Chapter 10
A REST Framework for Dynamic Client Environments

Erik Albert and Sudarshan S. Chawathe

Abstract The REST Framework for Dynamic Client Environments (RFDE) is a method for building RESTful Web applications that fully exploit the diverse and rich feature-sets of modern client environments while retaining functionality in the absence of these features. For instance, we describe how an application may use a modern JavaScript library to enhance interactivity and end-user experience while also maintaining usability when the library is unavailable to the client (perhaps due to incompatible software). These methods form a framework that we have developed as part of our work on a Web application for presenting large volumes of scientific datasets to nonspecialists.

Introduction

The REST Framework for Dynamic Client Environments (RFDE) is a method for building RESTful Web applications (Fielding and Taylor 2002; Fielding 2000; Pautasso et al. 2008) that fully exploit the diverse and rich feature-sets of modern client environments while retaining functionality in the absence of these features. For instance, we describe how an application may use a modern JavaScript library to enhance interactivity and end-user experience while also maintaining usability when the library is unavailable to the client (perhaps due to incompatible software). These methods form a framework that we have developed as part of our work on a Web application for presenting large volumes of scientific datasets to nonspecialists.

The key problem addressed by the framework is: How do we build a robust and scalable Web application that, on one hand, uses to its advantage the numerous and increasingly capable clients and client-side libraries (e.g., Scriptaculous,

S.S. Chawathe (✉)
Department of Computer Science, University of Maine, 237 Neville Hall,
Orono, ME 04469-5752, USA
e-mail: chaw@cs.umaine.edu

E. Wilde and C. Pautasso (eds.), *REST: From Research to Practice*,
DOI 10.1007/978-1-4419-8303-9_10,

OpenLayers) but, on the other hand, retains all important functionality when one or more such client features are unavailable? More specifically, how do we combine the benefits of the REST approach to Web application design with those of active client-side features such as JavaScript and techniques such as Ajax (Asynchronous JavaScript and XML) (Garrett 2005)?

To reach a wide audience, a Web application must be able to support a wide range of client capabilities. Some mobile clients and clients on older computers often cannot use the latest Web technologies such as Adobe Flash, scalable vector graphics (SVG) (Jackson and Northway 2005), Java applets, or even advanced JavaScript. In order to develop an application that is accessible to the largest audience, developers often design for a simple set of capabilities and eschew the newer technologies. Alternatively, developers utilize new technologies and provide an alternative, reduced-functionality version for clients that cannot support the chosen technologies. And very often, unfortunately, Web applications will simply display a requirements message to the reduced-capability clients and provide no functionality at all. The RFDE framework provides a much more attractive option, as it permits the use of modern JavaScript and other features while retaining usability on clients without these features, and permits the Web programmer to support all such clients without explicitly writing code to handle the many cases. In an RFDE Web application, requests from a client returns a version of the application that is best matched to that client's supported, and active, features. The RFDE framework also endows an application with the ability to automatically upgrade itself using JavaScript and Dynamic HTML (DHTML) to a representation that can take advantage of more dynamic and advanced client features when they are available.

In the remainder of this chapter, we will describe the Climate Data Explorer, a climatological web application that inspired the RFDE framework, and identify the types of applications that can benefit from this approach. We will then introduce widgets and application templates, which are the building blocks of an RFDE application, and describe how they can be designed to target a large number of client environments with varying capabilities. Next, we will describe how we represent and maintain the state of a dynamic and event-driven application that is implemented using a RESTful, stateless application server. Finally, we will describe some of the work related to the RFDE framework, summarize the approach, and describe some possible future enhancements.

Motivating Case Study: A Climate Data Explorer

We describe a concrete application, the *Climate Data Explorer* (henceforth, *CDX*), that motivates our design criteria and also serves as a running example for illustrating the RFDE framework in this chapter. The primary goal of CDX is enabling nonspecialists to intuitively and interactively explore an integrated view of a large and diverse collection of datasets related to climate, with emphasis on the spatial and temporal attributes of this data.

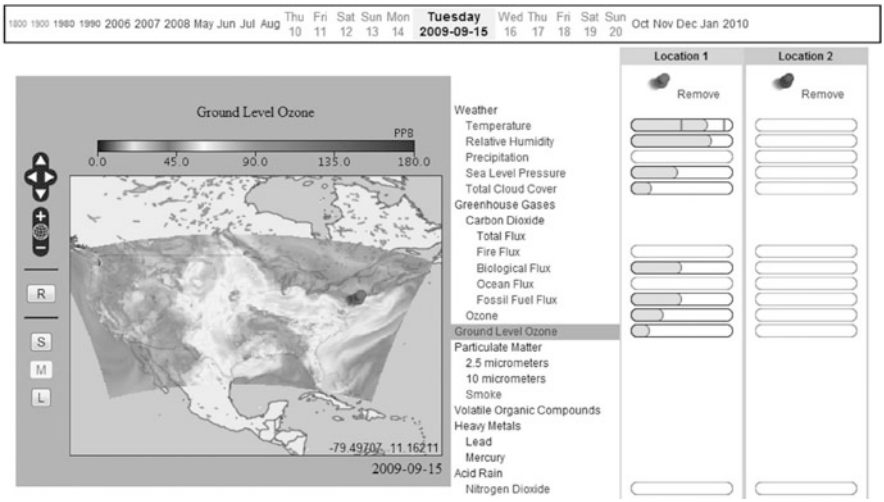

Fig. 10.1 A screen-shot of the Climate Data Explorer (CDX) application, which provides an integrated and interactive view of a large and diverse collection of datasets. CDX combines REST and modern dynamic client features using the RFDE framework

Various government and other organizations routinely publish data with direct relevance to climate. Examples of such organizations in the U.S. include the Environmental Protection Agency, the National Oceanic and Atmospheric Administration, and various state agencies such as the Maine Department of Environmental Protection. Data from these organizations differs in format and encoding, spatial and temporal coverage, measured or modeled attributes, and several other characteristics. As a result, it is difficult even for specialists to effectively use this data, even though most of it is publicly available on the Web. For example, a record of the global temperature and humidity fields for, say, December 31, 1984 is conceptually trivial to obtain based on datasets available on the Web. However, actually generating a suitable map-based representation of these fields is a difficult, laborious, and time-consuming (several hours) task for a specialist, and completely unworkable for a nonspecialist. In CDX, this representation may be generated in a matter of seconds using only a few mouse clicks and with no need for specialized knowledge.

