CHAPTER 1

Building A Simple Contact Book Application

In this tutorial, you will develop a simple contact book application with a web interface. It consists of a couple of classes for the model and the UI of the application. Figure 1.1 shows the resulting application. As you see it is simple but covers many aspects of defining and deploying a web applications.

You will develop the web interface using Seaside (http://ww.seaside.st[1]). Seaside is a powerful web framework for developing highly dynamic and complex web application. For more information you can read the book: 'Dynamic Web Development with Seaside' which is freely available at http://book.seaside.st. In this little tutorial we will only use some simple server side Seaside behavior. We will use the Twitter Bootstrap library that is fully integrated to Seaside.

Note that the presented solution is often simple and we will list ideas of further improvements that you could add. In addition, it is worth to know that Pharo developers often use object descriptions (as done via the Magritte framework) to generate new web components instead of manually developing them.

About development style

While we love coding test first and coding in the debugger (since it let us go much faster), in this tutorial we will not follow such style because it requires a lot of text to mention what to do with the user interface of the Pharo tools. We decided to take a neutral stand point and to let you decide if you code

[1] http://www.seaside.st

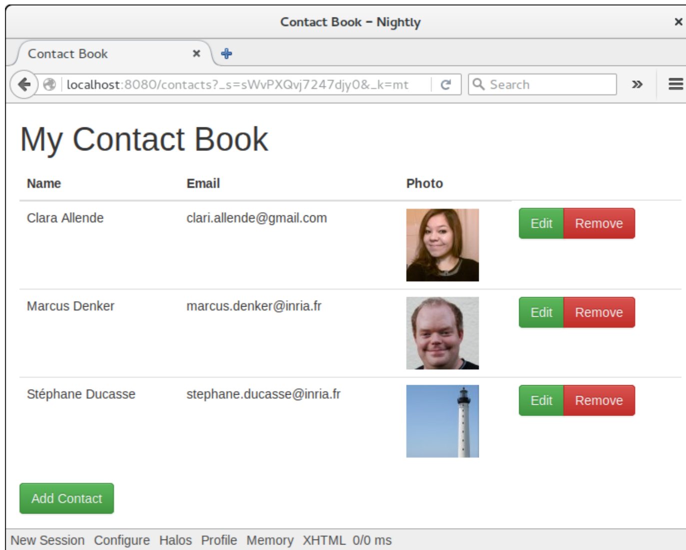

Figure 1.1 Screenshot of the finished contact book application.

first the tests, if you prefer to go slowly or not. This tutorial presents the mandatory information so that you can get it done. Program it the way you want.

Getting Seaside

In this tutorial, we use the Seaside web framework to define the views of our application. Seaside must be loaded in the image to start using it. You have several ways to obtain Seaside

- either you start coding in the default pharo image and when needed you open the *Catalog browser* tool and search for Seaside. Install the stable version. Since Seaside is large it can take a moment.
- You can also download a pre-installed version of Seaside available at https://ci.inria.fr/pharo-contribution/job/Seaside/ but you will need to migrate your code to this new image. This is easy with the code versioning control system: save from the current image to a code repository and load from the Seaside ready image from this repository. We prefer the second version.

1.1 The Model

The contact book is composed of two model classes: `Contact` and `ContactBook`. Let us start with the `Contact` class.

The Contact Class

A contact has two instance variables `fullname` and `email`, both strings. Create the `Contact` class with its instance variables:

```
Object subclass: #Contact
   instanceVariableNames: 'fullname email'
   classVariableNames: ''
   package: 'ContactBook'
```

It should be possible to create a new contact by executing the following code:

```
contact := Contact newNamed: 'Marcus Denker' email:
    'marcus.denker@inria.fr'.
```

It means that we send a message (`newNamed:email:`) to the class `Contact` itself and that it should return a `Contact` instance.

