PART I

Welcome to XForms

CHAPTER 1

XML Powered Web Forms

Since their inception in 1993, HTML forms have come to form the underpinnings of user interaction on the World Wide Web (WWW). The convenience afforded by the ability to deploy applications at all levels of complexity on the Web, and thereby provide consistent end-user access to information and application services via a universal Web client, created a platform-independent environment for electronic commerce across the Internet.

However, building on the essential simplicity of HTML forms has resulted in an extremely complex Web programming model; today, Web application developers are forced to work at different levels of abstraction—and, consequently, duplicate application and business logic in multiple programming languages—in order to deliver a satisfactory end-user experience. These problems become even more complex given the need to perform electronic transactions with a variety of different devices and user interface modalities. As we deploy Web access to software at all levels of complexity ranging from business back-ends to simple electronic transactions, these issues can be better addressed by revisiting the design of HTML forms that are the essential underpinnings of the transactional Web.

XForms—a revision to the existing HTML forms technology developed by the World Wide Web Consortium (W3C)[1]—builds on the advantages of XML to create a versatile forms module that can stand the Web in good stead for the next decade.

[1] http://www.w3.org

Section 1.1 traces the evolution of Web applications and describes the problems inherent in the present-day Web programming model that motivated the work on XForms. We design a simple questionnaire application using HTML forms in Section 1.2 and enumerate the software components needed to deploy the questionnaire on the Web. We then recast this application in XForms in Section 1.3 to give a bird's-eye view of the various components that constitute W3C XForms. We conclude this chapter with a summary of key XForms features and their benefits in Section 1.4. The remaining chapters of this book will cover these components in detail along with numerous examples that illustrate their use.

1.1 Background

HTML forms for creating interactive Web pages were introduced in 1993. The addition of a handful of simple markup constructs that allowed HTML authors to create input fields and other user interaction elements enabled Web sites to deploy Web pages that could *collect* user input as simple name-value pairs. The values input by the user were transmitted via HTTP and processed on the server via Common Gateway Interface (CGI) scripts. This new innovation spawned a multitude of interactive Web sites that experimented with these constructs to create interfaces ranging from simple user surveys to prototype shopping applications.

As electronic commerce on the Web gained momentum during the mid-90s, Web developers moved from experimenting with HTML forms to deploying real end-user applications to the Web. These forms-powered HTML pages provided a Web interface to standard transaction oriented applications. In this programming model, the Web developer authored a user interface in HTML and created corresponding server-side logic as CGI scripts that processed the submitted data before communicating it to the actual application. The combination of the HTML user interface and the server-side logic used to process the submitted data is called the *Web application*. The *Web application* in turn communicates user input to the application, receives results, and embeds these results in an HTML page to create a user interface to be delivered as a *server response* to the user's Web browser.

This method of deploying electronic transactions on the Web where end users could interact via a standard browser created a platform-independent environment for conducting electronic commerce across the Internet. Examples of such Web applications range from CNN's opinion polls to electronic storefronts like amazon.com. During this period, the server-side components making up Web applications gained in sophistication. The simple-minded CGI script came to be superseded by server-side technologies such as Java servlets, specialized Perl

environments optimized to run in a server context such as Apache's modperl,[2] and a plethora of other server-side processing tools. These technologies were designed to aid in the processing of user data submitted via HTTP and the generation of dynamic content to be transmitted as the *server response*. The underlying HTML forms technology however remained unchanged—except for the addition of file upload and client-side Javascript to enable a more interactive end-user experience on the client.

As the *e*commerce Web matured, vendors rushed to deploy server-side middleware and developer tools to aid in the authoring and deployment of interactive Web applications. By this time, such tools were almost mandatory, since the essential simplicity of HTML forms resulted in scalability problems when developing complex Web applications.

Consider some of the steps that are carried out by a typical Web application. User data obtained via HTTP is validated at the server within servlets or other server-side software. Performing such validation at the server after the user has completed the form results in an unsatisfactory end-user experience when working with complex forms; the user finds out about invalid input long after the value is provided. This in turn is overcome by inserting validation scripts into the HTML page. Notice that such scripts essentially duplicate the validation logic implemented on the server side. This duplication often has to be repeated for each supported browser to handle differences in the Javascript environment.

