

Chapter 11: Creating a Form

In this chapter we look at creating an HTML form element to collect information from the user.

1. Define a new component to edit events.

```
WAComponent subclass: #LBEventEditor
  instanceVariableNames: 'model'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'LosBoquitas'
```

a. Add a render method to show that we are displaying the component.

```
renderContentOn: canvas

  canvas heading: self class name.
```

b. Create an initialize method on the instance side.

```
initialize: anEvent

  self initialize.
  model := anEvent.
```

c. Now create an instance creation method *on the class side*. Note that we are calling 'basicNew' rather than 'new.' This is because 'new' would call 'initialize' on the instance, and we want to call 'initialize:' explicitly and let it call 'initialize.'

```
on: anEvent

  ^self basicNew
    initialize: anEvent;
    yourself.
```

2. Now we can return to the LBScheduleComponent and arrange to call our new editor.

a. Add an 'edit:' method to the instance side of LBScheduleComponent.

```
edit: anEvent

  | editor answer |
  editor := LBEventEditor on: anEvent.
  answer := self call: editor.
  answer
    ifTrue: [self inform: 'Edits were saved']
    ifFalse: [self inform: 'Edits were cancelled'].
```

- b. Modify `LBScheduleComponent` >>#`'whatReportColumn'` so that the column has a click block. Note that since we have the column definition in its own method we don't have to modify a large initialize method.

```
whatReportColumn

^WAreportColumn new
  title: 'What';
  selector: #what;
  clickBlock: [:each | self edit: each];
  yourself.
```

- c. In your web browser, click on the <Events> link to show the what field as a link. Click on any row and see that the schedule list is replaced with the event editor component (which simply displays some text).
3. Add true editing to the editor.
 - a. Modify `LBEventEditor` >>#`renderContentOn:` to lay out a table with headings and input fields. (Yes, we are using a table for formatting; the next step will refactor this method to avoid using a table.) Note that for this first round we are using only text fields so have set the 'when' field to be read only.

```
renderContentOn: canvas

canvas form: [
  canvas table: [
    canvas tableBody: [
      canvas tableRow: [
        canvas tableHeading: 'Who:'.
        canvas tableData: [
          canvas textInput
            value: model who;
            callback: [:value | model who: value].
        ].
      ].
    ].
    canvas tableRow: [
      canvas tableHeading: 'What:'.
      canvas tableData: [
        canvas textInput
          value: model what;
          callback: [:value | model what: value].
      ].
    ].
    canvas tableRow: [
      canvas tableHeading: 'When:'.
      canvas tableData: [
        canvas textInput
          value: model when printString;
          yourself.
      ].
    ].
  ].
].
```


Chapter 11: Creating a Form

```
renderWhatOn: canvas

| tagID |
canvas div: [
  canvas label
    for: (tagID := canvas nextId);
    with: 'What:'.
  canvas textInput
    id: tagID;
    value: model what;
    callback: [:value | model what: value].
].
```

```
renderWhenOn: canvas

| tagID |
canvas div: [
  canvas label
    for: (tagID := canvas nextId);
    with: 'When:'.
  canvas textInput
    id: tagID;
    value: model when;
    yourself.
].
```

```
renderWhereOn: canvas

| tagID |
canvas div: [
  canvas label
    for: (tagID := canvas nextId);
    with: 'Where:'.
  canvas textInput
    id: tagID;
    value: model where;
    callback: [:value | model where: value].
].
```

```
renderButtonsOn: canvas

canvas div: [
  canvas cancelButton
    callback: [self answer: false];
    with: 'Cancel'.
  canvas submitButton
    callback: [self answer: true];
    with: 'Save'.
].
```

- b. Modify 'renderContentOn:' to call the new methods.

```
renderContentOn: canvas

canvas form
  class: 'eventEditor';
  with: [
    self
    renderWhoOn: canvas;
    renderWhatOn: canvas;
    renderWhenOn: canvas;
    renderWhereOn: canvas;
    renderButtonsOn: canvas;
    yourself.
  ].
```

- c. View this in a browser and observe that the layout has each <div> element on a new line. Now we can edit the CSS to make this a bit more fancy. Add the following lines to the text in LbFileLibrary>>#'boquitasCss' inside the existing string (i.e., after the first single quote character and before the last single quote character).

```
.eventEditor { display: table; }
.eventEditor > div { display: table-row; }
.eventEditor > div > * { display: table-cell; }
```

- d. Refresh the page in your web browser, and note that the positioning is now controlled by the CSS. We have separated the text markup (HTML) from the style (CSS). This is considered a much better way to build web sites, but does rely on some CSS features that might not be supported in older browsers. For example, Internet Explorer 7 (and earlier) does not recognize table formatting.