Add the `newNamed:email:` class method in the `instance creation` protocol of the `Contact` class. To define a class method, pay attention that you should browse the *class side* of the `Contact` class.

```
Contact class >> newNamed: aNameString email: anEmailString
   ^ self new
        fullname: aNameString;
        email: anEmailString;
        yourself
```

Adding a Test

You can now define a test (in the `tests` protocol) to make sure that you can create an instance and that it contains the right information.

```
TestCase subclass: #ContactTest
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'ContactBook'
```

```
ContactTest >> testCreation
   | contact |
   contact := Contact newNamed: 'Marcus Denker' email:
    'marcus.denker@inria.fr'.
   self assert: contact fullname = 'Marcus Denker'.
   self assert: contact email = 'marcus.denker@inria.fr'.
```

For this test to work you need to define the following methods (in the `accessing` protocol).

```
Contact >> fullname
   ^ fullname

Contact >> fullname: aString
   fullname := aString
```

```
Contact >> email
   ^ email

Contact >> email: aString
   email := aString
```

Run the test to make sure it passes. It is a good idea to version your code and save the image.

Further extensions. You can add some extra logic to make sure that the name is not surrounded by space. In addition handling whether you take care of note about lowercase and uppercase should be addressed. You can use messages such as `asLowercase` to, for example, store only lowercased strings.

Enhancing Object Textual Interface

If you inspect or print the `contact` variable created above, you will see that the string representation of this `Contact` instance is `"a Contact"` which says nothing about the contact itself. This is problematic when debugging and developers typically appreciate nicer string representations. You can change that by overriding the method `Object>>printOn:` (in the `printing` protocol) in the `Contact` class and sending several `nextPutAll:` messages to the stream argument as follows:

```
Contact >> printOn: aStream
   aStream
      nextPutAll: self fullname;
      nextPutAll: ' <';
      nextPutAll: self email;
      nextPutAll: '>'
```

The string representation of the above `contact` variable will then be:

```
'Marcus Denker <marcus.denker@inria.fr>'
```

Class Comment and Saving

We hope you didn't forget to add a comment to the `Contact` class. If you did, here is a possible one:

```
I represent a person with a name and an email address. I'm usually
    part of a contact book.
```

Note that in a real application, it might be better to use an `Email` class to represent a contact's email address.

Save your image and save the code using the version control browser (Monticello Browser).

The Contact Book Class

A contact book contains a collection of `Contact` instances:

```
Object subclass: #ContactBook
   instanceVariableNames: 'contacts'
   classVariableNames: ''
   package: 'ContactBook'
```

Initializing Contact Books

To initialize the `contacts` variable to an empty collection, you can either define an `initialize` method (that is automatically invoked at instance-creation time) or use lazy initialization. The following code uses lazy initialization. Add this method in the `accessing` protocol. Pay attention that with lazy initialization, you should systematically use accessors else you could access to a variable not well-initialized.

```
ContactBook >> contacts
   ^ contacts ifNil: [ contacts := OrderedCollection new ]
```

It should be possible to add and remove contacts from the collection. Add the necessary methods in the `action` protocol:

```
ContactBook >> addContact: aContact
   self contacts add: aContact

ContactBook >> removeContact: aContact
    self contacts remove: aContact
```

Add tests. Define some tests to cover the addition and removal of contacts. You will see in particular that the definition for `removeContact:` is not that robust when we want to remove a contact that is not in the collection: Removing an unexisting contact raises an error. Change the definition of the `removeContact:` method (check other remove: methods in collection) and define a test that covers this particular aspect.

Providing a Default Contact Book

To simplify further development, we define a default contact book with predefined contacts inside. We do this by adding the method `createDefault` as a class method to the 'default instance' protocol of `ContactBook class`:

```
ContactBook class >> createDefault
   ^ self new
      addContact: (Contact
                  newNamed: 'Damien Cassou'
                email: 'damien@cassou.me');
        addContact: (Contact
                  newNamed: 'Marcus Denker'
```

```
                 email: 'marcus.denker@inria.fr');
           addContact: (Contact
                   newNamed: 'Tudor Girba'
                 email: 'tudor@tudorgirba.com');
       addContact: (Contact
                   newNamed: 'Clara Allende'
                 email: 'clari.allende@gmail.com');
       yourself
```

Don't forget to comment the class and to save your image and version your code.

1.2 A First Web View

Now that we have the model, we need a web view. We will use Seaside for that. Seaside is a component framework: Seaside web applications are built by aggregating components. Typically, an application consists of a top-level component delegating parts of its rendering to sub-components. Let us define first a simple component.