Using this programming model, developing, deploying, and maintaining a simple shopping application on the Web require authoring content in a variety of languages at several different levels of abstraction. Once developed, such Web applications remain expensive to maintain and update. Notice that the move from experimenting with Web interaction technologies to deploying real-world applications to the Web brings with it a significant change; in most real-world scenarios, the application to be deployed to the Web *already* exists. Even in the case of new applications, only a portion of the application in question gets deployed to the Web. As an example, only the electronic storefront of a shopping site gets deployed to the Web; such shopping sites are backed by software that manages customer information, product catalogs, and other business objects making up an electronic store. Thus, deploying complex interactive sites involves creating the business logic and then exposing relevant portions of this application to a *Web application* that creates a user interface accessible via a Web browser.

[2] http://www.apache.org/modperl

One way of simplifying this development process is to make business applications themselves aware of the need to deliver a Web user interface. This approach is followed by many of today's popular middleware solutions, with some commercial database engines going so far as to incorporate a Web server into the database. However, making back-end business applications aware of the details of the user interface markup can make systems difficult to evolve and maintain. The resulting lack of separation of concerns ties Web applications to a particular back-end system.

As we deploy Web access to software at all levels of complexity ranging from business applications to electronic transactions, the problems outlined earlier can be better addressed by revisiting the essential underpinnings of the transactional Web. Today, Web applications need to be accessible from a variety of access devices and interaction modalities—Web applications may be accessed from a variety of clients ranging from desktop browsers to smart phones capable of delivering multimodal[3] interaction. Thus, a travel application that is being deployed to the Web needs to be usable from within a desktop browser, a Personal Digital Assistant (PDA), and a cell phone equipped with a small display. Thus, the interface needs to be usable when interacting via a graphical or speech interface.

Notice that the problems with HTML forms outlined earlier become even more serious when confronted with the need to perform electronic transactions with a variety of different end-user devices and user interaction modalities.[4] W3C XForms[5]—a revision to the existing HTML forms technology from 1993—builds on the advantages of XML to create a powerful forms module that can stand the Web in good stead for the next decade.

1.2 A Simple Web Application

This section introduces a simple Web application developed using today's HTML forms and illustrates the various software modules that would be authored on the client and server to deploy a complete end-to-end solution. This sample application will be recast using XForms in the remaining sections of this chapter to illustrate the advantages inherent in the various components making up the XForms architecture.

[3] The use of multiple means of interaction, for example, synchronized spoken and visual interaction, is called *multimodal* interaction.

[4] A user interaction modality denotes one of possibly many different means for providing input and perceiving output.

[5] http://www.w3.org/tr/xforms

1.2.1 Questionnaire Form

Consider a questionnaire application that collects the following items of user information:

name	User's first and last names as strings
age	User's age as a number
gender	User's gender: `m` for male and `f` for female
birthday	Fields making up the user's date of birth
address	Fields making up the user's mailing address
email	User's email address as a string
ssn	User's Social Security number, if available

Data collected by this questionnaire will be communicated to a survey application that imposes the following validity constraints on the data:

- All requested data items *must* be provided.
- Values *must* be legal for each field; for example, value of field `age` must be a number; fields making up the value of `birthday` need to be valid date components.
- Field `ssn` must contain a 9-digit Social Security number; if none is available, a default value of `000-000-000` must be used.

1.2.2 Developing the Web Application

The Web developer models the various items of data to be collected as simple name-value pairs; for example, `age=21`. Compound data items like `address` and `name` are made up of subfields, and in modeling these as simple string value pairs, the developer introduces additional field names such as `name.first`. Here is a list of the field names the developer might create for this application:

```
   name.first          name.last            age
address.street     address.city       address.zip
 birthday.day     birthday.month     birthday.year
    email               ssn              gender
```

Next, the developer creates the server-side software component that will receive the submitted data as name-value pairs—this typically starts off as a stand-alone CGI script that evolves to encompass more and more functionality as the application gains in sophistication. Functions performed by this component include:

- Produce the HTML page that is displayed to the user; this generates the initial user interface and displays default values if any.
- Receive submitted data as name-value pairs via HTTP.
- Validate received data to ensure that all application constraints are satisfied.
- Generate a new HTML page that allows the user to update previously supplied values if one or more fields are found to be invalid.
- Make all fields *sticky*, that is, user does not lose previously supplied values during client-server round-trips.
- Marshal the data into a structure that is suitable for the survey application when all fields have valid values. This is necessary because intermediate fields created by the Web developer such as `name.first` may not match what the survey application expects.
- Transmit the collected data to the back-end, process the resulting response, and communicate the results to the user by generating an appropriate HTML page.