Figure 10.1 depicts a screen-shot of the CDX Web application, illustrating its use for exploring climate data on a world map. For clients that support the required capability (mainly, modern JavaScript), the map uses common map features such as the ability to click and drag the map in order to pan around the globe, and balloon windows providing instantaneous feedback with more information on a clicked feature.

Some of the other components used by the CDX application include a *historical graph* and a *level indicator*. At the broadest level of client compatibility, these controls are both implemented using static images with hyperlinks to new windows

containing additional or explanatory information. When clients support more advanced browser features, these components are rendered using SVG and support animation, panning, mouse-over tooltips, and other advanced usability features. When a new value is displayed in a level indicator (see Fig. 10.5 on page 48 for an example indicator), the horizontal bar is animated as filling from left to right, and the color changes as values transition from healthy to unhealthy ranges. The historical graph allows the user to pan the visible area of a very long time line. This is accomplished by clicking and dragging the display when supported, or by clicking on panning control buttons when the browser does not support client-side rendering of the data.

While the advanced interface features are important for enhanced usability and for designing a compelling and attractive application, their use may be counterproductive if it were to lock out some, or many, users with low-powered computers, older browsers (or sometimes very new ones), or some mobile browsers from being able to view the same information. Having the ability to easily support clients with a varying array of capabilities is one of the most important and challenging requirements of this application, and one that motivates much of the work described in the rest of this chapter.

Target Applications

We outline some characteristics of the applications that are best suited to the RFDE framework, using the CDX application of "Motivating Case Study: A Climate Data Explorer" as a typical and concrete example. The target Web applications for RFDE are essentially those for which the three requirements of, briefly, *portability*, *interactivity*, and *scalability* are of primary importance. These requirements are elaborated below.

The portability requirement refers to the ability to run on numerous and diverse computing environments, including various combinations of hardware (desktop computers, smart phones, kiosks, and more), operating systems, and Web browsers. For our CDX example, this requirement is crucial in ensuring that the benefits of exploring climate data are available to as many people as possible, including those using older hardware and software, and those with special accessibility needs. A similar comment also applies to, say, a Web store that would like to attract as large a customer base as possible.

The interactivity requirement refers to the need to have a strong visual impact and maintain user interest, based on a dynamic interface design that includes familiar modern Web widgets and provides instant feedback to user actions. Examples of these widgets include ones for browsing tiled maps, updating lists and selections based on user actions, and displaying pop-up windows with hints and error messages. Also included are widgets designed primarily to provide a visually pleasing experience, such as those for providing smooth transitions between images, and fade-in and -out of displayed items. While it may be tempting to

write off the latter as frivolous decorations, their presence often makes a significant difference to the overall success and user acceptance of the application. For the CDX application, for instance, retaining user interest to encourage progressively more detailed exploration of the datasets and the underlying scientific and societal issues is greatly aided by such widgets.

The scalability requirement refers to the ability to easily increase the number of concurrent users supported by an application over several orders of magnitudes. For the CDX application, it is important that the implementation scale easily from hundreds to several tens of thousands of users as interest in the application grows and, further, that this scalability be achieved in a predictable manner by incorporating more hardware resources but without any significant qualitative change in the core design.

The portability and scalability requirements argue for the use of well documented and widely implemented Web standards. In particular, the REST approach is very natural and attractive design choice. The interactivity requirement argues for the use of modern Web widgets, tools, and JavaScript libraries that take advantage of recent developments in various parts of the client computing environment. Unfortunately, these two design choices are, without further work, largely incompatible. The core REST design and its typical implementations are based on the early interaction model between Web clients and servers, where most client actions generate a round-trip to the server, with concomitant implications for response times. Further, it is not immediately clear how one may apply the REST design to a Web application in which many actions, and state changes, occur through mechanisms such as Ajax (Asynchronous JavaScript and XML). This apparent incompatibility and its resolution are the core topics addressed by the RFDE framework, and this chapter.

While the RFDE approach itself is not dependent on any specific programming languages, scripting libraries, or client technologies, our implementation of the RFDE framework built to support the CDX application uses a number of specific languages that we will use in the examples throughout this chapter. Server-side code is written in the Java programming language, and client-side libraries and dynamically generated scripts are written in the JavaScript scripting language.

Widgets

The fundamental *resource* (in REST terminology) used by RFDE is the *widget*, which is a reusable user-interface element that allows one to view and manipulate application data. Common Web application widgets include form-entry fields, buttons, pull-down menus, checkboxes, radio buttons, and images. Widgets can also be built using other widgets, allowing for more complicated interface elements to be created quickly from the existing library, while also reducing proliferation of very similar code. By building a large collection of widgets, both general purpose and application specific, we can quickly create new Web applications that are portable, interactive, and scalable.

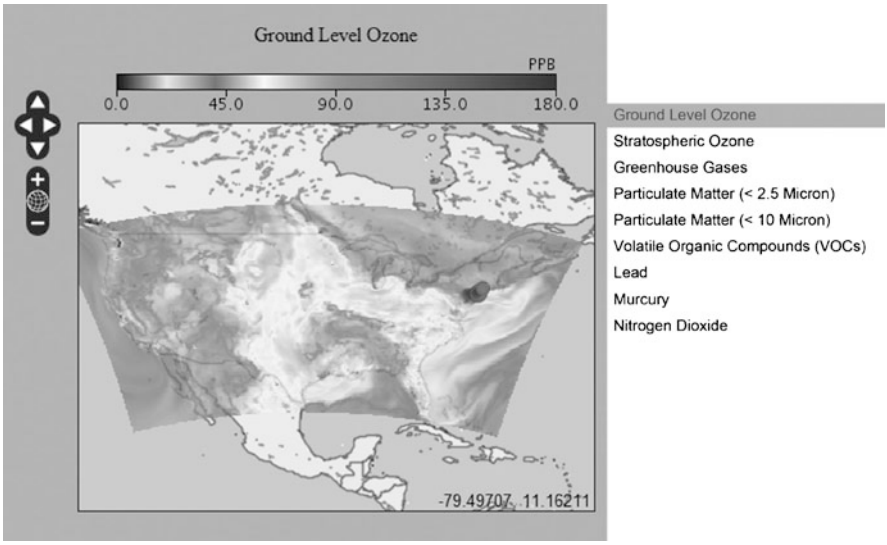

Fig. 10.2 A screen-shot depicting the use of the `mapview` application template in the CDX application

The CDX application (Motivating Case Study: A Climate Data Explorer) uses several application-specific widgets that allow the user to view and manipulate data from a multi-terabyte climate database. The screen-shot in Fig. 10.2 shows a simpler version of the climate-data browsing interface that consists of three primary widgets. The central widget is a *map widget* that supports the display of geographical distribution of the concentration of a climate parameter, such as the pollutant lead or stratospheric ozone. To the left of the map is a *navigation control widget*, consisting of several button widgets, that enables the user to pan and zoom the map. The third widget, displayed as the list of climate parameters to the right of the map, is a *selectable-list widget* that permits the selection of a parameter to display on the map.