Defining the WAContactBook Component

Our simple application consists of a top-level component represented by the `WAContactBook` class, subclass of `WAComponent`. Create the `WAContactBook` class:

```
WAComponent subclass: #WAContactBook
   instanceVariableNames: ''
   classVariableNames: ''
   package: 'ContactBook'
```

Rendering a Title

Every Seaside component class must override the `renderContentOn:` method to specify how a component is rendered. Define this method in the `rendering` protocol:

```
WAContactBook>>renderContentOn: html
   "Main entry point of the view. Render a title."

   html heading
      level: 1;
      with: 'My Contact Book'.
```

The method argument `html` acts as a canvas that can emit adequate HTML code. Above code asks the heading brush to the canvas and uses it to emit this HTML code: `<h1>My Contact Book</h1>`. Seaside abstracts you from the details of the syntax of the HTML expressions.

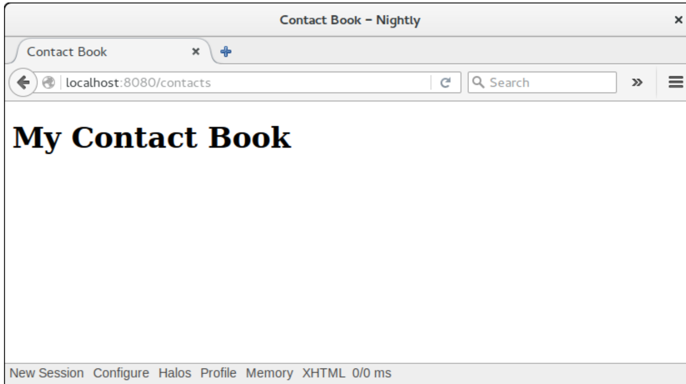

Figure 1.2 Screenshot of the contact book application title

Remark. It is not a good style to generate component UI using directly HTML commands representing styles (table, section...). Usually developers emit tags that map with css class tags provided by an application designer. We will show you that later.

Registering our 'App'

The next step is to register the `WAContactBook` class to the `/contacts` URL path. This way you will be able to reach the application at http://localhost:8080/contacts. Do this by implementing an `initialize` class method in `WAContactBook class` (`class initialization` protocol):

```
WAContactBook class >> initialize
    WAAdmin register: self asApplicationAt: 'contacts'.
```

An `initialize` class method is executed when the class is loaded in memory. Because our class is already loaded, we should execute it once manually. Write and execute this code, as method comment or in the Playground for example:

```
WAContactBook initialize
```

Starting the Server

The last step before getting something on the web browser is to start the web server. Open the `Seaside control panel` tool (open the Pharo menu and go to the Tools sub-menu). Now, add a `ZnZincServerAdaptor` by right-clicking on the top pane and choosing `Add adaptor...`. When prompted, choose a port, for example 8080 and press `Start`.

Open your favorite web browser on http://localhost:8080/contacts and you should see something similar to Figure 1.2. Currently, the title of the web

page is ”Seaside”, as we can see it in the web browser window title and the tab title. This can be changed by overriding the `updateRoot:` method (in the `updating` protocol):

```
WAContactBook >> updateRoot: anHtmlRoot
    super updateRoot: anHtmlRoot.
    anHtmlRoot title: 'Contact Book'
```

You can refresh the page in the web browser to see the result.

Accessing the Model

Now we define some methods to access to the model from the view. We define them in the `accessing` protocol:

```
WAContactBook >> contacts
   ^ self contactBook contacts
```

```
WAContactBook >> contactBook
   ^ contactBook ifNil: [ contactBook := ContactBook createDefault ]
```

The variable `contactBook` is an instance variable of the `WAContactBook` class. In the ’iterating’ protocol add the method `contactsDo:`.

```
WAContactBook >> contactsDo: aBlock
    self contacts do: aBlock
```

This `contactsDo:` method is not as useless as it might seem. This method hides the existence of a `contactBook` collection and could be useful later to replace the collection by a database.