1.2.3 Developing the User Interface

The user interface is delivered to the connecting browser by producing appropriate HTML markup and transmitting this markup via HTTP to the user's browser. Interaction elements, for example, input fields, are contained in HTML element ⟨**form**⟩ that also specifies the `URI` where the data is to be submitted; the HTTP `method` to use, for example, `GET` or `POST`; and details on the encoding to use when transmitting the data. HTML markup for user interface controls, for example, ⟨**input**⟩, is used to create input fields in the resulting user interface. This markup refers to the field names defined earlier, for example, `name.first`, to specify the association between the field names defined by the Web developer and the values provided by the end user. The markup also encodes default values, if any, for the various fields. Notice the tight binding between the HTML markup and the server-side logic developed earlier with respect to the following:

- Field names used in the HTML markup need to match the names used in the server-side component.
- Making all fields *sticky*, that is, retaining user supplied values during multiple client-server round-trips requires that the previously received values be embedded in the generated HTML.

To achieve this, early Web applications produced the HTML markup from within the CGI script. Though this works in simple cases, this approach does not scale for creating more complex Web applications. This is because of the lack of separation of concerns that results from mixing user interface generation with server-side application logic. Maintaining and evolving the user interface markup require the developer to edit the server-side component. However, the skills required to edit server-side software components are different from those needed to design good user interfaces. This increases the cost of designing good Web user interfaces and makes it tedious to keep the result synchronized with the software components that implement the interaction.

1.2.4 A More Sophisticated Implementation

The lack of separation of concerns that arises when incorporating presentational markup within executable CGI scripts is typically overcome by developing Web applications using more sophisticated server-side technologies such as Hypertext Preprocessor (PHP),[6] Java Server Pages (JSP)[7] or Active Server Pages (ASP).[8] All of these technologies follow a Model, View, Controller (MVC) decomposition by factoring out the user interface markup from the program code that implements the server-side application logic. Thus, the user interface is created as an XML[9] document with special tags that invoke the appropriate software components when processed by the server. As a result, the user interface designer can work with intuitive authoring tools that generate XML markup while the software developer builds software objects using traditional programming tools.

Thus, the simple Web application developed earlier would be created as a set of software objects that implements the validation and navigation logic and a set of markup pages used to generate the user interface at each stage of the interaction—see Figure 1.1 for a high-level overview of the resulting components and their interdependencies.

Higher level Web application frameworks, such as struts[10] based on JSP and servlets, provide further abstractions that allow the Web developer to create the application by defining the *model* as Java beans, defining the user interaction *views* as JSP pages, and wiring up the resulting *model* and *views* via a standard *controller* component that manages the navigation among the various views.

[6] http://www.php.net
[7] http://java.sun.com/products/jsp
[8] http://www.asp.net/
[9] http://www.w3.org/tr/REC-xml.html
[10] http://jakarta.apache.org/struts

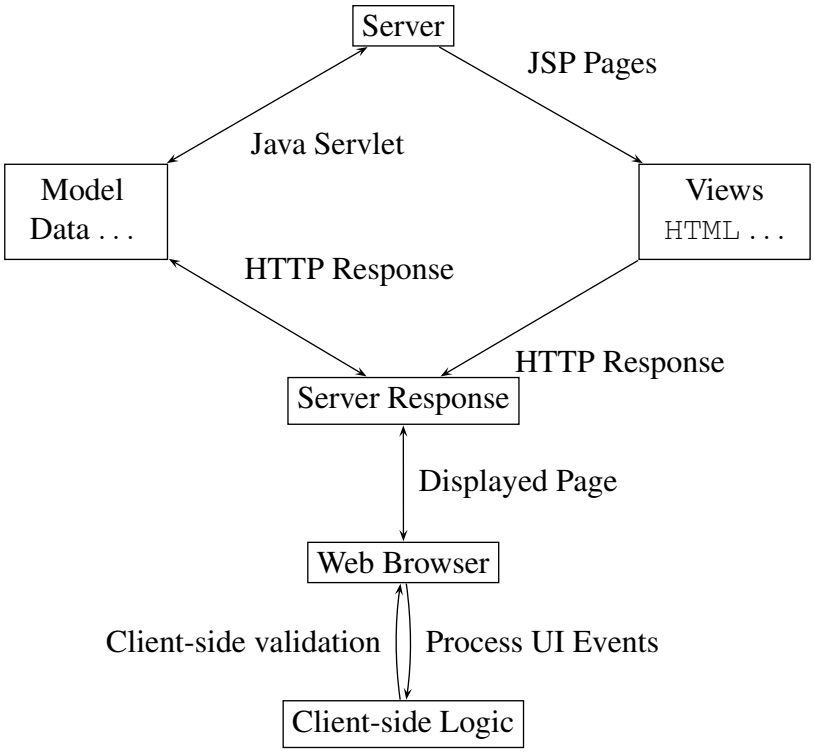

Figure 1.1. A simple Web application made up of software components that together implement server-side validation and client-side user interaction.