A widget is implemented using one or more representations (e.g., a static image, DHTML, Flash, etc.) that correspond to the *capability set* (Client Capability Tiers) of the client. In the CDX application, the map widget is represented using the OpenLayers JavaScript library when the client supports it; otherwise, it is represented using a static image rendered on the server-side. Likewise, the map navigation buttons and the climate variable list items are represented using HTML anchor tags when JavaScript is not available, and as JavaScript supported clickable markup when it is.

Widgets can perform tasks through invokable methods and registered event handlers. The map widget is implemented using several methods such as `moveNorth`, `moveSouth`, `moveEast`, and `moveWest` which pan the map in the given direction; `center`, which centers the map on a given latitude and longitude; and methods to control the zoom level such as `zoomIn`, `zoomOut`, and `zoomWorld`. Each of these methods have dual implementations in the CDX library: one in Java

that implements the method on the server, and one in JavaScript that can be invoked directly on the client when the widget is represented using the OpenLayers library (in general, each additional tier would require another implementation of the widget class). In addition to its methods, a widget can also identify a set of events that it generates. An event typically corresponds to a user action, such as changing the zoom level of the map widget (an `onZoom` event), and an application can specify what actions are performed when a given event occurs. Further details on methods and event handling, including examples, appear in "Event Handling".

The RFDE server publishes a common widget interface that can be used to obtain the value of a specific widget (that has a derived value) given a set of parameters. The value of the widget is represented using a language that is appropriate for programmatic use, such as such as XML or JavaScript Object Notation (JSON) (Crockford 2006). Later, we will discuss how this interface is used to implement much of the application dynamically, on the client side, when this feature is supported by the client environment.

Application Templates

An RFDE Web application is built using *application templates*, each of which is a composite resource (in REST terminology) that consists of collections of widgets that implement a common application usage pattern. In addition to its widgets, an application template also encodes the logic that controls the behavior of the widgets in the context of the template. A Web application contains only one instance of each application template, although a template may be replicated on multiple servers for load sharing.

An important property of RFDE templates, and one required by REST, is that the server side of an application does not save the state of a template for any of its clients. Instead, the client sends a request to the template that includes an encoding of its state, and the template returns a representation of the application at that state. For example, the CDX application uses a `mapview` template as suggested by Fig. 10.2. This template is initialized to a specific location, zoom level, and climate parameter; however, by manipulating the state value in the URI of the application, the client can change what information is displayed on the map.

The definition of the example `mapview` template is given in Listing 10.1. In lines 1–2, the template is created and assigned a CSS style sheet. The map widget and its corresponding navigation widget are created in lines 4–6. The navigation widget combines all of the map navigation buttons and automatically adds event handlers that invoke the corresponding methods of the map widget. Next, the list of parameters is created and populated with all of the possible variables that can be displayed on the map. In lines 14–15, an action is added to the list widget's `onChange` event handler that causes the `parameter` state variable of the map widget to be changed when the user selects a new value from the list. Finally, the widgets are added to the template (using a horizontal panel) and the template is initialized. This initialization routine involves the generation and caching of

```
template = new AppTemplate("CDX mapview Example", "mapview");
template.addStyleSheet("cdx");

MapWidget map =
   new MapWidget(40.7166, -74.0067, 1, 400, 300, "o3");
MapNavigator nav = new MapNavigator(map);

List plist = new List();
plist.addItem("Ground Level Ozone", "o3");
plist.addItem("Stratospheric Ozone", "o3strat");
// ... additional values omitted ...
plist.addItem("Nitrogen Dioxide", "no2");

plist.onChange().addAction(
   new StateChangeAction(
      map, "parameter", plist, "selectedValue"));

template.addWidget(new HorizontalPanel(nav, map, plist));
template.init();
```

Listing 10.1 The definition of the `mapview` application template

static markup that will be used in every document generated by the template; the creation of an explicit representation of the default state of the template, based on the parameters specified in the template definition (Representation of Application State); and the use of a widget dependency graph to create a valid ordering for instantiation in client-side code.

When a new template request is made, the server program that hosts the application is responsible for translating the encoded application state into a state object, "executing" the template, and returning the resulting document. Executing a template requires generating the markup language for each of the widgets based on the current state of the application, as well as creating initialization parameters for client-side versions of the widget implementation classes. The resulting document contains static references to external resources used by the document (such as style sheets), references to the RFDE libraries that implement the client-side versions of the widget classes used by the template, the generated upgrade parameters and event handlers, and finally, the markup the implements the page and its widgets (example markup for an *image push button* widget is given on page 247).

Client Capability Tiers

The RFDE framework supports the development and deployment of Web applications that support, concurrently and interchangeably, client environments with diverse and changing capabilities. For instance, one user may run the application on

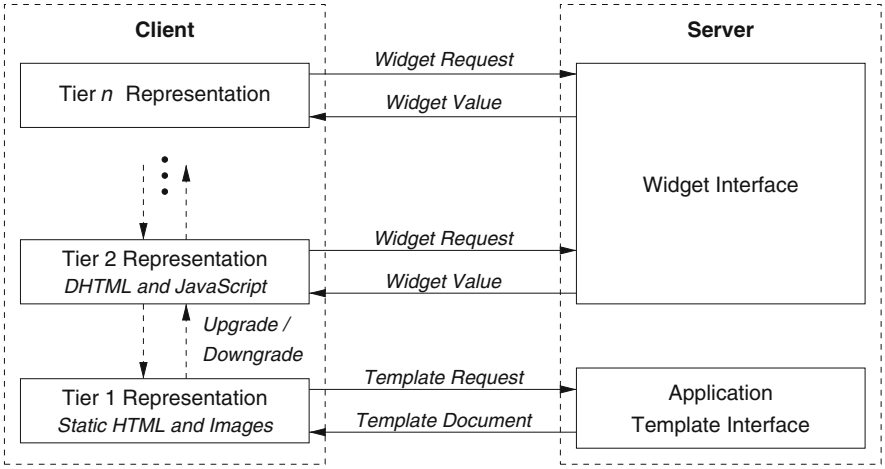

Fig. 10.3 Client capability tiers in RFDE

a desktop computer running Windows XP and Internet Explorer 8 while others (or the same user) access it using, variously, a smart phone running Symbian and Opera, a kiosk running GNU/Linux and a customized version of Firefox, or a computer with software that is several years behind the current versions.

