Reengineering to the Web: A Reference Architecture

Uwe Zdun
Specification of Software Systems
Institute for Computer Science
University of Essen, Germany
zdun@acm.org

Abstract

Reengineering existing (large-scale) applications to the web is a complex and highly challenging task. This is due to a variety of mostly demanding requirements for interactive web applications. Usually high performance is required, old interfaces still have to be supported, high availability requirements are usual, information has to be provided to multiple channels and in different formats, pages should contain individual layout across different channels, styles should be imposed over presentation, etc. To achieve these goals a variety of different technologies and concepts have to be well understood, including HTTP protocol handling, persistent stores/databases, various XML standards, authentication, session management, dynamic content creation, presentational abstractions, and flexible legacy system wrapping. In a concrete project, all these components have to be integrated properly and appropriate technologies have to be chosen. On basis of practical and theoretical experience in the problem domain, in this paper, we try to identify the recurring components in reegineering projects to the web, lay out critical issues and choices, and conceptually integrate the components into a reference architecture.

1. Laying out the Problem Domain

Reengineering to the web most often means to develop an interactive web application that is connected to existing legacy components or (partly) replaces them. That is, the legacy application has to be provided with an (additional) interactive web interface that (a) decorates the outputs of the system with HTML markup and (b) translates the (e.g. form-based) inputs via the web browser into the (legacy) system's APIs. In our experience, the domain of building interactive web applications is often underestimated. Thus, in many upfront designs simplistic architectures and implementation techniques are chosen that lead to severe maintenance problems as the web application evolves. Note that some web applications, developed from scratch, have similar characteristics, therefore, they often face similar maintenance problems as well.

In Figure 1 we can see such a simplistic three-tier architecture for interactive, web-based applications. A web user agent, such as a browser, communicates with a web server. The web server "understands" that certain requests have to be handled interactively. Thus, it forwards the request and all its information to another module, thread, or process that translates the HTTP request to the legacy system's APIs. An HTML decorator has to build up the appropriate HTML page out of the system's response and trigger the web server to send the page to the web client.

Using this simple architecture for a given legacy application, we can derive a simple process model for migrating an application to the web. In particular, the following steps have to be performed in general. Of course there are several feedback loops during these steps, and they are not in particular order:

- *Providing an Interface API to the Web:* To invoke a system's functions/methods and decorate them with HTML markup, we first have to identify the relevant components and provide them with distinct interfaces (which may, in the ideal case, exist already). These interfaces have to satisfy the web engineering project's requirements. To find such interfaces, in general, for each component there are two possibilities:
  - *Wrapper:* Wrap the component with a shallow wrapper object that just translates the URL into a call to the legacy system's API. The wrapper forwards the response to an HTML decorator or returns HTML fragments by itself.
  - *Reengineered/Redeveloped Solution:* Sometimes it is suitable to reengineer/redevelop a given component completely, so that it becomes a purely web-enabled solution. E.g. this makes sense,

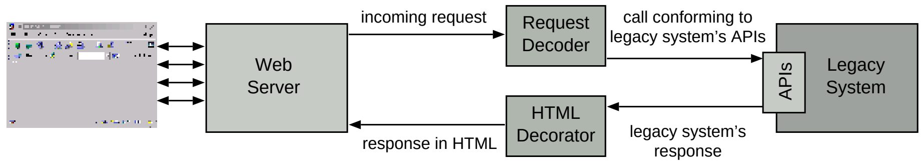

Figure 1. Simplistic Three-Tier Architecture for (Re-)engineering to the Web

when the legacy user interface has not to be supported anymore.

- *Implementing a Request Decoder:* A component has to map the contents of the URL (and other HTTP request information) to the legacy system's API, or the wrapper's interface respectively.

- *Implementing an HTML Decorator Component:* A component has to decorate the legacy system's or wrapper's response with HTML markup.

- *Integrating with a Web Server:* The components have to be integrated with a web server solution. For instance, they can be CGI-programs, run in different threads or processes, or may be part of a custom web server.

In our experience, this architecture is in principal applying for most reengineering efforts to the web and for many newly developed, web-enabled systems. However, this simple architecture does not model the major strategic decisions and the design/implementation efforts involved in a large-scale web development project well. There are many issues, critical for success, that are not tackled by this simplistic architecture, including: technology choices, conceptual choices, representation flexibility and reuse, performance, preserving states and sessions, user management, and service abstraction.

In this paper we will survey and categorize critical aspects in modern web development. Thus we will step by step enrich the simple architecture presented above to gather a conceptual understanding which components are required for reengineering a larger system to the web. The idea is to document and integrate the solutions that have shown their success in practice. We will use a reference architecture to combine the various elements of the architecture in such a way that we can incrementally evolve web applications. Moreover, as a side effect, we will build up a framework to categorize web development environments and packages. An important goal is to assemble a feature list for mapping requirements to concrete technological and conceptual choices so that new projects can easier decide upon used technologies.