Rendering a Table of Contacts

Below the title, we want a table containing the contacts of the contact book. For this, we need to change the `renderContentOn:` method and add a few new messages. We will decompose the behavior in several methods to facilitate understanding.

```
WAContactBook >> renderContentOn: html
   "Main entry point of the view. Render both a title and the list
    of contacts."

   html heading
      level: 1;
      with: 'My Contact Book'.
   self renderContactsOn: html
```

We just used a new method named `renderContactsOn:` (all rendering methods should be put in the `rendering` protocol). The method `renderContactsOn:` defines a table with a header and delegates the rest of the rendering to the method `renderContact:on:`.

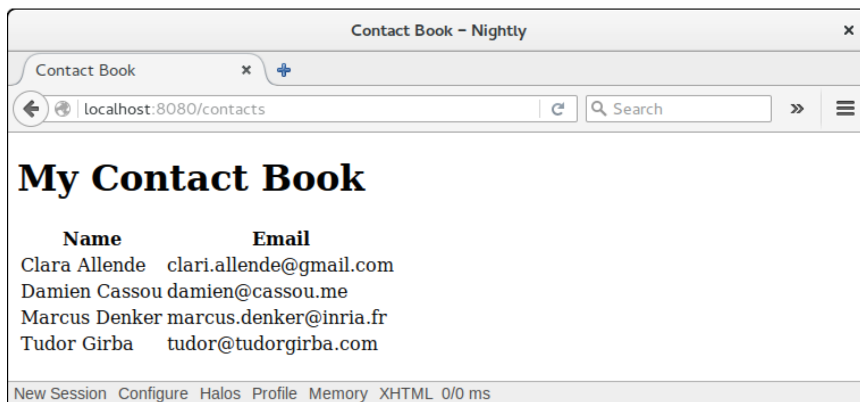

Figure 1.3 Screenshot of a contact book contacts.

```
WAContactBook >> renderContactsOn: html
   html table: [
      html tableHead: [
         html
            tableHeading: 'Name';
            tableHeading: 'Email' ].
      self contactsDo: [ :contact | self renderContact: contact on:
    html ] ]
```

The method `renderContact:on:` defines the rendering of a single contact in a table row.

```
WAContactBook >> renderContact: aContact on: html
   html tableRow: [
      html
        tableData: aContact fullname;
        tableData: aContact email ]
```

As we saw, the `renderContentOn:` method delegates the table rendering to the `renderContactsOn:` method. The latter creates a table with a heading row and delegates the contact rendering to the `renderContact:on:` method. This method renders a table row with the contact's details.

Refreshing the web browser should now show a list of contacts as can be seen in Figure 1.3.

1.3 Improving the View with Twitter Bootstrap

The rendering can be visually improved by adding some Cascading Style Sheets (CSS). In the following, we use the Twitter Bootstrap framework[2] that

[2] http://getbootstrap.com/

must be loaded in the image. It is loaded by default in the Seaside distribution. If not, open the *Catalog browser* tool and search for Bootstrap. Install the stable version.

Declaring Twitter Bootstrap Use

The contact book application must declare its dependency on Bootstrap. This is done by modifying the `initialize` class method:

```
WAContactBook class >> initialize
    (WAAdmin register: self asApplicationAt: 'contacts')
        addLibrary: JQDeploymentLibrary;
        addLibrary: TBSDeploymentLibrary
```

Execute this method manually again:

```
WAContactBook initialize
```

Using Twitter Brushes

The Seaside version of Bootstrap defines some special brushes as messages (such as `tbsContainer:` and `tbsTable`) to improve the application rendering. We now adapt our existing code to use these methods:

```
WAContactBook >> renderContentOn: html
   "Main entry point of the view. Render both a title and the list
    of contacts."

   html
      tbsContainer: [
         html heading
            level: 1;
            with: 'My Contact Book'.
         self renderContactsOn: html ]
```

```
WAContactBook >> renderContactsOn: html
   html tbsTable: [
      html tableHead: [
         html
            tableHeading: 'Name';
            tableHeading: 'Email' ].
      self contactsDo: [ :contact | self renderContact: contact on:
    html ] ]
```

As you can see, the adaptation consisted in adding a container with `tbsContainer:` and replacing a `table:` by a `tbsTable:` message.

