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. You will develop the web interface using Seaside (http://ww.seaside.st). 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. Now in this little tutorial we will only use some simple server side Seaside behavior.

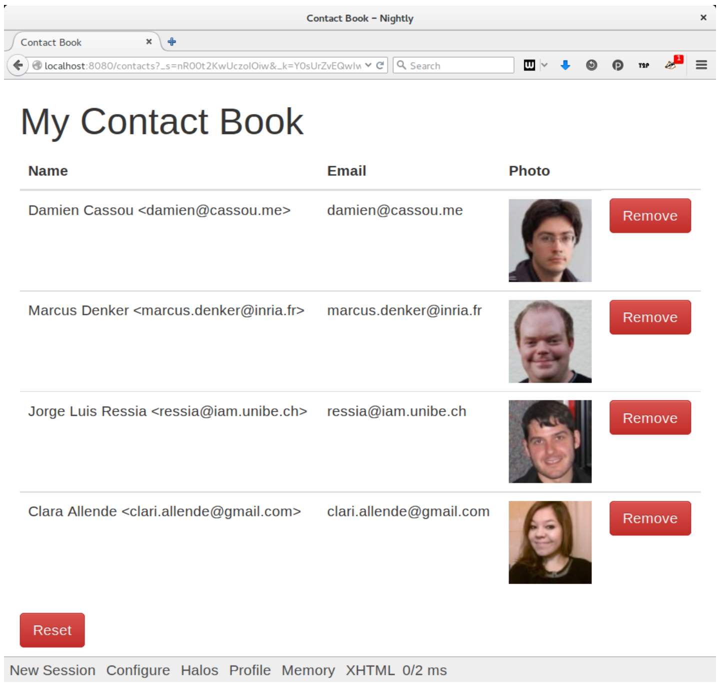
1.1 The Model

The contact book is composed of 2 model classes: `Contact` and `ContactBook`. Let us start with the `Contact` class.

The Contact

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: ''
   category: 'ContactBook'
```



Contact Book – Nightly

Contact Book

localhost:8080/contacts?_s=nR00t2KwUczolOiw&_k=YOsUrZvEQwlw

My Contact Book

Name	Email	Photo	
Damien Cassou <damien@cassou.me>	damien@cassou.me		Remove
Marcus Denker <marcus.denker@inria.fr>	marcus.denker@inria.fr		Remove
Jorge Luis Ressia <ressia@iam.unibe.ch>	ressia@iam.unibe.ch		Remove
Clara Allende <clari.allende@gmail.com>	clari.allende@gmail.com		Remove

Reset

New Session Configure Halos Profile Memory XHTML 0/2 ms

Figure 1.1: Screenshot of the finished contact book application.

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

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

```
Contact class>>newNamed: aString email: aString2
   ^ self new
     setFullname: aString email: aString2;
     yourself
```

You will need to define a instance method (`setFullname:email:`) to set all the information to the receiver.

```
Contact>>setFullname: aString email: aString2
   fullname := aString.
```

```
  email := aString2
```

Note that this is just one way to do it. Another way is to use simple setter messsages. We need to define setter, because in Pharo the only way to modify the state of an instance from the outside is by sending it messages.

You can define a test to make sure that you can create an instance without error and that it contains the right information.

```
  TestCase subclass: #ContactText
    instanceVariableNames: ''
    classVariableNames: ''
    category: 'ContactBook'
```

```
  ContactText>>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 accessors.

```
  Contact>>fullname
    ^ fullname
```

```
  Contact>>email
    ^ email
```

If you inspect or print the `contact` variable created above, you will see that the string representation of `Contact` instances 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 `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>'
```

Don't forget to categorize the methods in dedicated protocols and to comment the class.

Note that in future extensions you could

The Contact Book

A contact book contains a collection of `contacts`. Let us create the `ContactBook` class with its instance variable `contacts`:

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

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

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

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

You can either define an `initialize` method (that is automatically invoked at instance creation time) or use lazy initialization. The following method uses lazy initialization.

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

To simplify further development, we define a default contact book with predefined contacts inside. We do this by adding `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: 'Jorge Luis Ressia' email: 'ressia@iam.
      unibe.ch');
      addContact: (Contact newNamed: 'Clara Allende' email: 'clari.allende@gmail.
      com');
      yourself
```

Don't forget to categorize the methods in dedicated protocols and to comment the class.

1.2 A First Web View

Now that we have the model, we need a web view. 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. Open the *Catalog browser* tool and search for Seaside. Install the stable version. As it can take a while you can also download a pre-installed version of Seaside available at https://ci.inria.fr/pharo-contribution/job/Seaside/.