In the course of this paper we will explain some examples from different web development environments, including PHP [2], WebShell [21], and ActiWeb [16], to illustrate how identified features can actually be implemented. Due to the limited paper length, we cannot give a full technology overview, thus we use these examples because we feel they let us easily illustrate a few implementation choices. However, there are many other great solutions not mentioned explicitly, and in some projects it may be required to implement certain functionality from scratch.

First, we will discuss *legacy system wrapping*, i.e. enabling the legacy system to be connected to web components. Afterwards, we will summarize the *HTTP protocol handling* elements that have to be accessible for a reengineering project to the web. Then we discuss the choices and implementation variants for *dynamic content creation*, that is, content decoration with HTML and ensuring consistent representation styles. Finally, we will bring these fragments together and present them in an integrated reference architecture.

2. Legacy System Wrapping

An important part of migrating a legacy system to the web is wrapping an existing system. Wrappers are mechanisms for introducing new behavior to be executed before, after, in, and/or around an existing method or component. Wrapping is especially used as a technique for encapsulating legacy components. Common techniques of reusing legacy components by wrapping are discussed in [19]. Moreover, wrappers are often used to extend object-oriented structures, as discussed in [3]. There are several ways to implement wrappers in object-oriented languages, including simple Adapters [8] and more sophisticated schemes.

The Component Wrapper pattern [11] accesses a black-box component using a central placeholder for component access. A Component Wrapper is a white-box for the component's client and enables us to introduce changes and customizations, e.g. with Decorators and Adapters. Therefore, the pattern is often used for legacy wrapping. During the process of legacy wrapping in the web context some other central concerns have to be considered:

- *Mapping Calls into the Legacy System*: When we have decoded a URL, we have to look up the responsible wrapper and invoke it.
- *Cross-Cutting Concerns*: Some concerns are cutting across several elements of a legacy interface and/or several elements of a business transaction. Since HTTP is stateless, HTTP requests/responses and business transaction elements are usually not synchronous. Therefore, we have to integrate the wrapped legacy system with session management and state preservation means. Since we usually want to decouple such concerns from the legacy application, they have to be implemented on the wrapper. There are some high-level techniques for implementing such wrappers, including Aspects [13] and interception techniques on the component wrapper, as discussed in [9].
- *Multiple Channels*: Often different requests coming from different clients, communicating over different channels, have to be supported. It should not be required to change the business logic every time a new channel has to be supported or a new service is added to the application. Wrappers often have to be written for other channels than HTTP as well. All these wrappers should be integrated, and if possible, code should be reused.
- *Service Abstraction*: Often the legacy system provides multiple services as well. Therefore, there should be an abstraction for service providers. Again, these have to be dispatched on the wrapper.

In many business systems, the problems of multiple channels and service abstraction are occurring together. They are often resolved by a Service Abstraction Layer architecture [22]. A Service Abstraction Layer is an extra layer to the business tier containing the logic to receive and delegate requests. In Figure 2 we can see that the Service Abstraction Layer abstracts from different service providers, and that it implements different channel adapters to support calls via different protocols.

When pages have to be generated dynamically, as it is required in most interactive web applications, the Service Abstraction Layer may serve for more purposes: it may be a Message Redirector [10] for indirecting symbolic calls (e.g. encoded in an URL string) to actual implementations. This indirection can be implemented in a configurable way.

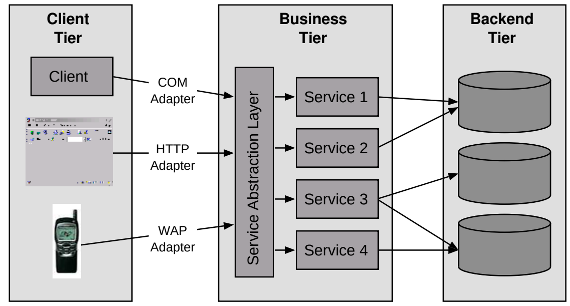

Figure 2. Service Abstraction Layer Architecture with Service Providers and Channel Adapters

For instance, in ActiWeb [16] the Message Redirector maps calls from URLs to web objects. It validates that the web object is exported and that it exports the targeted method. Otherwise an error is raised. Moreover, cross-cutting concerns can be registered here, such as user authentication and other security services.

3. HTTP Protocol Handling

A basics of any interactive, web-based application is the HTTP protocol [6]. Therefore, an obvious technology choice, that has to be made, is which web server should be used. For delivering static web pages only a few forces have to be considered in the technological decision, such as prices of web server products, ease-of-use, supported functionalities of the HTTP protocol, performance, and perhaps authorization and logging capacities. As we will see in the remainder of this paper, there are a lot of other forces that have to be considered as well, when web pages have to be generated dynamically. These vary in different web development environments and include the support for programming languages, packages, models of web decoration, means for integrating with (sub-)systems, persistence/database services, process/thread models, etc. In this section we will concentrate on the differences in HTTP protocol handling with relevance to reengineering to the web.