Finally, an interface where the user learns about invalid input only after submitting the data to the server can prove unsatisfactory. To achieve a rich interactive end-user experience, simple tests such as checking for valid values for `age` or `birthday` are better performed on the client during user interaction to provide immediate feedback. This is achieved by embedding client-side validation scripts in the HTML markup. Notice that such validation code duplicates the validation rules already authored as part of the server-side validation component, but this time in a client-specific scripting language. Variations in the scripting environment provided by Web browsers on various platforms make such scripts hard to develop, test, and maintain.

1.3 XForms Components

The previous section traced the development of a simple Web application using present-day technologies predicated by the HTML forms architecture. The questionnaire application evolved from a simple stand-alone CGI script to a more

complex Web application consisting of software components dedicated to managing the application state within a servlet, markup pages designed to create the user interaction, and navigation components designed to connect the various views with the application state. In doing so, we saw that the Web developer needed to implement significant application-specific functionality in custom software to deliver the questionnaire to a Web browser.

XForms leverages the power of using XML in modeling, collecting, and serializing user input. XForms has been designed to abstract much of this functionality into a set of components that enable the Web developer to rely on a standard set of services and off-the-shelf XML tools when developing the final Web application. This allows the Web developer to focus on key aspects of the application and rely on the underlying XForms platform for the following services:

- Produce user interfaces appropriate for the connecting device.
- Provide interactive user feedback via automated client-side validation.
- Validate user input on the server automatically.
- Marshal user input on the server into a structure suitable for the back-end application.

Based on what has been observed in the design of today's Web applications and the need to deliver such applications to an ever-increasing array of end-user devices, the overall XForms architecture has been divided into the following components. A key feature of this MVC decomposition is a clear separation of the *model* from its final *presentation*.

Model All nonpresentational aspects of a Web application are encapsulated by the XForms data model. The data model incorporates an XML instance that holds user input, the constraints used to validate this input, and the necessary metadata about how this user input should be communicated to the Web server.

UI XForms defines a user interface vocabulary that consists of abstract controls and aggregation constructs used to create rich user interfaces. The user interface vocabulary is designed to capture the underlying intent of the user interaction, rather than its final presentation on any given device or in any specific modality. This makes it possible to deliver XForms-based Web applications to different devices and modalities.

Submit This allows the Web application author to specify where, how, and what pieces of data to submit to the Web server. It also permits the application

developer to specify what actions to take upon receiving a response from the server.

1.3.1 XForms Overview

Next, we reexamine the questionnaire and recast it as an XForms application. The questionnaire will be created as an XHTML document that contains the XForms model and user interface components. The following subsections detail each of these components and show how they are used within an XHTML document. The XForms model (contained in ⟨**model**⟩) is placed within XHTML element ⟨**head**⟩. XForms user interface controls create the user interaction and appear within the body of the document, that is, within XHTML element ⟨**body**⟩, and are rendered as part of the document content. In this overview, we will describe a few of the XForms user interface controls to give the reader a feel for XForms markup; subsequent chapters will detail all the constructs defined in XForms 1.0.

1.3.2 XForms Model

As before, we start by enumerating the various items of user information collected by the Web application. Since we are now using XML, we no longer need restrict ourselves to a flat data model consisting of a set of untyped name-value pairs. Instead, we encapsulate the information collected from the user in a structured XML document. This is called the *XML instance*. Further, we pick the structure of this XML instance to suit the survey application—see Figure 1.2.

```
<model xmlns="http://www.w3.org/2002/xforms" id="p1">
  <instance>
    <person xmlns="">
      <name><first/><last/></name>
      <age/><email/>
      <address><street/><city/><zip/></address>
      <birthday><day/><month/><year/></birthday>
      <ssn>000-000-000</ssn>
      <gender>m</gender>
    </person>
  </instance>
</model>
```

Figure 1.2. Element ⟨**instance**⟩ declares the XML template that holds user input and default values.

```
<model xmlns="http://www.w3.org/2002/xforms"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  schema="person.xsd" id="p1">
  <instance>
    <person xmlns="">...</person>
  </instance>
</model>
```

Figure 1.3. Constraining instance data by specifying an XML Schema.