The result in Figure 1.4 already looks much nicer. However, in a real application, it is recommended to avoid Bootstrap specific methods such as `tbsContainer:` and `tbsTable:` but to use Bootstrap mixins instead. We explained the idea in Section 1.6.

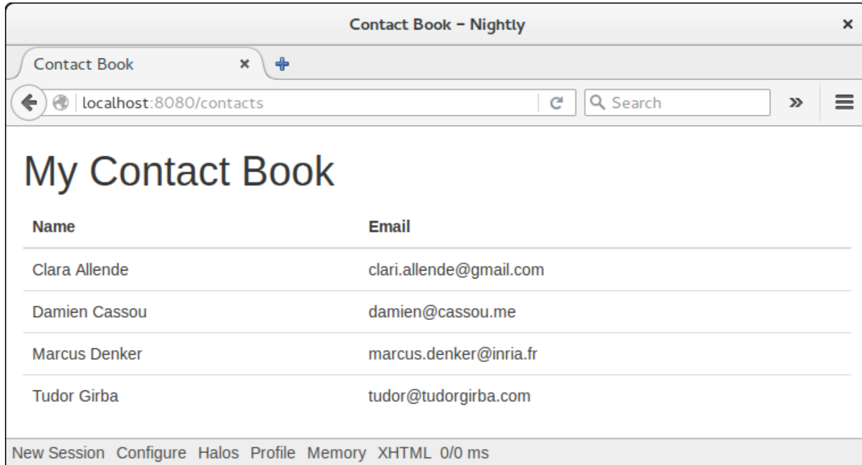

Figure 1.4 Screenshot of the contact book application with bootstrap.

1.4 Adding Photos

We now improve the contact book application by displaying photos next to each contact. We fetch these photos automatically from the web using Gravatar[3]. Gravatar provides a web API[4] to retrieve a photo from an email address:

```
Contact >> gravatarUrl
   ^ 'http://www.gravatar.com/avatar/', (MD5 hashMessage: email
    asString trimBoth asLowercase) hex, '.jpg'
```

For example, for marcus.denker@inria.fr, the Gravatar URL is:

```
'http://www.gravatar.com/avatar/c147c32f94baa71afa9d7be0a289766d.jpg'
```

The web application must be adapted with a new column for the photos:

```
WAContactBook >> renderContactsOn: html
   html tbsTable: [
      html tableHead: [
         html
            tableHeading: 'Name';
            tableHeading: 'Email';
            tableHeading: 'Photo' ].
      self contactsDo: [ :contact | self renderContact: contact on:
    html ] ]
```

```
WAContactBook >> renderContact: aContact on: html
   html tableRow: [
```