3.1. HTTP Request Handling

Usually, HTTP request handling is transparently performed by the web server. However, for many task we require at least a conceptual understanding of this process. Often we have even to manipulate request handling elements manually to obtain the desired result. Therefore, it is important that the chosen server technology offers interfaces to access and manipulate the vital parts of HTTP request handling.

In many web servers there is some kind of request handler or worker object that handles an individual request. On reactive servers events, such as client requests, are handled with a Reactor [18] that dispatches the connection events. A central Acceptor in an Acceptor/Connector [18] variant listens on the port. It establishes a socket connection for incoming requests and creates the request handler, or it invokes the responsible request handler, if it is already existing. This architecture is discussed in detail in [15].

There are several parts of such an architecture that web-enabled components, wrapping a legacy component, may be interested in:

- *URL Handling and Mapping*: In an interactive web application we have to map URLs to calls understandable for the legacy system's API. Important tasks are: encoding/decoding URLs (see Section 3.2), form decoding and accessing of form data (see Section 3.3), and mapping of URLs to calls (see Section 2). Ideally, all these tasks would be handled automatically by a web development environment, but often they have to be implemented by hand.

- *Error Handling*: Each server has to have a form of error handling, at least to handle the HTTP errors (such as setting the return code to 404, if a page is not found). Usually interactive web applications may have custom error events, and they are possibly able to handle errors on their own. Thus there should be at least a means to get and set the error state. E.g. PHP [2] additionally allows for setting custom error handlers and for writing to the server's error log. Sometimes the error handler can be extended or overloaded as well, as in [15].

- *Header Interpreter*: For incoming requests the HTTP header information has to be parsed. It contains the HTTP method (such as GET, PUT, POST, etc.), content length, content type, and other header information. It should be possible for an interactive web application to retrieve all this information so that, for instance, it can be used for the URL-to-call mapping decision (see Section 2). For example, in ActiWeb [16] it is possible to overload all parts of the header interpretation process and to retrieve all meta information from the header. E.g. the mobile code facilities of the framework transparently distinguish ordinary calls from agent migration by observing these information.

- *HTTP Redirect*: Some calls should be redirected to another destination. This can be done by creating a redirected request by hand or by using HTTP redirect (as specified in [6]).

3.2. URL Handling

To let an interactive web application map a given URL to a call, understandable for the legacy system's interface, we first have to decode the URL. For instance, in ActiWeb [16] web objects are running in so-called places that are unambiguously identified by host name and port. All web objects are also identified unambiguously by an URL. To let object-oriented calls be invoked using URLs, they are automatically transformed using the web standard CGI encoding/decoding (e.g., spaces are transformed to '+'). The general form for object-oriented calls via an URL in ActiWeb is:

```
http://hostname:port/objName+methodName+arguments
```

The *URL Decoder* of the place automatically transforms such URLs into an object-oriented call (see Section 2 for a discussion of the mapping architecture). To enable embedding calls into responded web pages (e.g. to integrate hyperlinks within a legacy application's web pages) we have to generate URLs as well. Thus there has to be a *Call Encoder* that encodes a call to the legacy system in an URL.

WebShell [21] allows for converting a given list of key and value pairs into a CGI encoded URL. PHP [2] only supports standard URL encoding and decoding, as well as parsing an URL into its fragments. WebShell, PHP, and ActiWeb support base64 encoding and decoding to transfer binary data encapsulated in HTTP requests.

An important part of generating URLs is to generate a self-reference. That is, a string-based identifier to the current process, object, method, function, etc. has to be obtained to build up a URL-encoded call to itself. This is required (a) to build up conceptual entities on the web (such as web objects) that handle a given task on their own and (b) to avoid hard-coding of names, IDs, or other identifiers, so that these information can change without breaking the application. Both goals are not well supported in many loosely-coupled CGI applications but are important for maintaining the system. "(a)" enables encapsulation and separation of concerns and thus eases understandability. "(b)" enables changeability of identifiers. Generating self-references can, for instance, be automated using introspection or reflection functionalities of the used programming language (if it supports them).

Every interactive web application requires means to encode calls, to decode URLs, and to generate self-references. Ideally all these tasks would be handled automatically by the web development environment, however, in many web applications these important tasks are only hand-coded. This may lead to considerable efforts during maintenance. For instance when object or method names change, we have to search for all occurrences of dependent hand-built self-references, scattered across the code. It is even worse, if the schemes of URL decoding or call encoding should be changed, or if an additional scheme should be support, because in this cases all occurrences have to be changed at once. For instance, both Actiweb [16] and WebShell [21] provide means for automated URL encoding, call decoding, and self-reference generation.