Notice that compound data items such as `address` are now modeled to reflect the structure of the data, unlike when using flat name-value pairs. This also obviates the need to introduce intermediate fields to hold portions of the user data and the subsequent need to marshal such intermediate fields into the structure required by the application.

Next, this XML instance can be annotated with the various constraints specified by the application, for example, `age` should be a number. When using XML, such constraints are typically encapsulated in an XML Schema[11] document that defines the structure of the XML instance—see Figure 1.3.

Alternatively, such type constraints can be specified as part of the instance using attribute *xsi:type*[12] as shown in Figure 1.4. Both techniques have their place in Web development; the former is especially relevant when creating Web applications that access existing business logic, and the latter is useful when creating a one-off Web application with relatively simple type constraints.

In the questionnaire application, the constraints shown in Figure 1.4 encapsulate type constraints—the default type is `string`. Complex schemas typically encapsulate more constraints, such as specifying the rules for validating a 9-digit Social Security Number or specifying the set of valid values for the various fields. Note that this example has been kept intentionally simple—later chapters will build real-world examples where we will use the full richness of the built-in type mechanisms provided by XML Schema.

The advantage of specifying such constraints using XML Schema is that the developer can then rely on off-the-shelf XML parsers to validate the data instance against the supplied constraints. With the increasing adoption of XML Schema by database vendors, complex business applications are likely already to have an XML

[11] http://www.w3.org/tr/xschema
[12] Attribute *xsi:type* is defined by XML Schema.

```
<model id="p1" xmlns:xsi=
  "http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.w3.org/2002/xforms"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <instance>
    <person xmlns="">
      <name><first/><last/></name>
      <age xsi:type="xsd:number"/>
      <birthday>
        <day xsi:type="xsd:number"/>
        <month xsi:type="xsd:number"/>
        <year xsi:type="xsd:number"/>
      </birthday>
      <address>
        <street/><city/><zip/>
      </address>
      <email/><ssn>000-000-000</ssn>
    <gender>m</gender></person>
</instance></model>
```

Figure 1.4. XML representation of the data collected by a questionnaire application, along with some simple type constraints.

Schema definition for the data model, and the developer can leverage such existing assets when creating a Web application. XML processor xerces[13] is available from the Apache project implements XML Schema and can be used to validate data collected on the server.

Finally, the constraints on the data instance are now encapsulated in declarative XML—as opposed to imperative program code—thus making it easier to maintain and revise these constraints using XML-aware tools without having to reprogram the application.

1.3.3 XForms User Interface

This section creates a sample XForms user interface for the questionnaire application and binds this user interface to the XForms data model defined in the previous section. XForms user interface markup appears within XHTML element ⟨**body**⟩

[13] http://xml.apache.org/xerces-j

along with other document markup. Notice that because of the separation of the model from the user interaction, XForms user interface markup can appear *anywhere* within the contents of XHTML element ⟨**body**⟩; in contrast, when using HTML forms, user interface controls can appear only within element ⟨**form**⟩.

In the questionnaire application, XForms user interface control ⟨**input**⟩ can be used to collect each item of data. User interface control ⟨**input**⟩ is intentionally designed to be generic. The type of information available about the underlying data item, for example, `birthday` is a *date*, can be used to advantage in generating a user interface representation that is appropriate to the connecting device, for example, rendering it as a calendar on a desktop browser. Notice that in addition to making the resulting user interface customizable for the connecting device, this design provides a rich level of accessibility for supporting users with different needs.

UI Control Input

Here, we review different aspects of UI control ⟨**input**⟩ in some detail; later chapters will review all of the XForms user interface controls. See Figure 1.5 for the markup that creates the input field for obtaining the user's age and Figure 1.6 for the resulting user interface. XForms controls encapsulate the following pieces of information

```
<input xmlns="http://www.w3.org/2002/xforms"
  xmlns:ev="http://www.w3.org/2001/xml-events"
  model="p1" ref="/person/age" class="edit"
  ev:event="DOMActivate" ev:handler="#speak"
  accesskey="a"><label>Age</label>
  <help>Specify your age as a number e.g., 21</help>
  <hint>How young are you?</hint>
  <alert>The age you specified,
    <output  ref="/person/age"/>is not a valid age.
</alert></input>
```

Figure 1.5. User interface control for obtaining the user's age.

