #save
      -> 'Save post'. #cancel -> 'Cancel' });
    tú mismo
```

Ahora el método `editPost:` obtiene el valor de la llamada: `mensaje` y guarda los cambios realizados.

```
TBPostsReport >> editPost: aPost | publicar
| post := self call: (self renderEditPostForm:
aPost). publicar ifNotNil: [guardar blog]
```

Eliminando una publicación

Ahora debemos agregar el método removeBlogPost: a la clase TBBlog:

```
TBBlog >> removeBlogPost: aPost
eliminar publicaciones: una publicación si está ausente: [].
salvarse a sí mismo
```

Agreguemos una prueba unitaria:

```
TBBlogTest >> testRemoveBlogPost
autoafirmación: el tamaño del blog es igual
a: 1. blog removeBlogPost: blog allBlogPosts anyOne.
autoafirmación: el tamaño del blog es igual a: 0
```

Para evitar una operación no deseada, utilizamos un diálogo modal para que el usuario confirme la eliminación de la publicación. Una vez que se muestra la publicación, la lista de publicaciones administradas por TBPostsReport cambia y debe actualizarse.

```
TBPostsReport >> deletePost: aPost
(autoconfirmación: '¿Quieres eliminar esta publicación?') ifTrue:
[blog removeBlogPost: aPost]
```

9.12 Actualización de publicaciones

Los métodos addPost: y deletePost: funcionan bien, pero la pantalla no se actualiza. Necesitamos actualizar las listas de publicaciones usando la expresión auto actualización.

```
Informe de publicaciones de TB >> informe de actualización
filas propias: blog allBlogPosts.
actualización automática.
```

```
TBMensajesReporte >> addPost
| publicar | post := self call:
(self renderAddPostForm: TBPost new). publicar ifNotNil: [ blog writeBlogPost:
publicar.

informe de actualización automática]
```

```
TBPostsReport >> deletePost: aPost
(autoconfirmación: '¿Quieres eliminar esta publicación?')
ifTrue: [ blog removeBlogPost: aPost. informe de
actualización automática]
```

9.13 Mejor apariencia de forma

El informe no funciona e incluso administra restricciones de entrada: por ejemplo, los campos obligatorios deben completarse.

9.13 Mejor apariencia de forma

Para aprovechar Bootstrap, modificaremos las definiciones de Magritte. Primero especificamos que la representación del informe se basa en Bootstrap.

Un contenedor en Magritte es el elemento que contendrá a los demás componentes creados a partir de las descripciones.

```
TBPost >> descripciónContenedor
  <magritteContainer> super
    ^ descripciónContenedor componenteRenderer:
      TBSMagritteFormRenderer; tú mismo
```

Queremos ahora poder prestar atención a los diferentes campos de entrada y mejorar su apariencia.

```
TBPost >> descriptionTitle
  <magritteDescription>
    ^ MAMStringDescription nueva etiqueta:
      'Título'; prioridad: 100; accesor:
      #título; requiredErrorMessage:
      'Una entrada de blog debe tener
      un título.'; comentario: 'Por favor ingrese un título'; componenteClase:
      TBSMagritteTextInputComponent; ser requerido; tú mismo
```

```
TBPost >> descripciónTexto
  <magritteDescripción>
    ^ MAMMemoDescription nueva
      etiqueta: 'Texto'; prioridad: 200;
      accesor: #texto; ser requerido;
      requiredErrorMessage: 'Una
      entrada de blog debe contener
      un texto.'; comentario: 'Por favor ingrese un texto'; componenteClase:
      TBSMagritteTextAreaComponent; tú mismo
```

```
TBPost >> descripciónCategoría
  <magritteDescripción>
    ^ MAStringDescription nueva etiqueta:
      'Categoría'; prioridad: 300; accesor:
      #categoría; comentario: 'Sin
      clasificar si está vacío';
```

Interfaz web de administración y generación automática de formularios

TinyBlog

Blog Admin

Title

Please enter a title

Text

Please enter a text

Category

Unclassified if empty

Date

Visible

New Session Configure Halos Profile Memory XHTML 2/3 ms

Figura 9-7 Adición de formulario de publicación con Bootstrap.

componenteClase: TBSMagritteTextInputComponent; tú mismo

```
TBPost >> descripciónVisible
<magritteDescripción>
  MABooleanDescripción nuevo
    checkboxLabel: 'Visible'; prioridad: 500;
    accesor: #visible; componenteClase:
      TBSMagritteCheckboxComponent; ser
      requerido; tú mismo
```

Según las nuevas descripciones de Magritte, los formularios generados ahora usan Bootstrap. Por ejemplo, la edición del formulario de publicación no debería parecerse a la Figura 9-7.

9.14 Conclusión

En este capítulo, definimos la administración de TinyBlog en función del informe creado a partir de las publicaciones contenidas en el blog actual. Agregamos enlaces para administrar CRUD de edad para cada publicación. Lo que mostramos es que agregar descripciones en la publicación nos permite generar componentes de Seaside automáticamente.

CAPÍTULO 10

Cargando código de capítulo

Este capítulo contiene las expresiones para cargar el código descrito en cada uno de los capítulos. Dichas expresiones se pueden ejecutar en cualquier imagen de Pharo 8.0 (o superior). Sin embargo, usar la imagen Pharo MOOC (cf. Pharo Launcher) suele ser más rápido porque ya incluye varias bibliotecas como: Seaside, Voyage, ...

Cuando inicia por ejemplo el capítulo 4, puede cargar todo el código de los capítulos anteriores (1, 2 y 3) siguiendo el proceso descrito en la siguiente sección 'Capítulo 4'.

Obviamente, creemos que es mejor que uses tu propio código, pero tener nuestro código a mano puede ayudarte en caso de que te quedes atascado.

10.1 Capítulo 3: Ampliación y prueba del modelo

Puede cargar la corrección del capítulo anterior de la siguiente manera:

```
Metacello nuevo
línea de base: 'TinyBlog';
repositorio: 'github://LucFabresse/TinyBlog:chapter2/src'; onConflict: [ :ex
| ex useLoaded ]; carga
```

¡Haz las pruebas! Para hacerlo, puede usar TestRunner (menú Herramientas > Test Runner), buscar el paquete TinyBlog-Tests y hacer clic en "Ejecutar seleccionados". Todas las pruebas deben ser verdes.

10.2 Capítulo 4: Persistencia de datos usando Voyage y Mongo

Puede cargar la corrección del capítulo anterior de la siguiente manera:

```
[ Metacello nuevo
  línea de base: 'TinyBlog';
  repositorio: 'github://LucFabresse/TinyBlog:chapter3/src'; onConflict: [ :ex
  | ex useLoaded ]; carga
```

Una vez cargado ejecutar las pruebas.

10.3 Capítulo 5: Primeros pasos con Seaside

Puede cargar la corrección del capítulo anterior de la siguiente manera:

```
[ Metacello nuevo
  línea de base: 'TinyBlog';
  repositorio: 'github://LucFabresse/TinyBlog:chapter4/src'; onConflict: [ :ex
  | ex useLoaded ]; carga
```

Ejecutar las pruebas.

Para probar la aplicación, inicie el servidor HTTP:

```
[ ZincServerAdaptor startOn: 8080.
```

Abra su navegador web en <http://localhost:8080/TinyBlog>

Es posible que deba volver a crear algunas publicaciones de la siguiente manera:

```
[ Restablecimiento de TBBlog; crearDemoPublicaciones
```

10.4 Capítulo 6: Componentes web para TinyBlog

Puede cargar la corrección del capítulo anterior de la siguiente manera:

```
[ Metacello nueva
  línea de base: 'TinyBlog';
  repositorio: 'github://LucFabresse/TinyBlog:chapter5/src'; onConflict: [ :ex
  | ex useLoaded ]; carga
```

Mismo proceso que el anterior.

10.5 Capítulo 7: Gestión de categorías

Puede cargar la corrección del capítulo anterior de la siguiente manera:

10.6 Capítulo 8: Autenticación y Sesión

Metacelso nuevo

```
línea de base: 'TinyBlog';
repositorio: 'github://LucFabresse/TinyBlog:chapter6/src'; onConflict: [ :ex
| ex useLoaded ]; carga
```

Para probar la aplicación, inicie el servidor HTTP:

```
[ ZnZincServerAdaptor startOn: 8080.
```

Abra su navegador web en <http://localhost:8080/TinyBlog>

Es posible que deba volver a crear algunas publicaciones de la siguiente manera:

```
[ Restablecimiento de TBBlog; crearDemoPublicaciones
```

10.6 Capítulo 8: Autenticación y Sesión

Puede cargar la corrección del capítulo anterior de la siguiente manera:

Metacelso nuevo

```
línea de base: 'TinyBlog';
repositorio: 'github://LucFabresse/TinyBlog:chapter7/src'; onConflict: [ :ex
| ex useLoaded ]; carga
```

Para probar la aplicación, inicie el servidor HTTP:

```
[ ZnZincServerAdaptor startOn: 8080.
```

10.7 Capítulo 9: Interfaz web de administración y generación automática de formularios

Puede cargar la corrección del capítulo anterior de la siguiente manera:

Metacelso nuevo

```
línea de base: 'TinyBlog';
repositorio: 'github://LucFabresse/TinyBlog:chapter8/src'; onConflict: [ :ex
| ex useLoaded ]; carga
```

10.8 Última versión de TinyBlog

La versión más actualizada de TinyBlog se puede cargar de la siguiente manera:

Metacelso nuevo

```
línea de base: 'TinyBlog';
repositorio: 'github://LucFabresse/TinyBlog/src'; onConflict:
[ :ex | ex useLoaded ],
```

Cargando código de capítulo

|
└ carga.

CAPÍTULO 11

Guarda tu código

Cuando guarda la imagen de Pharo (Menú de Pharo > Guardar entrada de menú), contiene todos los objetos del sistema, así como todas las clases. Esta solución es útil pero frágil. Le mostraremos cómo los Pharoers guardan su código directamente usando Iceberg.

Iceberg es la herramienta de control de versiones de código de Pharo (introducida en Pharo 7.0) que envía código directamente a sitios web conocidos como: github, bitbucket o git lab.

Le sugerimos que lea el capítulo del libro "Managing Your Code with Iceberg" (disponible en <http://books.pharo.org>).

Aquí enumeramos los puntos clave:

- Crear una cuenta en <http://www.github.com> o similar.
 - Crear un proyecto en <http://www.github.com> o similar.
 - Agregue un nuevo proyecto a Iceberg eligiendo la opción: clonar de github.
 - Cree una carpeta 'src' con la lista de archivos o usando la línea de comando en el carpeta que acaba de clonar.
 - Abra su proyecto y agregue sus paquetes (defina una línea de base para poder recargar su código; consulte <https://github.com/pharo-open-documentation/pharo-wiki/blob/master/General/Baselines.md>)
 - Confirme su código. •
- Empuje su código en github.