3.3. URL-Encoded and Multi-Part Form Data

On the web, data, to be transfered to the server, is usually obtained through forms. Thus, for any web application that does not only show its information on the web, but is interactive as well (i.e., it allows for queries and changes of the information), the form data has to be transfered to the legacy application.

In general, form data may be URL-encoded or provided as multi-part form data. We have already discussed URL encoding in the previous section. In principal this mechanism can be used to append any form data to a URL, like:

`http://<base-url>?elt1=content1&elt2=content2...`

As described in the previous section, this form can easily be mapped to arguments of methods in an automated fashion. However, this form of transfer has some limitations. First, the content is visible on the browser's location bar. That means, confidential information may be visible, and the URLs are cumbersome to edit for the users. Moreover, some browsers do not accept very long URL strings.

At least when long contents should be transfered (such as files), multi-part form data should be used. It requires support for the POST HTTP method and the encryption type `multipart/form-data` (see RFC 1867). Moreover, base64 de-/encoding, supported by most web development environments, is required to transfer binary data. Multi-part form data encodes the message in MIME format (see RFC 1521) within the HTTP request.

For example ActiWeb, WebShell, and PHP support both schemes. ActiWeb and WebShell support a generic evaluation scheme, so that an application does not have to know which variant is used. In ActiWeb [16] for both variants a lightweight form data object is created which can be introspected and queried for the form data. In WebShell [21] so-called form variables are provided, i.e. a link to a specified global variable. In PHP [2] we have to distinguish the two forms by hand.

3.4. Session Management and State Preservation

The HTTP protocol is stateless, however, most legacy applications are at least in part requiring states. For instance, when a user has to log in and out, there has to be a session maintained for that time period. To map a stateless client request properly to the correct session, we have different options:

- *URL Encoding*: We can encode vital information, such as user name and password, in the URL, by attaching them as standard URL parameters to the base URL. This works in almost any setting, but we have to be aware that some browsers have limitations regarding the length of the URL. Moreover, the information are readable on the user's screen, and they are sent in unencrypted form. For sensitive applications we have to use further security measures (see Section 3.5).
- *HIDDEN Form Fields*: We can embed information in a form that is hidden from display by using HIDDEN form fields. However, of course, they are readable as plain text in the HTML page's source.
- *Cookies*: Cookies are a way to store and retrieve information on the client side of a connection by adding a simple, persistent, client-side state settable by the server. However, the client can deactivate cookies in the user agent, thus, cookies do not work always. Cookies can optionally be sent via a secure SSL connection.

On server-side the session context has to be kept persistent. Thus, the session management usually requires persistent objects. ActiWeb [16] includes various persistent stores that transparently make objects persistent. PHP [2] integrates persistent database connections via SQL links. It is important that the persistent connection has not to be build up each time a different request handler requires it (e.g. handled through persistent connections). Moreover, it should be possible to share open connections established earlier. Usually, sessions have to expire after a while and a re-login is necessary. If only a user identity has to be preserved, we can also use only HTTP basic or digest authentication (see next section).

3.5. Authorization and Encryption

Security issues are relevant to most reengineered web applications, at least for login with user name and password. Moreover, secure communication or securing transfered data is required. These issues have to be tightly integrated with session management. In general, we require user authentications and encryption as typical means to secure an interactive web application, in particular:

- *HTTP Basic Authentication*: The definition of HTTP/1.1 [6] contains some means for access control of web pages, called basic authentication scheme. This simple challenge-response authentication mechanism lets the server challenge a client request and clients can provide authentication information. The basic authentication scheme is not considered to be a secure method of user authentication, since the user name and password are passed over the network in unencrypted form.

- *HTTP Digest Authentication*: Digest Access Authentication, defined in RFC 2617 [7], provides another challenge-response scheme, that does never send the password unencrypted, which is the most serious flaw of basic authentication.

- *Encrypted Connection (Using SSL)*: Using a secure network connection, supported by most servers, we can secure the transaction during a session.

- *URL Encryption*: To avoid readability of encoded URLs we can encrypt the attached part of the URLs.

3.6. Logging, Testing, and Deployment

An important aspect of most web applications is the required high availability. Usually a web site should run without any interruptions. This has several implications that have to be considered when choosing technologies, concepts, implementations, etc. At least the following functionalities are usually required:

- *Permanent and Selective Logging*: All relevant actions have to be logged so that problems can be traced. Some selection criteria should be supported, otherwise it may be hard to find the required information out of the possibly large number of log entries. Sometimes (e.g. for legal reasons) even more information has to be logged, such as user transaction traces for e-commerce stores. Thus logging has to be highly configurable. E.g. WebShell [21] supports log filters that distinguish different log levels and redirect log entries to different destinations, like files, stdout, or SMS.