[3] http://gravatar.com/
[4] http://en.gravatar.com/site/implement/

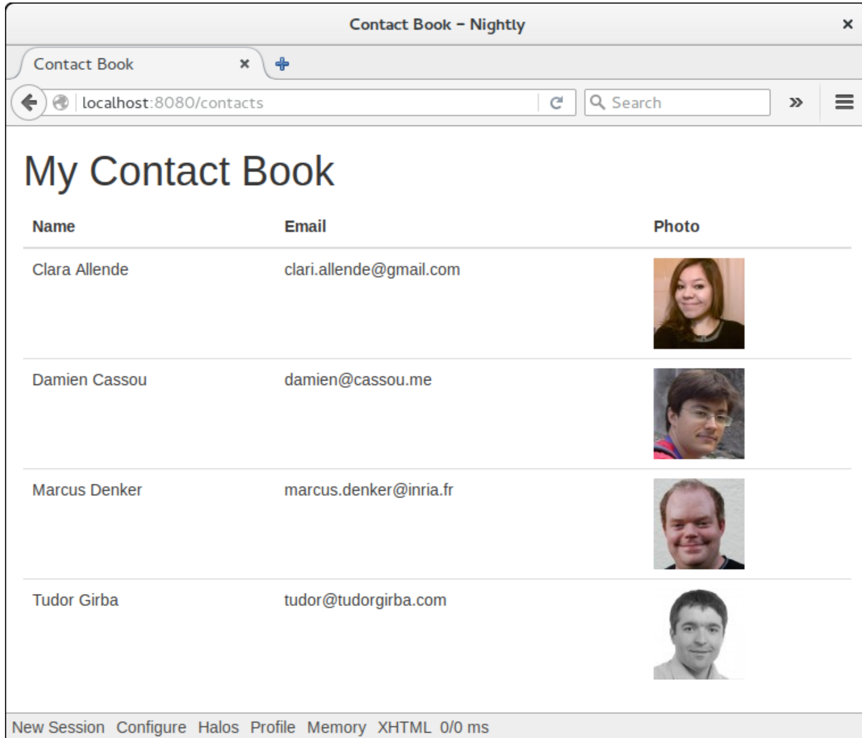

Figure 1.5 Screenshot of the contact book application with photos

```
      html
         tableData: aContact name;
         tableData: aContact email;
   tableData: [ self renderPhotoOf: aContact on: html ] ]
```

```
WAContactBook >> renderPhotoOf: aContact on: html
   html image url: aContact gravatarUrl
```

The result in Figure 1.5 contains a new column for the contact photos, automatically fetched from a web service.

1.5 Adding Actions

We now add buttons to add a new contact and to remove and edit an existing contact.

Adding a Remove Button

We first add a remove button on each contact line in the table.

```
WAContactBook >> renderContact: aContact on: html
   html tableRow: [
      html
         tableData: aContact name;
         tableData: aContact email;
         tableData: [ self renderPhotoOf: aContact on: html ];
   tableData: [ self renderRemoveButtonForContact: aContact on: html
     ]]
```

```
WAContactBook >> renderRemoveButtonForContact: aContact on: html
   html tbsButton
      beDanger;
      callback: [ self contactBook removeContact: aContact ];
      with: 'Remove'
```

You can refresh the page in the web browser and you will see the remove buttons. However, none of them will work because an HTML form must wrap the buttons. This can be done by modifying the `renderContentOn:` method again and add `tbsForm:`:

```
WAContactBook >> renderContentOn: html
  "Main entry point of the view. Render both a title and the list of
    contacts."

  html
    tbsContainer: [
      html heading
        level: 1;
        with: 'My Contact Book'.
      html tbsForm: [ self renderContactsOn: html ] ]
```

The remove buttons should now work fine.

Add/Edit new Contact

Implementing buttons to add a new contact or edit an existing one is a bit more involving because it requires creating a new component to edit the contact fields as in Figure 1.6.

Creating a new Component

We know create a component to be able to edit a contact. This is typically such task that the use of a system like Magritte can avoid. Create a new subclass of `WAComponent` which contains a `contact` instance variable.

```
WAComponent subclass: #WAContact
   instanceVariableNames: 'contact'
   classVariableNames: ''
   package: 'ContactBook'
```

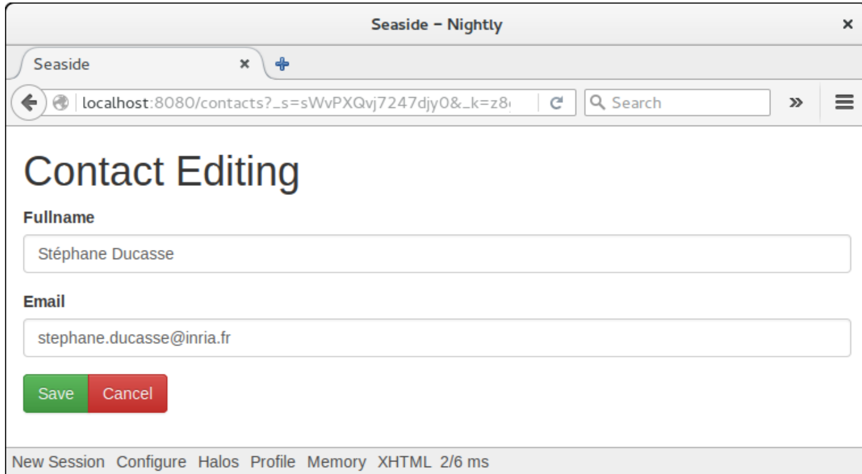

Figure 1.6 Screenshot of the contact editor.