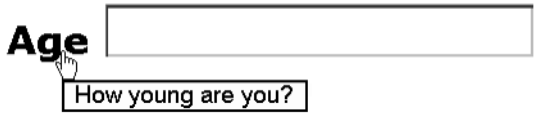

Figure 1.6. User interface for obtaining the user's age.

needed to render the interaction and connect the result to the appropriate portion of the XForms data model:

- Binding attributes that wire control to model
- Metadata for giving feedback to the user
- Wiring up of events and event handlers
- Presentation hints
- CSS style rules
- Keyboard shortcuts and navigation hints

Figure 1.5 illustrates the following XForms features:

Binding Attributes *model* and *ref* on element ⟨**input**⟩ specify the portion of the instance to be populated by the value obtained via this control. Attribute *model* gives the `id` of the `person` model; Attribute *ref* addresses node `/person/age` of this instance. The syntax used to address portions of the instance is defined by XPath.[14]

Metadata Elements ⟨**label**⟩, ⟨**help**⟩, ⟨**hint**⟩, and ⟨**alert**⟩ encapsulate metadata to be displayed to the user at various times. Notice that in XForms the label for the user interface control is *tightly bound* to the associated control; this is an extremely useful accessibility feature. Elements ⟨**hint**⟩ and ⟨**help**⟩ encapsulate tool tip text and detailed help to be displayed upon request. Finally, element ⟨**alert**⟩ holds the message to be displayed in the case of invalid input. Notice that the alert message uses element ⟨**output**⟩ to access the value supplied by the user. Element ⟨**output**⟩ uses XPath to address the relevant portion of the data model.

For simplicity, this example has shown elements ⟨**label**⟩, ⟨**hint**⟩, and ⟨**help**⟩ with inline content. For more complex applications, the contents of these elements can be specified indirectly via a URI by using attribute *src*. This feature can be used to advantage in localizing XForms-based Web applications by factoring all such messages into an XML file, referring to portions of that XML file using URIs within the XForms Web application, and loading these XML files to provide locale-specific messages.

[14] http://www.w3.org/tr/xpath

Eventing	Attribute *ev:event* on element ⟨**input**⟩ sets up control ⟨**input**⟩ to respond to event DOMActivate by calling the handler located at `#speak`. This uses the syntax defined in XML Events[15] for authoring DOM2 Events and is described in Section 2.3.
Presentation	Hints can be provided via attribute *appearance*.
Styling	Attribute *class* specifies a Cascading Style Sheet (CSS[16]) style to use for styling this control. CSS is a style sheet language that allows authors and users to attach style (for example, fonts, spacing, and aural cues) to structured documents (for example, XHTML documents).
Navigation	Attribute *accesskey* specifies an accelerator key for navigating to this control. This was an accessibility feature first introduced in HTML and has been incorporated into XForms.

UI Control select1

The field corresponding to the user's gender can have one of two legal values, `m` or `f`. The user *must* pick one of these values. Using traditional HTML forms, the corresponding user interface would be authored as a group of radio buttons. Notice that the HTML design hard-wires a particular presentation (radio buttons) to the underlying notion of allowing the user to select *one and only one* value. However, radio buttons may not always be the most appropriate (or even feasible) representation, given the device or modality in use; for instance, a radio button does not make sense when using a speech interface.

XForms separates *form* from *interaction* by capturing abstract notions such as *select from a set*. This enables the XForms author to create user interfaces that can be delivered to different target modalities and devices. XForms user interface control ⟨**select1**⟩ can be used instead of ⟨**input**⟩ to obtain the user's gender in the questionnaire example—see Figure 1.7 for the XML markup and Figure 1.8 for the resulting user interface.

As in the previous example, binding attributes *model* and *ref* specify the location where the value is to be stored. Attribute *appearance* is set to `full` to indicate that the client should create a full representation of this control; in the case of a visual presentation, this might be realized by using a group of radio buttons. Element ⟨**item**⟩ encodes each of the available choices. Subelement ⟨**label**⟩ contains the *display value*; subelement ⟨**value**⟩ encodes the value to be stored in the instance. The default value `m` is obtained from the model—see Figure 1.2. The author can style the interface further by using Cascading Style Sheets (CSS).