- *Notification of Events*: In cases when certain events happen, such as certain error states, a person or application should be notified. For instance, when an error message is recurring, an email may be sent to the system's administrator.

- *Testing*: Load generators and an extensive regression test suite are required for testing under realistic conditions.

- *Incremental Deployment*: Dynamically loadable components enable incremental deployment so that the application has not to be stopped to deploy new functionality. For such tasks, a highly dynamic programming language allows for changing functionality on the fly. Often introspection facilities are required as well to find out the current architectural setting of the system.

4. Content Creation and Representation

Content creation and representation for the web is usually the central task of a project dealing with migration to the web. This seems to be a relatively simple effort, especially when a given legacy system with a distinct API should be reengineered to the web. In our experience, this viewpoint is fundamentally wrong and leads to severe problems when the resulting system have to be further evolved later on. Often we find systems in which the HTML pages are simply generated by string concatenation, such as the following code excerpt:

```
DString htmlText;
char* name = legacyObject.getName();
htmlText.append("<BR> <B> Name: </B>");
htmlText.append(name);
...
```

This hard-coding of HTML markup in the program code may lead to severe problems regarding extensibility and flexibility of content creation. Content, representation style, and application behavior should be changeable ad hoc.

When the user agent (e.g. a web browser) sends a request to the web server, the requested page may be available on the file system of the server or it may be dynamically created. In this paper we will concentrate on interactive, web-based applications. However, it is important to consider the balance between static and dynamic content creation carefully. For a small web development project it may be acceptable to create all pages on-the-fly. For a large-scale project the balance between static and dynamic content is often crucial. Creating and delivering an HTML page on-the-fly usually costs significantly more performance than delivering a static page from the file system.

In this section, we will discuss two conceptually different approaches for decorating with HTML markup: template-based approaches and constructive approaches. Moreover, web-based applications typically have to represent the business logic on the web in a coherent way, say, in a common representation style. In Section 4.3 we will discuss ensuring common representation styles.

4.1. Template-Based Approaches

Template-based approaches, such as PHP [2], ASP, JSP, or ColdFusion, let developers write HTML text with special markup. The special markup is substituted by the server, thus, a new page is generated, which composes content dynamically into the template. For instance, in PHP the PHP code is embedded by escaping HTML e.g. by using a special comment:

```
<body>
  <h1> <?php echo("My PHP Heading\n"); ?> </h1>
</body>
```

On the first glance, the approach is simple and well-suited for end-users, say, by using special editors. The HTML design can be separated from the software development process and can be fully integrated with content management systems.

However, real web-based applications usually require more complex interactions than simply expressible with templates. Sometimes, the same actions of the user should lead to different results in different situations. Most approaches do not offer high-level programmability in the template or conceptual integration across the template fragments. Thus, the same fragments of a template often have be implemented redundantly. Application parts and design are not clearly separated. Thus template fragments cannot be cleanly reused. Complex templates may quickly become hard to understand and maintain.

Sometimes integration of different scripts can be handled via a shared dataspace, such as a persistent database connection. Sometimes, we require server-side components for integrating the scripts on server side.

4.2. Constructive Approaches

Constructive approaches generate a web page on the fly with a distinct API for constructing web pages. Usually they are not well-suited for end-users since they require knowledge of a full programming language. However, they allow for implementing a more complex web application logic.

The most simple constructive approach is the CGI interface [4]. It is a standardized interface that allows web servers to call external applications with a set of parameters. The primary advantages of CGI programming are that it is simple, robust, and portable. However, one process has to be spawned per request, therefore, on some operating systems (but, for instance, not on many modern UNIX variants) it may be significantly slower than using threads. Usually different small programs are combined to one web application. Thus conceptual integrity of the architecture, rapid changeability, and understandability may be reduced significantly compared to more integrated application development approaches. Since every request is a new process and HTTP is stateless, the application cannot handle session states in the program, but has to use external resources, such as databases or central files/processes.

A variant of CGI is FastCGI [17] which allows a single process to handle multiple requests. The targeted advantage is mainly performance. However, the approach is not standardized and implementations may potentially be less robust.

A similar approach integrated with the Java language are servlets. They are basically Java classes running in a Java-based web server's runtime environment. They are a rather low-level approach for constructing web content. In general, HTML content is created by programming the string-based page construction by hand. The approach offers a potentially high performance.

Most web servers offer an extension architecture. Modules are running in the server's runtime environment. Thus a high performance can be reached and the server's feature (e.g. for scalability) can be fully supported. Examples are Apache Modules [20], Netscape NSAPI, and Microsoft ISAPI. These approach often mean to code the web page construction in C, C++, or Java at a fairly low level. Moreover, most APIs are quite complex, and applications tend to be monolithic and hard to understand.