We define a class method `editContact:` to set the corresponding contact.

```
WAContact class >> editContact: aContact
   ^ self new
        setContact: aContact;
        yourself
```

This time we do not use lazy initialization but we initialize the object by specialising the method `initialize`.

```
WAContact >> initialize
   super initialize.
   contact := Contact new.
```

```
WAContact >> setContact: aContact
    contact := aContact

WAContact >> contact
    ^ contact
```

Rendering a Contact

We define a new method `renderContentOn:` for the class `WAContact`

```
WAContact >> renderContentOn: html
   html tbsContainer: [
      html heading with: 'Contact Editing'.
      html tbsForm with: [
         self renderFieldsOn: html ] ]
```

```
WAContact >> renderFieldsOn: html
    self renderFullnameFieldOn: html.
    self renderEmailFieldOn: html
```

The method `renderFullnameFieldOn:` and `renderEmailFieldOn:` both render a label and an input field.

Fields

Here is the `renderFullnameFieldOn:` method:

```
WAContact >> renderFullnameFieldOn: html
   html tbsFormGroup: [
      html label: 'Fullname'.
      html textInput
         tbsFormControl;
         placeholder: 'fullname';
         callback: [ :value | self contact fullname: value ];
         value: (self contact fullname ifNil: '') ]
```

The `tbsFormGroup:` method is a Bootstrap method to visually group a label and an input field together. Sending the `textInput` message to `html` creates a new text input. The other messages configure it:

- `tbsFormControl` adds Bootstrap-specific HTML markup;
- `placeholder:` adds a ghost text indicating the purpose of the field and expected value that the user is expected to enter;
- `callback:` attaches some code to be executed when the form is validated: the first parameter of the block is the value typed in the input field;
- `value:` writes a default value in the field.

The `renderEmailFieldOn:` is very similar:

```
WAContact >> renderEmailFieldOn: html
   html tbsFormGroup: [
      html label: 'Email'.
      html emailInput
         tbsFormControl;
         placeholder: 'your@email.eu';
         callback: [ :value | self contact email: value ];
         value: (self contact email ifNil: '') ]
```

The only difference with `renderFullnameFieldOn:` lies in the fact that the input field is dedicated to entering email addresses (the `textInput` message has been replaced by the `emailInput` message). In the `callback:` block, the parameter `email` is an instance of `WAEmailAddress`: it is necessary to send the message `address` to this object to get a string.

Save/Cancel Buttons

We now need to add 2 buttons: one to save the changes and one to cancel the changes. We introduce the message `renderButtonsOn:` in the `WAContact` as follows:

```
WAContact >> renderContentOn: html
   html tbsContainer: [
      html heading with: 'Contact Editing'.
      html tbsForm with: [
         self renderFieldsOn: html.
         self renderButtonsOn: html ] ].
```

Then we define the corresponding methods as follows:

```
WAContact >> renderButtonsOn: html
   html tbsFormGroup: [
      html tbsButtonGroup: [
         self
            renderSubmitButtonOn: html;
            renderCancelButtonOn: html ] ]
```

```
WAContact >> renderSubmitButtonOn: html
   html tbsSubmitButton
      beSuccess;
      bePrimary;
      callback: [ self answer: self contact ];
      with: 'Save'
```

It is important to see that the callback of the submit button is using the message `answer:`. This message is part of the `call:` and `answer:` protocol of Seaside. `call:` schedules a component and `answer:` unschedule it and return a value to the caller.

```
WAContact >> renderCancelButtonOn: html
   html tbsButton
      beDanger;
      cancelCallback: [ self answer: nil ];
      with: 'Cancel'
```

In the cancel button, we use `answer: nil`, since there is no value returned. Still the component (here the editor) should be closed.

Adding the Add/Edit Buttons to the ContactBook

The last piece of the puzzle is the addition of the add and edit buttons on the contact book component.