[15] http://www.w3.org/tr/xml-events
[16] http://www.w3.org/TR/CSS2/

```
<select1 xmlns="http://www.w3.org/2002/xforms"
  model="p1" ref="/person/gender" appearance="full">
  <label>Select gender</label>
  <help>...</help>
  <hint>...</hint>
  <item><label>Male</label>
  <value>m</value></item>
  <item><label>Female</label>
  <value>f</value></item>
</select1>
```

Figure 1.7. XForms user interface control for selecting a single value.

Figure 1.8. XForms user interface for selecting a single value.

1.3.4 XForms Submit

The final stage of the questionnaire user interaction is to have the user submit the information. Using HTML forms, this is achieved by creating a *submit* button within HTML element ⟨**form**⟩. Activating the corresponding user interface control results in *all* values created as part of the containing ⟨**form**⟩ being submitted to the URI specified via attribute *action*.

As mentioned earlier, a key feature of XForms is to separate the model from the interaction. XForms preserves this separation in its design of data submission. Submission details covering

- What to submit,
- Where to submit,
- How to submit

that are independent of the presentation are encapsulated by element ⟨**submission**⟩ within element ⟨**model**⟩. XForms user interface control ⟨**submit**⟩ when activated dispatches an appropriate **xforms-submit** event to the relevant ⟨**submission**⟩ element. Upon receiving this event, the XForms processor serializes the values stored in the instance before transmitting the result as specified by element ⟨**submission**⟩.

For the questionnaire example, we first extend the model shown in Figure 1.2 with an appropriate ⟨**submission**⟩ element—see Figure 1.9.

```
<submission xmlns="http://www.w3.org/2002/xforms"
  id="s0" method="post"
action="http://example.com/survey"/>
```

Figure 1.9. Element ⟨**submission**⟩ models what, where, and how to submit.

User interface control ⟨**submit**⟩ in Figure 1.10 uses attribute *submission* to connect the user interface to the model. We show the resulting user interface in Figure 1.11.

```
<submit xmlns="http://www.w3.org/2002/xforms"
  submission="s0"><label>Submit</label>
</submit>
```

Figure 1.10. XForms user interface control for submitting the questionnaire.

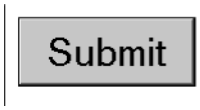

Figure 1.11. Visual representation of XForms submit control.

1.3.5 The Complete XForms Questionnaire

This section combines the model and user interface developed so far to create the complete XForms questionnaire. The resulting XForms application is contained in an XHTML document. The complete example uses XML namespaces so that markup elements defined by different XML languages such as XForms and XHTML are clearly identified—see Figure 1.12.

1.3.6 Deploying the XForms Questionnaire

The XForms questionnaire can be deployed in a variety of ways depending on the XForms processor being used. This section details a variety of deployment scenarios.

The questionnaire can be deployed on an XForms-aware Web server that provides the following:

Serve content The server produces a presentation that is appropriate for the connecting device.

Code generation The XForms server can generate client-side validation code to be embedded in the markup being served to the connecting