Seaside is a component framework: Seaside web applications are built by aggregating components. Typically, an application consists of a top-level component delegating part of its rendering to sub-components. Our simple application will only consist of a top-level component represented by the `WAContactBook` class, subclass of `WAComponent`.

Create the `WAContactBook` class with a `contactBook` instance variable:

```
WAComponent subclass: #WAContactBook
 instanceVariableNames: 'contactBook'
 classVariableNames: ''
 category: 'ContactBook'
```

Rendering a Title

Every Seaside component class must override the `renderContentOn:` method to specify how a component is rendered. Define this method to 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 brush that can be parametrized to emit adequate XHTML. Here we request the heading brush and set it to level one. Then we specify the contents of the corresponding section.

The next step is to register the `WAContactBook` class to the `/contacts` URL path. This way you will be able to reach the application as on http://localhost:8080/contacts. Do this by implementing a class `initialize` method in `WAContactBook class`:

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

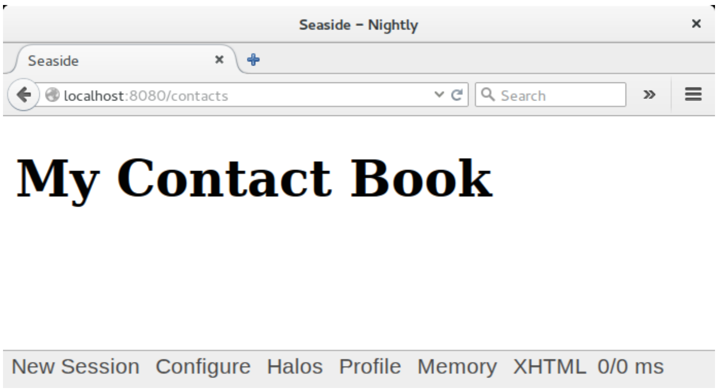


Figure 1.2: Screenshot of the contact book application title

Class initialize method are executed when the code is loaded in memory, now since the class is already loaded, we should executed it once manually:

```
WAContactBook initialize
```

The last step before getting something in the web browser is to start the web server. Open the `Seaside control panel` tool, add a `ZnZincServerAdaptor` on port 8080 and start it.

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 `updateRoot:` as follows.

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

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

Rendering a Table of Contacts

Below the title, we now want a table containing the contacts of the contact book. For this, we need to change the `renderContentOn:` method and add a few new methods. We will decompose the behavior we want to be able to reuse it if necessary. First we introduce a new method named `renderContactsOn:`.

```
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
```

The method `renderContactsOn:` defines a table with a table header and delegates to the the method `renderContact:on:`.

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

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

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

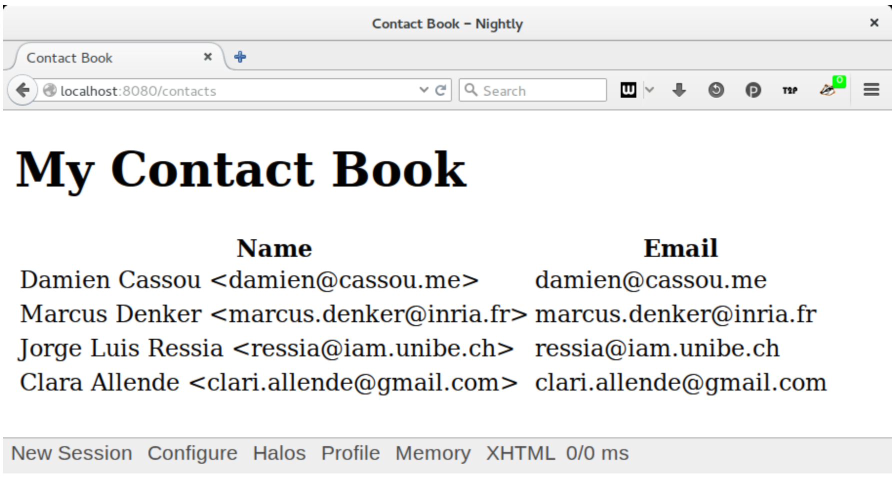
As we saw, the `renderContentOn:` method delegates the table rendering to the `renderContactsOn:` method. The latter creates a table with a heading row and delegates the contact rendering to the `renderContact:on:` method. This method renders a table row with the contact's details.