Of course, once we start down the path of separating content from style we need to learn CSS and how it interacts with HTML.

- e. Notice that the label width is unusually wide. It turns out that this is because the buttons in the fifth row are lumped together in the first column.

- f. To move the buttons to the second column, add an empty label before the buttons.

```
renderButtonsOn: canvas

canvas div: [
  canvas label: [canvas space].
  canvas cancelButton
    callback: [self answer: false];
    with: 'Cancel'.
  canvas submitButton
    callback: [self answer: true];
    with: 'Save'.
].
```

- g. We intended that each element inside a div inside the eventEditor would be treated as a table-cell. It turns out that the CSS specification (see <http://www.w3.org/TR/CSS21/conform.html#conformance>) allows browsers to ignore CSS properties for form controls (including input fields like buttons): “CSS 2.1 does not define which properties apply to form controls and frames, or how CSS can be used to style them.”

- h. If you want the buttons to be in separate columns, enclose them in another element, such as span.

```
renderButtonsOn: canvas

  canvas div: [
    canvas span: [
      canvas cancelButton
        callback: [self answer: false];
        with: 'Cancel'.
    ].
    canvas span: [
      canvas cancelButton
        callback: [self answer: true];
        with: 'Save'.
    ].
  ].
```

5. Adding a new event.

- a. Modify `LBScheduleComponent>>#renderContentOn:` to add an `<Add>` link.

```
renderContentOn: canvas

  listComponent rows: LBEvent events asSortedCollection.
  canvas render: listComponent.
  canvas anchor
    callback: [self add];
    with: 'Add'.
```

- b. Try it out and note that you get a walkback because the `add` method is not implemented.
- c. Add the following method:

```
add

  | event editor |
  event := LBEvent new.
  editor := LBEventEditor on: event.
  (self call: editor) ifTrue: [
    LBEvent events add: event.
  ].
```

- d. Refresh your browser and try adding an event. Try opening the editor but cancelling the new event.
- e. Note how we are reusing a component—to add and to edit. The component doesn't know how it is being used which provides for good encapsulation.
- f. Note also that the `answer` is useful in this case. If the user pressed the Cancel button, we don't want to add the new event. Before going further let's cleanup

LBScheduleComponent>>#’edit:’ so that we don’t alert the user to whether the ‘Cancel’ or ‘Save’ button was clicked. Note how much simpler the method is now.

```
edit: anEvent

self call: (LBEventEditor on: anEvent).
```

6. Edit ‘when’ with WADateTimeSelector. We left the ‘when’ field as read only since we are storing an instance of DateAndTime (rather than an instance of String). Let’s give this editor some more usability.
 - a. As we discovered in Chapter 7, Seaside provides a number of sample components that can be used to present typical information on a web page. Change the schema for LBEventEditor to add another instance variable.

```
WComponent subclass: #LBEventEditor
  instanceVariableNames: 'model dateTimeSelector'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'LosBoquitas'
```

- b. Modify the ‘initialize:’ method to obtain a new component and set its initial value.

```
initialize: anEvent

self initialize.
model := anEvent.
dateTimeSelector := WADateTimeSelector new
  dateAndTime: model when;
yourself.
```

- c. Modify the ‘renderWhenOn:’ method to use the new component. The new component is enclosed in a span element so that the label can be associated with the component.

```
renderWhenOn: canvas

| tagID |
canvas div: [
  canvas label
    for: (tagID := canvas nextId);
    with: 'When:'.
  canvas span
    id: tagID;
    with: [canvas render: dateTimeSelector].
].
```

- d. Since we have a subcomponent we need to add the ‘children’ method to our subclass to let Seaside know about the new component. This will ‘override’ the ‘children’ method contained in the superclass that would return the wrong value (an empty Array).