Edit Button

We start with the edit button, at the end of each contact row in the table:

```
WAContactBook >> renderContact: aContact on: html
   html tableRow: [
      html
         tableData: aContact fullname;
         tableData: aContact email;
         tableData: [ self renderPhotoOf: aContact on: html ];
         tableData: [ self renderButtonsForContact: aContact on:
    html ] ]
```

```
WAContactBook >> renderButtonsForContact: aContact on: html
   html tbsButtonGroup: [
         self
            renderEditButtonForContact: aContact on: html;
            renderRemoveButtonForContact: aContact on: html ]
```

```
WAContactBook >> renderEditButtonForContact: aContact on: html
   html tbsButton
      beSuccess;
      callback: [ self call: (WAContact editContact: aContact) ];
      with: 'Edit'
```

In the edit button callback, the message `call:` is sent to self to temporarily replace the contact book component by the contact editor. When the contact editor sends `answer:` to itself (in `WAContact>>renderSubmitButtonOn:` and `WAContact>>renderCancelButtonOn:`), the control flow comes back to this button's callback.

Adding the Add new Button

The add new contact button is as simple: We add a message `renderGlobalButtonsOn:`.

```
WAContactBook >> renderContentOn: html
   "Main entry point of the view. Render both a title and the list
    of contacts."

   html tbsContainer: [
      html heading
         level: 1;
         with: 'My Contact Book'.
      html tbsForm: [
         self renderContactsOn: html.
   self renderGlobalButtonsOn: html ] ]
```

The method `renderGlobalButtonsOn:` defines a simple button.

```
WAContactBook >> renderGlobalButtonsOn: html
   html tbsButtonGroup: [
      html tbsButton
         beSuccess;
         callback: [ self addContact ];
         with: 'New contact' ]
```

```
WAContactBook >> addContact
   (self call: WAContact new)
      ifNotNil: [ :contact | contactBook addContact: contact ]
```

The message `call:` returns the same object that was returned by the corresponding message `answer:` (in `WAContact>>renderSubmitButtonOn:` and `WAContact>>renderCancelButtonOn:`). Because pressing the cancel button in the contact editor passes `nil` to `answer:`, the message `call:` may return `nil`. If `call:` returns something different, it will be a new contact which should be added to the contact book.

You should now get the same result as in Figure 1.1.

1.6 About using Bootstrap tags

Now as we mentioned before when using the Seaside tags `tbs` we are promoting a bad practice. We did so because we did not want to spend time explain CSS and HTML. But you should really change your code.

We are adding styling to the places where it should not be and we are breaking the idea that the code and its display should be separated. You can read the following blog to get a deeper view on it: http://ruby.bvision.com/blog/please-stop-embedding-bootstrap-classes-in-your-html

Here is a summary

The argument is that having html code like this:

```
    <div class="row">
      <div class="span6">...</div>
      <div class="span6">...</div>
    </div>
```

goes against the separation of rendering and content that CSS is supposed to bring. Adding all those bootstrap classes everywhere in the HTML is not better than adding `<table>` tags to layout web pages.

The solution is to use SCSS/SASS/Less mixins to add sanity to your html. Something like:

```
    <div class="book-previews">
      <div class="book-preview">...</div>
      <div class="book-preview">...</div>
    </div>
```

and the SCSS/SASS/Less stylesheet:

```
    .book-previews {
      .makeRow();         // Mixin provided by Bootstrap
      ...
    }
```

1.7 Summary

During this tutorial we defined a simple model and two simple web views. We could have written tests and implement methods in the debugger while the application is running. Pharo developers often prefer to write code in the debugger because they go faster and the execution provides objects to play with.

Further Development

As you see the current functionality is rather limited, here is a list of possible extensions.

- Add more information to describe a contact. For this we suggest that you use Magritte (Magritte is a framework to describe data and its Seaside extension defines automatically Seaside component). You can read the Magritte chapter in the book as well as the Magritte tutorial available at https://github.com/SquareBracketAssociates/PharoInProgress. With Magritte, the components such as our `WAContact` are automatically generated.
- Use Twitter mixin class tags instead of hardcoded brushes.
- Save and load contacts in an external format such as JSON or STON (STON is a Pharo object notation format. It is by default since Pharo 50).
- Save the contacts in a MongoDB using the Voyage framework (Check the chapters available on http://books.pharo.org)