It would be foolhardy to attempt to explicitly address every possible combination of the components of a client environment: hardware, operating system, Web browser, and so on. Instead, RFDE models the features and abilities of the computing environment on the client side using *client capability tiers*. These tiers classify client environments by specifying the properties required for tier membership. RFDE includes a default definition of these tiers, but application programmers may easily modify both the number of tiers and the individual tier definitions, and such modification is expected and encouraged. The lowest tier (Tier 1) is designed to be as inclusive as possible, and thus specifies the bare minimum for what is needed for the application to function. A guideline for Tier 1 is to include only those requirements without which there is no reasonable way to accomplish the key tasks of the application. As suggested by Fig. 10.3, each higher tier adds increasingly demanding requirements for the client environment. When a client interacts with an RFDE application, the framework automatically uses the highest (most capable) tier that the client's environment supports. This default behavior may be changed, and the tier may be explicitly set to a desired one by using *tier selection widgets* which are typically used during testing.

Tier 1 clients that support only the minimum requirements are able to use a fully-functional version of the Web application, although some of the visual and usability enhancements afforded by more capable environments may be missing. As a simple example, a client without scripting support may not provide immediate feedback on potentially incorrect data. However, not only are the functions implemented by

the form (perhaps a purchase) fully supported, but also the feedback on incorrect data is provided, albeit with a slightly longer response time due to a server round-trip and page refresh. If the client environment supports additional capabilities, the application widgets will be automatically *upgraded* to versions that use these capabilities to improve the speed, responsiveness, usability, or appearance of the application. A special JavaScript class in the client-side RFDE library, called the *widget manager*, is responsible for the instantiation and automatic upgrade of all of the widgets in an application document based on the identified tier level of the client environment.

For the CDX application, consider the `mapview` template of Fig. 10.2. In Tier 1, the user is able to pan and zoom the map but must do so using the navigation buttons on the left. A more direct manipulation of the map by clicking and dragging on the map itself is not supported because the client environment capabilities (JavaScript, etc.) that are needed to implement such manipulation are not part of Tier 1. Map manipulations, and most other actions, in this tier also require full page refreshes and a new rendering of the visible area of the map, with the associated, typically noticeable, delays. In Tier 2, the map is more interactive. In addition to the direct manipulation using dragging, it also permits zooming in and out using scroll wheels and similar input modes. Further, map features are associated with pop-up balloon windows with hints or other brief messages. Map tiles and other images are loaded asynchronously and partial updates of the displayed Web page are accomplished by manipulating the DOM tree; these enhancements avoid full page refreshes in most cases and so greatly improve responsiveness.

Figure 10.3 depicts this tiered approach of a RFDE Web application. At the lowest level of the figure, a client communicates with the application server to request an updated view of the application. At this level, the document returned contains the entire application template, including all the widgets in the template. The state of the application is explicitly encoded in the URI that the client sends, and the application view that is returned is represented using a markup language such as HTML. The hyperlinks in the document contain URIs that encode new states for the application, so that when the user clicks on a link, the net effect is that the state of the application is updated and the new view of the data is returned.

Embedded in a Tier 1 client document is a small script that checks client capabilities when the document is loaded. If the client does not support the scripting code, it will simply be ignored and the client will remain at this tier for the duration of the exchange. If the capability check determines that the browser supports a higher tier, the client-side widget manager will automatically upgrade all the widgets on the page to their higher-tier representations. For example, Tier 1 may represent the application using HTML and static images, Tier 2 may add JavaScript and client-rendered images, and a third tier may use Adobe Flash or advanced SVG graphics to render the application. If the client supports JavaScript, but not Flash or advanced SVG, the client code will upgrade the widgets to their Tier 2 versions and future interactions with the server will take place at the RFDE widget interface.

In addition to the server-side Tier 1 widget library, an RFDE server supports an arbitrary number of additional levels of higher-capability, client-side widget

libraries. At these higher tiers, the client requests the value of individual widgets, instead of entire application templates, through the common widget interface. This design allows the client to use asynchronous transactions to replace the value for individual widgets in a template, improving the application's responsiveness. The upper-tier widget libraries use representations for widget values that are more appropriate than HTML, such as JSON, allowing for any type of client technology (such as HTML, DHTML, SVG, Flash, etc.) to be used to render the widget.

The initial framework developed for the CDX application consists of two tiers of client capability. However, additional tiers are likely to be added based on the expected mix of client categories and an important aspect of RFDE is that such additions can be made easily, without affecting existing code and application functionality. In the lowest tier, the widgets are represented using HTML 3.2, static images, image maps, and hyperlinks. The application also uses cascading style sheets to control the look and feel of the page. These style sheets are ignored by browsers that do not support them. Images, such as the tiles in the visible area of the map, are rendered by the server and sent to the client in a widely supported format such as JPEG, GIF, or PNG. In this level, each user interaction with the application (informally, each click) requires a complete page refresh. For example, a single-button widget, such as the zoom-in button in the map navigation control widget, is represented using the following HTML:

```
<a id="ImagePushButton5"
   class="ImagePushButton ImagePushButton-t1"
   href="/cdx/1.0/mapview?state=&e5=zoomIn">
  <img src="/images/map/zoom-in.jpg"
       alt="Zoom In"
       border="0" />
</a>
```

The widget is represented as a simple hyperlinked image in this tier. When the user clicks on the image, indicating a zoom-in event, the state of the application is updated (in a REST-compatible manner) following a round-trip interaction with the application server and subsequent page refresh at the client. Event handling is discussed further in "Event Handling".