Chapter 11: Creating a Form

```
children
  ^Array with: dateTimeSelector.
```

- e. Now we need some way to get the value out of the component in case the user changed the value. Since we are simply rendering a subcomponent, we don't have a 'callback:' that we can add to it. Instead, we need to do something when the 'Save' button is clicked. Modify the 'renderButtonsOn:' method to call a new 'save' method.

```
renderButtonsOn: canvas

  canvas div: [
    canvas span: [
      canvas cancelButton
        callback: [self answer: false];
        with: 'Cancel'.
    ].
    canvas span: [
      canvas submitButton
        callback: [self save];
        with: 'Save'.
    ].
  ].
```

- f. Add the new 'save' method.

```
save

  model when: dateTimeSelector dateAndTime.
  self answer: true.
```

7. Edit 'who' with a drop-down list. Often we want to constrain the value of a field to something taken from a list. This will demonstrate how to do that.
 - a. Add a method to LBEvent that returns a list of allowed values for 'who.'

```
whoList

  ^#('players' 'family' 'guests' 'staff').
```

- b. Modify LBEvent>>#initialize' to use the new list.

```
initialize

  super initialize.
  who := self whoList first.
  what := 'practice'.
  when := DateAndTime noon.
  where := 'field'.
```

Chapter 11: Creating a Form

- c. Now return to LBEventEditor and edit 'renderWhoOn:' so that we create a <select> element with a series of <option> elements (view the source if you are curious).

```
renderWhoOn: canvas

| tagID |
canvas div: [
  canvas label
    for: (tagID := canvas nextId);
    with: 'Who:'.
  canvas select
    id: tagID;
    selected: model who;
    list: model whoList;
    callback: [:value | model who: value].
].
```

8. Edit 'what' with a single-select list.
 - a. Add a method to LBEvent that returns a list of allows values for 'what.'

```
whatList

^#('practice' 'registration' 'game' 'staff meeting' 'party').
```

- b. Modify LBEvent>>#initialize' to use the new list.

```
initialize

super initialize.
who := self whoList first.
what := self whatList first.
when := DateAndTime noon.
where := 'field'.
```

Chapter 11: Creating a Form

- c. Now return to LBEventEditor and edit 'renderWhatOn:' so that we create a <select> element with a series of <option> elements (view the source if you are curious). Note that the only difference from a drop-down list is that the size is specified and is greater than one.

```
renderWhatOn: canvas

| tagID |
canvas div: [
  canvas label
    for: (tagID := canvas nextId);
    with: 'What:'.
  canvas select
    id: tagID;
    selected: model what;
    list: model whatList;
    size: 4;
    callback: [:value | model what: value].
].
```

9. Edit 'where' with a multi-line text area.
 - a. Modify 'renderWhereOn:' to replace the textInput with a textArea.

```
renderWhereOn: canvas

| tagID |
canvas div: [
  canvas label
    for: (tagID := canvas nextId);
    with: 'Where:'.
  canvas textArea
    id: tagID;
    value: model where;
    callback: [:value | model where: value];
    yourself.
].
```

- b. Modify the CSS to make the field larger. Edit LBFFileLibrary>>#'boquitasCss' to add the following line. (The width will vary depending on your browser's selection of a font for the text area. Because of this it might be better to use a pixel width.)

```
.eventEditor textarea { height: 4em; width: 30em; }
```

Chapter 11: Creating a Form

10. Make 'when' more readable in the table.

- a. Add a method to LBEvent to return a more readable version of the when value.

```
whenString  
  
^when asDate printString , ' ' , when asTime printString.
```

- c. Modify LBScheduleComponent>>#'whenReportColumn' to use the new method.

```
whenReportColumn  
  
^WAScheduleComponent new  
  title: 'When';  
  selector: #whenString;  
  clickBlock: nil;  
  yourself.
```

11. In order to demonstrate checkboxes, radio buttons, and some JavaScript interaction with CSS, we will add an attribute to LBEvent to keep track of whether a game is home or away.