Custom web servers, such as AOL Server [5], TclHttpd [23], WebShell [21], Zope [14], the Ars Digita community system [12], or ActiWeb [16] provide more high-level environments on top of ordinary web servers. Often they provide integration with high-level languages, such as scripting languages, for rapid customizability. Most often a set of components is provided which implement the most common tasks in web application development, such as: HTTP support, session management, content generation, database access/persistence services, legacy integration, security/authentication, debugging, and dynamic component loading. Some approaches, such as WebShell, offer modules for web servers, as in this case Apache, as well.

WebShell [21] uses global procedures to combine web pages:

```
proc dl {code} {
  web::put "<dl>"
  uplevel $code
  web::put "</dl>"
}
```

To code these procedures we have to interfere with HTML markup. However, we can avoid it later on when we combine such procedures to HTML fragments:

```
  dl {
    b {My first page}
    em {in Web Shell}
  }
```

Here we have created a `<dl>` list entry with a bold and an emphasized text in it.

In Actiweb [16] user interface builders are used to build up pages. Thus we can use the same code to construct pages for different user interface types. A simple example just builds up a web page:

```
HtmlBuilder htmlDoc
htmlDoc startDocument \
  -title "My ActiWeb App" \
```

```
  -bgcolor FFFFFF
htmlDoc addString "My ActiWeb App"
htmlDoc endDocument
```

We instantiate an object `htmlDoc`, then start a document, add a string, and end the document. The user sees no HTML markup at all. The page is automatically created by the builder class.

Template-based approaches are usually faster than purely constructive approaches. Therefore, many approaches combine the template-based and the constructive approach. However, often the two used models are not well-integrated, that is, the user has to manually care for the balance between static and dynamic parts.

4.3. Representation Styles

Representation styles are important to provide a cooperate identity for a set of related pages. E.g. logos, surrounding frames, style sheets, etc. should be provided. A simple way to achieve a consistent style are cascading style sheets (CSS). However, some required changes to the content are not solely expressible by cascading style sheets, but require behavioral specifications of the changes to be performed. A simple example is that the incoming format is not HTML, but e.g. HTML fragments or XML. Some systems, such as the document management system DocMe, directly manipulate given HTML files so that they are presented in the style of their document class.

"Conceptually cleaner" solutions for such constructively imposed representation styles use intermediate formats, such as XML, for document description. The representation styles are themselves defined in a decoupled (and usually generic) way. For instance XSLT can be used for defining the transformations independently. The transformer may as well be a programming language script or class. In [24] a generic architecture of such transformations is presented that allows for integrating all these variants.

5. Reference Architecture Overview

In the previous sections, we have identified and described different parts of viable architectures for reengineering to the web, and we have described open issue and problems in the various domains involved. In this section we will combine these elements to an integrated reference architecture. Figure 3 shows the interplay of the various components, discussed in this paper, as a reference architecture overview. We can see that this more realistic overview of a web engineering architecture is significantly more complicated than the simplistic web reengineering architecture, as explained in Section 1.

In the figure we can easily see the different elements of a web engineering architecture, as they were discussed in the previous sections. In a reengineering project, usually we would start up by wrapping one or more essential components with Component Wrappers. When other channels have to be served as well, such as the legacy user interface, we will most likely introduce a Service Abstraction Layer to integrate each Component Wrapper as a service. In this layer, we also compose them with channel abstractions. To integrate different Component Wrappers and compose them with the other elements of the reference architecture, often a Message Redirector is used to indirect calls. Most often is also triggers (parts of) URL decoding, since URLs are used as symbolic calls in the Message Redirector. These elements of legacy component wrapping are discussed in Section 2 in detail.

Different components are used by these architectural elements to decorate the results with HTML markup and provide a common representation style (as discussed in Section 4). Moreover, session management, as discussed in Section 3.4 has to be tightly integrated with the components that wrap the legacy system's session abstractions (if there are none, perhaps they have to be implemented in the Component Wrapper layer or they can be omitted).

URL decoding, as in Section 3.2, is usually triggered either by the Reactor in the HTTP protocol handling components or by the Message Redirector. The components should have a uniform way to receive encoded information such as form data. The second part of URL handling, call encoding, is usually triggered directly from the components or the Message Redirector automatically replaces parts of the results, returned by the legacy components.

The requests are received using the Acceptor and Reactor of the HTTP protocol handling architecture (see Section 3), and are directed to the Message Redirector using a request handler. The Message Redirector handles the call to the legacy API and all conversions. After the call to the legacy system returns, the Message Redirector can send a resulting HTML page back to the request handler object, which sends it to the Connector of the web server. Moreover, there are a set of (optional) components for authentication, secure communication, HTTP redirections, and error handling integrated.