If the client supports JavaScript, DHTML, Ajax, and SVG, it is automatically promoted to second tier functionality when the application is loaded. In this tier, each upgradeable widget is replaced with its JavaScript and DHTML implementation. After such an upgrade, client interactions no longer require a full page refresh. Widgets change their displayed forms by using client technologies, such as JavaScript and DHTML. For example, the upgrade dynamically replaces the earlier static-HTML representation of the zoom-in button with its second tier equivalent:

```
<img id="ImagePushButton5"
     alt="Zoom In"
     src="/images/map/zoom-in.jpg"
     class="ImagePushButton ImagePushButton-t2">
```

Unlike the earlier representation, there is no longer a static hyperlink and the widget identifier now appears in the image tag. The `ImagePushButton-t1` CSS class has been replaced with the `ImagePushButton-t2` class, allowing

for independent styling of the two tiers. When upgraded, a JavaScript-class implementation of the widget is instantiated and the class registers any required event handlers (such as `onClick` for this button) with the browser. If a widget has a derived value, a value that is determined by its parameters that is also dependent on other information, such as a database, the widget will update its value using an asynchronous callback to the RFDE widget interface. These changes allow a control to remain dynamic without requiring the full-page refresh caused by the hyperlink-based implementation. The two representations of the zoom-in button are visually and functionally nearly identical; however, in Tier 1 pressing the zoom in button requires a complete page refresh to perform the operation, while in Tier 2, the event is handled completely in the browser without requiring a page refresh.

Representation of Application State

Following REST conventions, the current state of an RFDE Web application is explicitly encoded in the application's URIs. The advantages of this design are similar to those of other REST-based ones: By using a completely stateless protocol, multiple servers can implement the application, client requests can be handled by any available server, and the application can be scaled by increasing the number of available servers in a load-sharing environment. This design also allows the use of caching strategies to optimize common requests, such as the most recent map images for frequently queried areas of the United States. Finally, by explicitly representing the state of the application in the URI, users of the application can bookmark and revisit a particular view of the application, or share their experiences with others, in a robust and standard manner.

Our implementation of RFDE identifies state variables using a positional scheme in order to reduce the total size of the state encoding (compared to an alternative named-variable scheme, as used for HTML query strings). To further reduce the size of the state string, values that have not changed from their template-specific default values are omitted from the encoding. The state of each widget (the collection of its parameters) is represented as a string composed of the widget identifier followed by a colon delimited list of the state values. The state of the entire application template consists of an asterisk-delimited list of widget states. In order to support long-term bookmark compatibility as an application and its widgets evolve over time, each application template URI includes the application version. When an application receives a request with an old version number, it should attempt to construct an equivalent URI compatible with the latest version and redirect the client (using an HTTP 301 Moved Permanent redirection).

Figure 10.4 shows an example hierarchical state representation for the `mapview` application template from Fig. 10.2. This application template consists of three widgets; however, the map navigation widget does not have any internal state and is omitted from the state representation. The map widget has the `Map1` identifier and the selectable list of climate parameters is given the identifier `List1`. The default

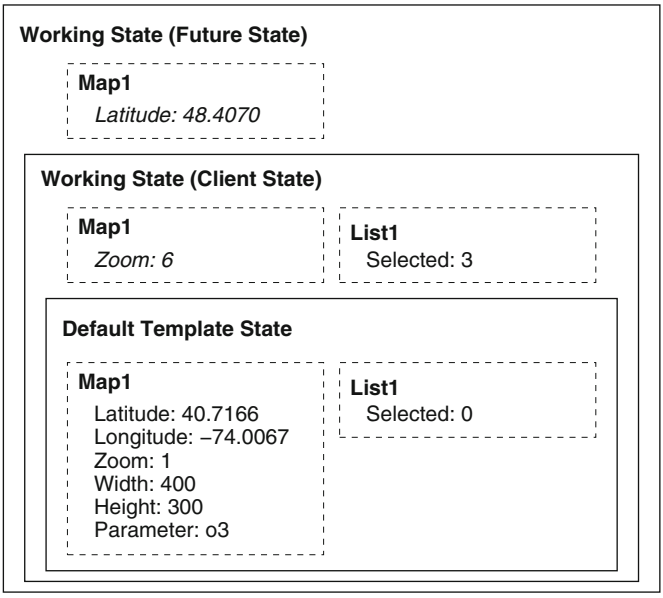

Fig. 10.4 An example of the representation of an application template's state. The innermost state represents the application's default state while the outer states are specific to a client request

application state is shown in the innermost layer of the diagram, which contains values for all of the properties for the two widgets. The order of the properties in the diagram corresponds to the order of the values in the state value string, so latitude is the first, longitude is the second, and so on. When the client does not specify a value for the application state, the default state is used (1.0 identifies the version of the web application):

```
/cdx/1.0/mapview
```

A client may also use the following complete state representation, even when the application is at its default state.

```
/cdx/1.0/mapview?state=Map1:40.7166:-74.0067:1:400:300:o3*List1:0
```

A *working state* is a representation of state that keeps track of changes from another state (typically the default state, but working states may also be nested). When the client sends a request for an application template, the server builds the working state for the request which is then used to generate the document that is returned. If a state variable is not included in a working state, the default value for the variable is used. The middle layer in Fig. 10.4 represents the current client state, in which the values of two state variables have been changed from their defaults. This state is created in response to a client request with the following application URI:

```
/cdx/1.0/mapview?state=Map1:::6*List1:3
```

The only changed property of the map widget is the zoom level, which is the third property of the widget. The colons corresponding to the first two properties of the map widget must be included in the encoding to ensure proper positional representation; however, additional colons at the end of an encoding may be dropped. In this example, the colons corresponding to the last three properties (width, height, and parameter) are dropped because these properties retain their default values.

The outermost working state in Fig. 10.4 represents a potential future state that may be used to generate proper URIs for inclusion in the current application hypertext. In this example, this future state represents the state of the application if the user were to pan the map to the north, and this state could be encoded in the hyperlink URI for the corresponding map control button:

```
/cdx/1.0/mapview?state=Map1:48.4070::6*List1:3
```

The default state representation for an application template is a constant value that is only initialized once, when the template is created, and then shared among all client requests. When the server receives a new request, it only has to instantiate a more light-weight working state to represent the changes from the default state. When a working state is created, the RFDE application server automatically performs type and sanity checking of the state values based on constraints that can be specified when a widget registers a new state variable in the default state.