- a. Add 'gameType' as an instance variable to LBEvent. We will treat this value as a three-state flag: nil (for 'not a game'), #'home' (a Symbol, or string singleton), and #'away' (also a Symbol). The initial value is nil.

```
Object subclass: #LBEvent  
  instanceVariableNames: 'who what when where gameType'  
  classVariableNames: ''  
  poolDictionaries: ''  
  category: 'LosBoquitas'
```

- b. Using the class refactoring menu, add an accessor for the new variable.
- c. We will have three form elements (a checkbox on one line and two radio buttons on another line) to capture this data (three radio buttons would be more efficient, but this gives a good demo!). Because of the way Seaside processes the callbacks associated with these fields, we won't simply assign a value to the model during any one callback. Instead we will have two instance variables in the editor that capture various pieces of state that we will merge as part of the save process. To do that, add 'isGame' and 'gameType' to the definition of LBEventEditor.

```
WAScheduleComponent subclass: #LBEventEditor  
  instanceVariableNames: 'model dateTimeSelector isGame gameType'  
  classVariableNames: ''  
  poolDictionaries: ''  
  category: 'LosBoquitas'
```

- d. Modify `LBFileLibrary>>#'`boquitasCss'`'` to add a line allowing a `<div>` element to be hidden if it has a class attribute of `'hidden.'`

```
.eventEditor div.hidden { display: none; }
```

- e. Return to `LBEventEditor` and modify `'renderContentOn:'` to call a couple new methods (you will be prompted that the selector does not exist; go ahead and confirm the spelling).

```
renderContentOn: canvas

canvas form
  class: 'eventEditor';
  with: [
    self
      renderWhoOn: canvas;
      renderWhatOn: canvas;
      renderWhenOn: canvas;
      renderWhereOn: canvas;
      renderIsGameOn: canvas;
      renderGameTypeOn: canvas;
      renderButtonsOn: canvas;
      yourself.
  ].
```

- f. Add `'renderIsGameOn:'` to `LBEventEditor`. Note that this method has JavaScript code that is added to the checkbox. The JavaScript finds the element created below and changes its class depending on whether the checkbox is checked or not. Based on the class, the CSS defined above will be applied.

```
renderIsGameOn: canvas
  | script tagID |
  script := "Workaround for IE bug (thanks to Stephan Eggermont)"
    'document.getElementById("idGameType").setAttribute("class",' ,
      'this.checked? ":"hidden");' ,
    'document.getElementById("idGameType").setAttribute("className",' ,
      'this.checked? ":"hidden");'.
  canvas div: [
    canvas label
      for: (tagID := canvas nextId);
      with: 'Is Game:'.
    canvas checkbox
      id: tagID;
      value: model gameType notNil;
      callback: [:value | isGame := value];
      onClick: script;
      yourself.
  ].
```

- g. Add 'renderGameTypeOn:' to LBEventEditor. Note that the HTML class attribute of the div is set to 'hidden' or nil depending on whether gameType is nil. This div element will have its class changed by the JavaScript code above.

```
renderGameTypeOn: canvas

| tagID group |
canvas div
  id: 'idGameType';
  class: (model gameType isNil ifTrue: ['hidden'] ifFalse: [nil]);
  with: [
    canvas label
      for: (tagID := canvas nextId);
      with: 'Type:'.
    canvas span
      id: tagID;
      with: [
        group := canvas radioGroup.
        canvas radioButton
          id: (tagID := canvas nextId);
          group: group;
          selected: model gameType ~= #'away';
          callback: [gameType := #'home'].
        canvas label
          for: tagID;
          with: 'Home'.
        canvas radioButton
          id: (tagID := canvas nextId);
          group: group;
          selected: model gameType = #'away';
          callback: [gameType := #'away'].
        canvas label
          for: tagID;
          with: 'Away'.
      ]].
```

12. Try out the various combinations and note how the game type is displayed and hidden based on the checkbox value. This demonstrates the use of JavaScript in Seaside. Note, however, that the value is not saved (thanks to Stephan Eggermont for noticing this and suggesting a fix). Edit LBEventEditor>># 'save' as follows.

```
save

model when: dateTimeSelector dateAndTime.
model gameType: (isGame == true ifTrue: [gameType] ifFalse: [nil]).
self answer: true.
```

13. Try out the various combinations and then save your Squeak image.