The reference architecture is a rather conceptual model. That is, some concrete architectures vary in details, do not provide certain components, or do provide additional components. Having said that, we believe most applications reengineered successfully to the web and most web development environments can be well categorized into the reference architecture.

In our experience, there are the following main uses of the reference architecture:

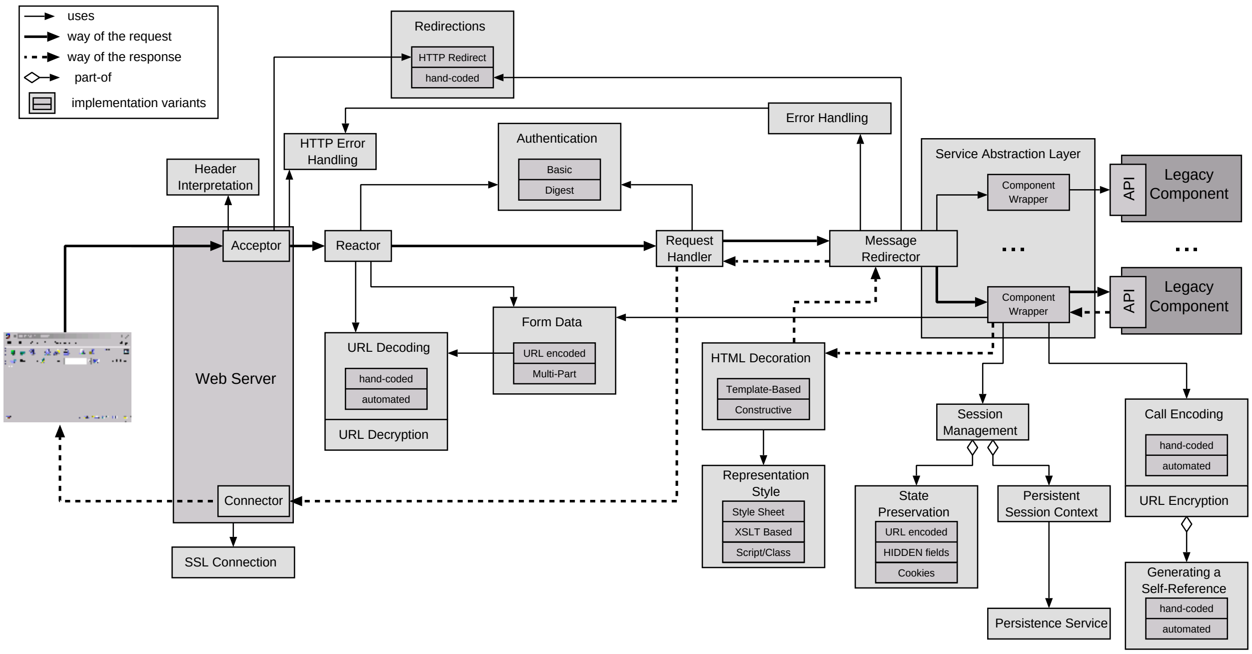

Figure 3. Reference Architecture Overview: Bringing a Legacy Application to the Web

- *Understanding Web Engineering:* Web Engineering and related disciplines are emerging, but they are not fully understood yet. In our experience many developers and projects do not see the challenging issues of large-scale web development on the first glance. Thus, first architectures and solutions are often not capable to cope with typical requirements. In turn, some typical requirements, such as non-stopping applications, very high hit rates, significant difference in the content presented, and multi-channels, are hard to handle during maintenance, if the architecture is not designed for incremental evolution.

- *Categorization of Web Development Environments:* In this paper, we have given a few examples from different web development environments to illustrate the described functionality. We believe the full range of web development environments can be categorized in how far they support each part of the reference architecture.

- *A Framework for Conceptual and Technological Decisions*: In (consulting) projects with the aim of reengineering an application to the web, the decision for a concrete web development environment and estimations of the project duration are rather hard. This is mainly because the "web wrapper" is highly application-context dependent. Therefore, it may be difficult to decide on first glance which components of the reference architecture are required in which variant. Moreover, it has to be assessed which web development environment or package provides which components of the reference architecture, how different packages can be integrated, and which components of the reference architecture have to be programmed by hand. The reference architecture is (a) designed for incremental evolution, so that such decision can be made stepwise, and (b) it provides a conceptual framework for discussing such decisions.

- *A Process Model for Bringing Applications to the Web*: As we have already sketched in Section 1 there is an almost sequential path (with several feedback loops) for bringing an application to the web. It is usually more or less the reverse order of the way a request takes through the system (see Figure 3). That is, first a component is wrapped, then it is provided with HTML decorations, forms, and styles, then it is tested on the web, and finally advanced features, such as authentication are added. Usually one functionality is brought to the web at a time, therefore, there are often many iterations through the same loop. A similar process model can be found in [1]. This model rather builds on one concrete application case study; therefore, some aspect of the broader model, sketched here, are not tackled.