While operating at the lowest tier level, the client manages the application state implicitly, using state-encoded URIs in hyperlinks and HTML forms. When a client is upgraded to a higher tier level; however, the client becomes more actively responsible for keeping track of the state of the application. Many of the actions that are performed by an event handler are simple to complete on the client, such as changing the CSS classes used by the selectable-list widget in order to highlight a newly selected value, and requiring a round-trip exchange with the server in order to perform this task would be an unnecessary cause of latency that would affect the perceived responsiveness of the application.

The widget manager is responsible for keeping track of the application state on the client. While the server needs to explicitly model the default state and any changes to the default state made by each of the clients, the client only needs to keep track of the current state of the application. When the application state is changed, the widget manager requests any updated widget values from the server (if necessary) and updates the current application URI to allow the user to bookmark any particular view of the application.

Event Handling

We use the term *events* to refer to the interactions of a Web application user with the user interface. Examples of events include clicking on buttons, selecting items in drop-down menus, and panning a map. Each application widget recognizes the

events that relate to it. For example, a map may have an `onMove` event which corresponds to a user request for panning to a new location and an `onZoom` event which corresponds to a user request for changing the zoom level. Each event may be associated with a set of actions that are performed whenever the event occurs, and these actions may in turn affect other widgets in the application. A simple example in the CDX `mapview` template is that changing the selected climate parameter in the list widget also changes the parameter that is displayed in the map.

After creating the widgets in an application template, the programmer specifies the application behavior by associating actions with widget events. Actions can affect an application in various ways, such as changing a state variable, invoking a widget's method, or even firing another event, which may in turn trigger additional actions, recursively. For example, the zoom-out button in the navigation widget of the CDX application is assigned an action that invokes the map's `zoomOut` method when the user clicks on the button:

```
zoomOutButton.onClick().addAction(
   new InvokeMethodAction(map, "zoomOut")
);
```

The manner in which this event handler is executed depends on the client capability tier (Client Capability Tiers) that is active at the time of the event.

In Tier 1, an event is initiated by including an event identifier in a request query string. Events have an optional argument which is used to specify event parameters. For example, an event caused by the user selecting a different climate variable would be parameterized with the index of the new selection. When there is no actual parameter, the value 1 is used to indicate that the event was activated. In the CDX application, the zoom-out button's `onClick` event is assigned the identifier `e3` and the hyperlink has the following URI:

```
/cdx/1.0/mapview?state=List1:2&e3=1
```

This URI indicates that the only change from the default state of the `mapview` template is that the third climate parameter is selected in `List1` (using zero-based indexing) and that the zoom-out button has been pushed.

When the server receives a request that includes an event identifier, it immediately triggers the associated event handler. In this case, the only associated action is to execute the `zoomOut` method of the map widget, as specified by the following server-side code fragment:

```
public void zoomOut(WorkingState state, String param) {
   int zoom = state.getIntegerValue(getId(), "zoom");

   // Update the zoom state variable
   state.setStateVariable(getId(), "zoom",
          (int) Math.max(0, zoom - 1));

   onZoom().fireEvent(state, "out");
}
```

Since templates are stateless, the current application state is passed as the argument `state` to the `zoomOut` method. The method determines the current value of the

zoom, updates it, and modifies the working state. Finally, the method triggers the map widget's `onZoom` event which, by similar mechanisms, triggers the appropriate event-handling method for the map widget, which will cause any actions identified as side-effects to changing the zoom level to be also be executed (there are none in the `mapview` example template).

Once the server has completed executing all of the event handlers, the client is immediately redirected, using an HTTP 303 See Other redirect, to a URI that fully encodes the new application state based on the updated value of the working state. Thus, the non-transient URIs at the client never include pending events. As a result of the zoom-out widget's `onClick` event, the client is redirected to the following URI with a modified zoom value:

```
/cdx/1.0/mapview?state=Map1:::0&List1:1
```

At higher tier levels, more of the event handling is managed on the client side in order to increase the responsiveness of the application and to reduce the number of complete page refreshes. Tier 2 event handling in RFDE is performed by the JavaScript implementations of the widgets. When widgets are initialized, they are given JavaScript versions of event handlers. The zoom-out button is instantiated with the following event handler, which is automatically generated from the Java version of the event handler (the `$I` function returns the instance of the identified widget):

```
onClick: function(param) {
   $I('Map1').zoomOut(param);
}
```

The client-side JavaScript version of the map widget has the following implementation of the `zoomOut` method:

```
zoomOut: function(param) {

   // Update the zoom state variable
   this.state.zoom = max(0, this.state.zoom - 1);
   this.state.update()

   // Zoom out the JavaScript map
   this.map.setZoom(this.state.zoom);

   this.onZoom("out");
}
```

This client-side implementation of `zoomOut` is nearly identical to the earlier server-side implementation, but there are two notable differences: First, the widget manages its own state directly rather than requiring the state as an additional argument. Second, and more important, the client-side version of the method actually causes the map to zoom out as a direct side-effect. Recall that the server-side version only modifies the representation of the application state.

When a widget needs to update its value due to an event that is handled on the client side, it requests the value from the RFDE server's widget interface, based on its updated parameters. For example, one of the widgets in the CDX application is a *level indicator*, a widget that graphically presents the value and health implications of a specified climate parameter, such as a pollutant, at a location and time which

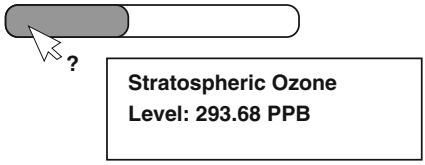

Fig. 10.5 An example of the level indicator widget which, when upgraded, uses asynchronous calls to the server to modify its value as a user changes the selected location or date being displayed

are specified using other widgets. Figure 10.5 depicts an example level indicator for stratospheric ozone (the ozone layer). The horizontal bar in the figure is filled to indicate the comparative value of the underlying parameter. The bar's color is mapped to health standards, with green denoting a healthy level, for instance. In our Tier 1 implementation of the level indicator widget, it is rendered as a static image generated on the server side. When the client is upgraded to Tier 2, the indicator is rendered on the client side and gains niceties such as animated filling of the bar and a textual description of the level that appears as a balloon activated by a pointer-hovering event.