Some methods are used above but not yet defined. The `WAContactBook>>contactBook` method is an accessor to the `contactBook` instance variable. If the variable is not yet set, we can set it to a default contact book like this:

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

The `contactsDo:` method is implemented in the `ContactBook` class like this:

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



My Contact Book

Name	Email
Damien Cassou <damien@cassou.me>	damien@cassou.me
Marcus Denker <marcus.denker@inria.fr>	marcus.denker@inria.fr
Jorge Luis Ressia <ressia@iam.unibe.ch>	ressia@iam.unibe.ch
Clara Allende <clari.allende@gmail.com>	clari.allende@gmail.com

Figure 1.3: Screenshot of a contact book contacts

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

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 Bootstrap

The rendering can be visually improved by adding some Cascading Style Sheets (CSS). In the following, we use the Bootstrap framework that must be loaded in the image. Open the *Catalog browser* tool and search for Bootstrap. Install the stable version.

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

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

Do not forget to reinitialize it manually again:

```
WAContactBook initialize
```

The Bootstrap framework defines some special methods (such as `tbsContainer:`, `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 contactBook 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. In a real application, it is recommended to avoid using Bootstrapspecific methods such as `tbsContainer:` and `tbsTable:` to use Bootstrap mixins instead. We will not cover that in our tutorial though.

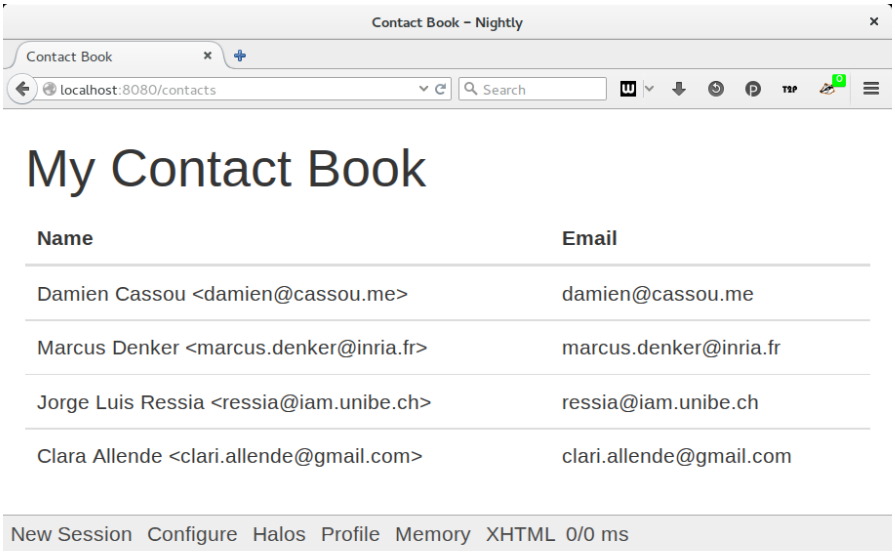
1.4 Finishing the Details

We will now add photos and buttons to our contact list to obtain the result in Figure 1.1.

Adding Photos

We will improve the contact book application by displaying photos next to each contact. We fetch these photos automatically fetched from the web using Gravatar or equivalent. Gravatar provides a web API to retrieve a photo from an email address:

```
Contact>>gravatarUrl
```



My Contact Book

Name	Email
Damien Cassou <damien@cassou.me>	damien@cassou.me
Marcus Denker <marcus.denker@inria.fr>	marcus.denker@inria.fr
Jorge Luis Ressia <ressia@iam.unibe.ch>	ressia@iam.unibe.ch
Clara Allende <clari.allende@gmail.com>	clari.allende@gmail.com

Figure 1.4: Screenshot of the contact book application with bootstrap

```
   ^ 'http://www.gravatar.com/avatar/', (MD5 hashMessage: email 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 contactBook contactsDo: [ :contact | self renderContact: contact on: html ] ]
```

Then we display the photo on the third column.

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

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

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

Adding Buttons

Finally, we add buttons to remove contacts one by one and to reset the contact book to the default one.

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:

```
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. To let the contact book user reset the contact book to its default state, we now add a reset button below the contact table:

```
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 renderButtonsOn: html ] ]
```

```
WAContactBook>>renderButtonsOn: html
  html
    tbsButtonGroup: [
      html tbsButton
        beDanger;
        callback: [ self resetContactBook ];
        with: 'Reset' ]
```

```
WAContactBook>>resetContactBook
  contactBook := nil
```

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

1.5 Summary

During this tutorial we defined a simple model and one simple web view. We follow a traditional development style where we define methods upfront. We could also have written tests are while they get executed define method directly in the debugger. Pharo developer often prefer this way because they go faster and the tests define more precise context.