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:xf="http://www.w3.org/2002/xforms"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <head><title>XForms Questionnaire</title>
    <xf:model schema="questionnaire.xsd">
      <xf:instance xmlns="">
        <person>...</person>
      </xf:instance>
      <xf:submission action="..."
      method="post" id="s0"/>
  </xf:model></head>
  <body>...
    <xf:input ref="/person/address/street">
    ...</xf:input>...
    <xf:submit submission="s0">
      <xf:label>Submit questionnaire</xf:label>
  </xf:submit></body>
</html>
```

Figure 1.12. The complete XForms questionnaire.

client. This provides client-side validation and immediate user feedback, but without the cost of requiring the Web developer to hand-craft such validation scripts.

Data Validation Validate user data against the constraints given in the model.

State management Maintain application state by implementing the XForms processing model. As a result, the developer of the questionnaire need write no special software for maintaining values submitted by the user between client-server round-trips.

The XHTML document hosting the XForms questionnaire could be served to conforming XForms clients. An XForms client would implement the following:

Consume Consume the XForms-authored application to produce the client-side user interface.

Validate Check user input against the validation rules to provide live feedback.

Submit Submit a valid XML instance on completion.

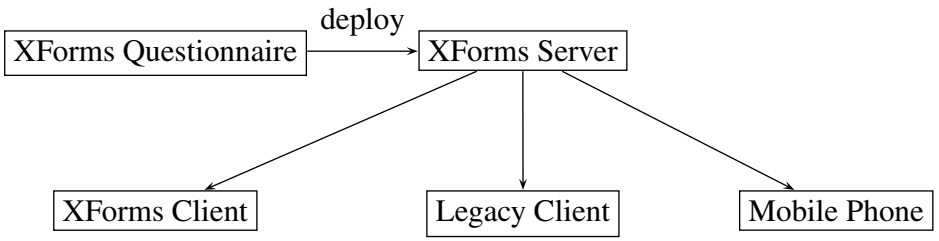

Figure 1.13. Deploying the XForms questionnaire.

Notice that in the deployment scenarios, the Web developer need only create the XForms questionnaire; contrast this with the application-specific components needed when deploying HTML forms—compare Figure 1.1 with the software components needed to deploy the XForms questionnaire shown in Figure 1.13.

1.4 Summary of XForms Benefits

We conclude this chapter with a summary of XForms features and highlight the consequent benefits over using traditional HTML forms for developing Web applications. A key differentiator when using XForms is the separation of purpose from presentation. The *purpose* of the questionnaire application is to collect information about the user. This is realized by creating a *presentation* that allows the user to provide the required information. Web applications typically render such a presentation as an *interactive document* that is continuously updated during user interaction. By separating the purpose from its presentation, XForms enables the *binding* of different interactions to a *single* model. In practice, this enables the application developer to deploy user interfaces that are appropriate to the target audience without having to create custom software components for processing the data collected via each distinct user interaction. Finally, by using structured XML to collect, validate, and communicate data, XForms processors can provide these functions to the Web developer as part of a standard XForms service.

1.4.1 XForms Features

Declarative Declarative authoring makes XForms applications easier to maintain.

Strong typing Submitted data is strongly typed and can be checked using off-the-shelf XML Schema tools. Strong typing also

	enables automatic client-side validation. As an example, a native XForms browser can use these types of constraints for validating user input; when serving the same XForms document to a legacy browser, these constraints can be used to generate client-side Javascript automatically.
Schema reuse	XForms enables the developer to reuse business rules encapsulated in XML Schemas. This obviates duplication and ensures that a change in the underlying business logic does not require reauthoring validation constraints at multiple layers of the Web application.
Schema augmentation	This enables the XForms author to go beyond the basic set of constraints available from the underlying business application. Providing such additional constraints as part of the XForms Model enhances the overall usability of the resulting Web application.
XML submission	This obviates the need for custom server-side logic to marshal the submitted data to the business application.
Internationalization	Using XML 1.0 for serializing data ensures that the submitted data is internationalization ready.
Accessibility	User interface controls encapsulate all relevant metadata such as labels, thereby enhancing accessibility of the application when using different modalities; as an example, a nonvisual client can speak relevant information when navigating through an XForms user interface.
Device independence	Abstract user interface controls lead to intent-based authoring of the user interface; this makes it possible to deliver the XForms application to different devices. Thus, Web pages authored using XForms can be deployed to a range of accessing devices including desktop browsers, PDAs, and cell phones with small displays.
Localization	Labels and help text can be referenced via URIs, thereby making it possible to localize XForms user interfaces.
Actions	XForms defines declarative XML event handlers such as ⟨**setfocus**⟩ and ⟨**setvalue**⟩ to obviate the most common use of scripting in Web applications. Consequently, most XForms applications can be statically analyzed; contrast this with the use of imperative scripts for event handlers.

1.5 XForms at a Glance

Table 1.1 shows the various XForms components at a glance; this also serves as a road map for the rest of this book. For each component, we enumerate its role in the XForms architecture, the underlying technology used by the component, and its concomitant benefits.

Table 1.1. XForms at a Glance

Component	Description
Model	• Encapsulates all data aspects of a form • Uses XML Schema to define constraints • Uses XPath to define model properties • Attaches model properties to instance nodes • Captures *what*, *how*, and *where* to submit
Properties	• Capture application constraints • Enable reactive user interfaces
UI Binding	• Connects user interface to the model using XPath
UI Controls	• Collect user input • *Bind* to underlying model • Encapsulate all relevant metadata • Access by design • Encourage device independence
UI	• Aggregates user interface controls • Encourages intent-based authoring • Creates dynamic user interaction
Events	• Bring user interface to life • Give access to eventing via XML Events • Attach dynamic behavior • Obviate common use of scripts via declarative actions