When the user changes the selected date or location, the level indicator must be updated to display the parameter value at the date or location. The widget sends an asynchronous request for the new value to the widget library. This REST-based interface can supply the value of a widget based on the widget's parameters, in a representation that is more appropriate for programmatic manipulation. For example, the following URI requests an updated value for a level indicator widget:

```
/w/1.0/LevelIndicator?state=o3strat:48.41:-74.01:2010-09-23
```

The server responds with a representation of that value, in this case encoded using JSON:

```
{
  "widget"     : "LevelIndicator",
  "version"    : 1.0,
  "state"      : { "parameter" : "o3strat" ,
                   "latitude"  : 48.41,
                   "longitude" : -74.01,
                   "date"      : "2010-09-23" },
  "uri"        : "/w/1.0/LevelIndicator?state=
                    o3strat:48.41:-74.01:2010-09-23",
  "param_name" : "Stratospheric Ozone",
  "param_alt"  : "The Ozone Layer",
  "units"      : "PPB",
  "level"      : 293.68,
  "US_limit"   : undefined,
  "EU_limit"   : undefined,
  "health_idx" : 0
}
```

On receiving this response from the server, the client-side widget code changes its internally stored value and re-animates the filling of the display bar.

A Sample User Session

We now illustrate some of the interactions outlined in earlier sections in the context of a simple session of user interactions with the CDX application of "Motivating Case Study: A Climate Data Explorer." First, the client loads the CDX portal, which is a directory for a number of CDX application templates, by sending a request to the server. Next, the user selects a link to the `mapview` application template. Finally, the user performs two events on this application page: (1) changing the selected parameter in the list to lead and (2) activating the zoom out control for the map.

Figure 10.6 illustrates this sequence of events for both Tier 1 and Tier 2 compatible clients. For simplicity, this figure focuses entirely on the interactions that represent the main application logic flow; not shown are the additional requests for document resources, such as embedded images, made by the client. The Tier 1 interactions are shown in Fig. 10.6 (a). The first two user requests (for the CDX portal and the `mapview` template) are made as standard HTTP GET requests; each results in the server generating and returning a complete Tier 1 document. The event

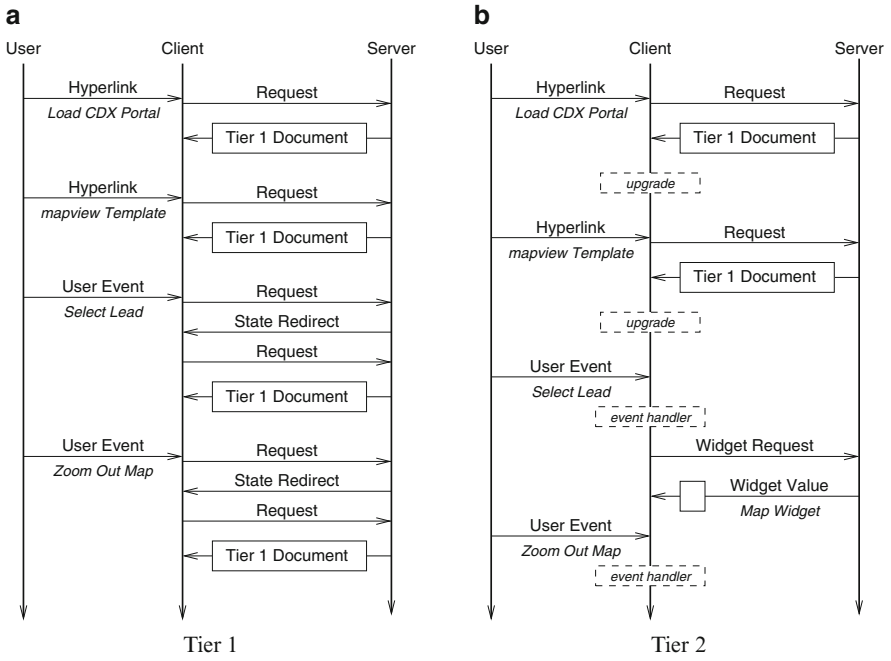

Fig. 10.6 An simple CDX session. At Tier 1, each request and event requires a full-page refresh. Event handlers on the server compute the modified state representation and redirect the client to the new URI. At Tier 2, event handling is performed on the client and only the values of individual widgets are requested from the server and updated on the client side. Some events may be handled completely on the client and do not require a request to the server (such as the `onZoom` event corresponding to the user zooming out the map in this example)

requests (selecting lead and zooming out) require server-side event handling. For both of these requests, the server receives the request from the client which includes the event identifier, executes the event handler which computes the new application state, and then redirects the client to the new application URI, which encodes the new state.

The corresponding sequence of events for a Tier 2 client is shown in Fig. 10.6 (b). For complete template requests, the Tier 2 interactions are handled exactly as in Tier 1; however, when the client loads a Tier 1 document, the embedded script upgrades the document to its Tier 2 equivalent. In Tier 2, event handling is performed on the client side, rather than requiring a complete page refresh, reducing latency and allowing the application a much greater level of responsiveness. When the user changes the parameter to lead, the event handler for the selectable list widget's `onChange` event (in client-side code) signals the widget manager to asynchronously request a new value for the map widget. The response from the widget interface, which is significantly smaller than a complete Tier 1 document, includes details that the map widget requires to properly render the new map and to request a new set of map tiles. Some events, such as when the user zooms the map out, can be handled completely on the client, and do not even require a request for an updated widget value. The underlying map tiles are requested from the server as usual, although they may also be cached on the client side by the usual browser mechanisms.

Related Work and Discussion

The RFDE framework described in this chapter is an advanced and REST-based *progressive enhancement* strategy (Wells and Draganova 2007; Parker et al. 2010) for Web development. This strategy uses, at the core, basic markup that is supported by the capabilities of the most primitive expected client. Advanced features and layout implemented through external links to JavaScript and Cascading Style Sheets. Progressive enhancement is based on the separation of document structure from the layout styling, and all presentation tasks are handled by style sheets. In contrast to strategies based on *graceful degradation* (Randell et al. 1978), which degrade to a more basic implementation when the client does not support the full implementation, the progressive enhancement strategy ensures that any client always obtains the full content and at least a minimal set of functionality and styling. This strategy is especially important for ease of indexing by search engines, and for users of assistive technologies which typically require that the basic content is always available and not hindered by dynamic content delivery.