Note that there are, of course, some severe limitations for a reference architecture. That is, the reference architecture only guides us to the important issues in web engineering, but does not replace practical experience. Many aspects, such as how different packages can be integrated, how static and dynamic aspects can be balanced, how the performance can be enhanced, how the memory consumption can be limited, are often highly application dependent. Moreover, some important aspects are not tackled in this paper, even though they are highly relevant to many web engineering projects. The reason is that some examples, such as integrating databases and persistence services, are more general than pure web development (and thus discussed elsewhere). Other examples, such as performance evaluations or scalability issues, are rather technology dependent. But, of course, all these aspects have to be considered as well in a concrete web engineering project.

6. Conclusion

In this paper we have identified and discussed the viable elements of software architectures for bringing a legacy application to the web, and integrated the various elements into an architectural overview. In our experience from several reengineering projects (and also projects that build web applications from scratch), the domain of web engineering is often underestimated in its complexity and diversity. Thus rather simplistic models, architectures, and implementation techniques are used for first implementations, what implies severe problems for maintaining the web application. We believe, a better understanding of the problem domain in advance can significantly decrease these problems. The reference architecture documents the architectural elements that can be found in numerous successful web applications. However, it is carefully designed in such a way that it can be applied incrementally; that is, the architecture has not to be applied as a whole before it can be used, but additional features and components can be added in a stepwise manner.

References

[1] L. Aversano, G. Canfora, A. Cimitile, and A. de Lucia. Migrating legacy systems to the web: an experience report. In *5th European Conference on Software Maintenance and Reengineering (CSMR'01)*, Lisbon, Portugal, Mar 2001.

[2] S. S. Bakken and E. Schmid. PHP manual. http://www.php.net/manual/en/, 1997-2001.

[3] J. Brant, R. E. Johnson, D. Roberts, and B. Foote. Evolution, architecture, and metamorphosis. In *Proc. of 12th European Conference on Object-Oriented Programming (ECOOP'98)*, Brussels, Belgium, July 1998.

[4] K. A. L. Coar. The WWW common gateway interface – version 1.1. http://cgi-spec.golux.com/draft-coar-cgi-v11-03-clean.html, 1999.

[5] J. Davidson. Tcl in AOL digital city the architecture of a multithreaded high-performance web site. In *Keynote at Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

[6] R. Fielding, J. Gettys, J. Mogul, H. Frysyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, 1999.

[7] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. Http authentication: Basic and digest access authentication. RFC 2617, 1999.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[9] M. Goedicke, G. Neumann, and U. Zdun. Design and implementation constructs for the development of flexible, component-oriented software architectures. In *Proceedings of 2nd International Symposium on Generative and Component-Based Software Engineering (GCSE'00)*, Erfurt, Germany, Oct 2000.

[10] M. Goedicke, G. Neumann, and U. Zdun. Message redirector. In *Proceeding of EuroPlop 2001*, Irsee, Germany, July 2001.

[11] M. Goedicke and U. Zdun. Piecemeal migration of a document archive system with an architectural pattern language. In *5th European Conference on Software Maintenance and Reengineering (CSMR'01)*, Lisbon, Portugal, Mar 2001.

[12] P. Greenspun and E. Andersson. Using the ArsDigita community system. *ArsDigita Systems Journal*, Feb 1999.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, Finnland, June 1997. LCNS 1241, Springer-Verlag.

[14] A. Latteier. The insider's guide to Zope: An open source, object-based web application platform. *Web Review*, 3(5), March 1999.

[15] G. Neumann and U. Zdun. High-level design and architecture of an http-based infrastructure for web applications. *World Wide Web Journal*, 3(1), 2000.

[16] G. Neumann and U. Zdun. Distributed web application development with active web objects. In *Proceedings of The 2nd International Conference on Internet Computing (IC'2001)*, Las Vegas, Nevada, USA, June 2001.

[17] Open Market, Inc. FastCGI: A high-performance web server interface. http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm, 1996.

[18] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.

[19] H. M. Sneed. Encapsulation of legacy software: A technique for reusing legacy software components. *Annals of Software Engineering*, 9, 2000.

[20] R. Thau. Design considerations for the Apache server api. In *Proceedings of Fifth International World Wide Web Conference*, Paris, France, May 1996.

[21] A. Vckovski. Tcl Web. In *Proceedings of 2nd European Tcl User Meeting*, Hamburg, Germany, June 2001.

[22] O. Vogel. Service abstraction layer. In *Proceeding of EuroPlop 2001*, Irsee, Germany, July 2001.

[23] B. Welch. The TclHttpd web server. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

[24] U. Zdun. Dynamically generating web application fragments from page templates. In *Proceedings of Symposium of Applied Computing (SAC 2002)*, Madrid, Spain, March 2002.