With the development of mobile Internet devices such as smart phones, eReaders, and tablets, there has been a large amount of work on multi-device user interfaces (e.g., Grundy and Yang 2003; de Oliveira and da Rocha 2005) that allow an application to use the native features of the host device. Many of these approaches have adopted a device-independent user-interface specification language such as

UIML (Edwards et al. 2000; Ali and Abrams 2001), and use an application-independent user interface library to realize the application on the host device.

Nokia has described a Remote MVC (model-view-controller) application controller (Stirbu 2010) that models user interfaces as REST resources and that uses an event-based system to keep the client and service synchronized. When this framework is initialized, applications can discover and acquire platform-specific representations of the user interface elements that use a device's native functionality and match the look-and-feel of the device.

A significant advantage of the RFDE framework of this chapter is that it allows the development of portable, interactive, and scalable applications. Rather than attempt to support the myriad of client devices natively, RFDE models common client features in tiers of capability. These tiers allow the development of specific representations for interface elements based on a small number of tiers that framework implementations support, rather than developing a representation for each possible device.

While our description in this chapter uses several specific technologies such as JavaScript and Java, the RFDE design does not depend in any significant manner on these technologies, and others may be substituted where appropriate. The framework also does not impose an architectural style (such as MVC) on an application, and the programmer may choose the one best suited to a task.

Application programmers who use RFDE can develop an application by adding widgets to an application template and then specifying actions that should occur as a result of their related events. This development approach is similar to application development for desktop applications and Web development frameworks that are modeled on desktop application development, such as the Google Web Toolkit (McFall and Cusack 2009). In this approach, the application programmer may treat widgets as abstract entities without immediate concern to their implementation on various client environments.

A major goal of this approach is to develop user interfaces that can become more responsive and intuitive according to the capabilities of the client environment, while affording all of the functionality of the application, even at the lowest level of capabilities. Here, the functionality of the application refers to what can be done with the application and not necessarily how it is performed. In the CDX application, for example, the user must use the buttons in order to navigate the map at Tier 1, but at Tier 2, the map becomes responsive to mouse control. Tier 1 users can still fully navigate the map, even though they need to do so via a slightly more primitive interface and changes require a full request/response cycle with the server. Tier 2 map users can still use the familiar button interface to navigate the map and both visual representations (assuming that basic CSS is supported by the Tier 1 clients) of the maps are identical at both tiers.

While the RFDE approach removes, or at least significantly reduces, the need for a Web application developer to produce device or platform specific interface elements, it does not preclude the development of native interface implementations. In fact, the REST uniform interface constraint supports and facilitates the development of these native interfaces. Devices can use their own implementations of the

interface widgets and populate any derived values via requests through the widget interface, in the same way that the CDX Tier 2 clients do. Optimized clients for the Climate Data Explorer are currently under development for several popular mobile device platforms.

The current version of the RFDE server implementation does not include a uniform and generic method for describing the logic of an application template (e.g., the widgets, layout, constraints, and event handlers that comprise the template); however, the development of such a language is an important next step in our research. This language would allow native implementations to utilize the same application templates that the Web applications use and reduce the development costs associated with adapting new application templates and updating existing templates as they are improved.

One potential drawback to the RFDE approach is that each widget needs to be implemented multiple times. For example, in order to create a widget for CDX, a Java class that implements the Tier 1 widget needs to be written, a JavaScript analog needs to be written for the Tier 2 client-side implementation, and (for some of the widgets) the RFDE widget interface needs to be updated to generate a JSON representation for the value of the widget. On the other hand, once the underlying widgets have been implemented, an application can be developed that automatically supports the various levels of capabilities of its clients – the alternative would be to develop multiple versions of the same application. One solution to the problem of handling the dual implementation of the widget library, and one that we plan on investigating for the next version of the framework, is the development of a language or library that can be used to write widgets that will automatically compile both the client-side JavaScript libraries as well as the server-side implementation from a single source.

Acknowledgements This work was supported in part by the U.S. National Science Foundation grant EAR-1027960 and the University of Maine.

References

Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. “big” web services: making the right architectural decision. In *WWW ’08: Proceeding of the 17th International Conference on World Wide Web*, pages 805–814, ACM, New York, NY, USA, 2008.

Dean Jackson and Craig Northway. Scalable vector graphics (SVG) full 1.2 specification. WD not longer in development, W3C, April 2005. http://www.w3.org/TR/2005/WD-SVG12-20050413/.

Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), 2006.

Jesse James Garrett. Ajax: A new approach to web applications. 2005.

John Grundy and Biao Yang. An environment for developing adaptive, multi-device user interfaces. In *AUIC ’03: Proceedings of the Fourth Australasian user Interface Conference on User Interfaces 2003*, pages 47–56, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2003.

John Wells and Chrisina Draganova. Progressive enhancement in the real world. In *HT '07: Proceedings of the Eighteenth Conference on Hypertext and Hypermedia*, pages 55–56, ACM, New York, NY, USA, 2007.

Mir Farooq Ali and Marc Abrams. Simplifying construction of multi-platform user interfaces using UIML. In *European Conference UIML*, 2001.

B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surveys (CSUR)*, 10(2): 123–165, 1978.

Rodrigo de Oliveira and Heloísa Vieira da Rocha. Towards an approach for multi-device interface design. In *WebMedia '05: Proceedings of the 11th Brazilian Symposium on Multimedia and the Web*, pages 1–3, ACM, New York, NY, USA, 2005.

Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

Ryan McFall and Charles Cusack. Developing interactive web applications with the google web toolkit. *J. Comput. Small Coll.*, 25(1): 30–31, 2009.

Stephen Edwards, Manuel A. Prez-quiones, Mary Beth Rosson, Robert C. Williges, Constantinos Phanouriou, and Constantinos Phanouriou. UIML: A device-independent user interface markup language. Technical report, 2000.

Todd Parker, Scott Jehl, Maggie Costello Wachs, and Patty Toland. *Designing with Progressive Enhancement: Building the Web that Works for Everyone*. New Riders Publishing, Thousand Oaks, CA, USA, 2010.

Vlad Stirbu. A restful architecture for adaptive and multi-device application sharing. In *WS-REST '10: Proceedings of the First International Workshop on RESTful Design*, pages 62–66, ACM, New York, NY, USA, 2010.