WHITE PAPER

Electronic forms solutions using XML and PDF

A technology primer for the Adobe XML architecture

Executive summary

The purpose of this white paper is to describe how Adobe's technology and products for electronic forms use Portable Document Format (PDF) and XML to meet the strict requirements of enterprise environments. As increasing numbers of people both inside and outside the firewall need to interact with information stored in core business systems, specialized client interfaces are not a feasible solution due to licensing, security, and manageability issues, among others. Electronic forms are often the best solution in these cases, but they present their own challenges of fidelity to precise visual layouts, integration with data formats supported by core systems, and generation of final documents from multistep processes.

The Adobe XML architecture addresses these challenges by leveraging the universal Adobe® Reader® client and a sophisticated processing model that separates user data, visual layout, and business logic. Users can work with familiar Adobe PDF forms, while applications can interact with XML data formatted in the schema used by the organization. These benefits are crucial in industries where a wide variety of users interact with information from a wide variety of systems such as insurance, financial services, and government.

The architecture assigns processing responsibility for each different type of information to a different module, accessible through its own application programming interface (API). This sophistication enables a clean separation of responsibility among the tasks of layout design, data integration, and application processing. Layout designers get visual tools, systems integrators provide their desired XML schema, and application developers use their off-the-shelf XML tools to extract the information they need. As a result, enterprises can quickly and cost-effectively extend their core processes and make them more efficient.

TABLE OF CONTENTS

Adobe®

Introduction

A wide variety of users need to interact with core systems

To participate in enterprise business processes, employees, customers, and partners need to interact with data stored in their own or others' enterprise applications—whether providing information or retrieving it. The primary users of such systems have specialized client interfaces that support frequent interaction. However, they often constitute a relatively small fraction of all system users. Secondary users such as employees in other departments, managers, customers, suppliers, and partners can represent the majority of users, but their occasional use of core systems makes specialized client interfaces impractical and inefficient. Instead, they require a much more general interface that provides a means to interact with many different types of core systems and supports their various work styles.

Simultaneously delivering ease of use and ease of integration is a challenge

Electronic forms are an ideal general interface because most processes are already driven by their paper counterparts. However, creating an effective electronic forms solution presents a number of technical challenges:

- They often require a precise visual layout that corresponds to an existing paper version of the form, to increase user adoption and meet regulatory requirements.
- In many cases, the time required to complete a form exceeds the time a user can remain online, so electronic forms must support offline work.
- The rise of XML as a standard for defining industry-specific data formats and integrating internal systems makes the support of arbitrary XML data a necessary capability.
- Meeting the needs of both end users and integration developers means that electronic forms must also have a great deal of intelligence. Such intelligence includes scripting functions that assist the user with form completion and validation logic that ensures the smooth flow of accurate data into core systems, reducing the need for error correction later.

The Adobe XML architecture provides a single reusable framework to meet both the technical and business requirements.

The technical challenges facing forms solutions are quite daunting. However, a single, reusable framework can cleanly address the issues of fidelity to visual layout, flexibility of data integration, intelligent interaction, multichannel delivery, and offline work. The Adobe XML architecture combines a universal client with a comprehensive array of server-based solutions. The technology embedded in Adobe Acrobat® 6.0 and Adobe 1.5 enables enterprises to deliver more capable forms solutions more quickly because is it uses XML as a universal data integration mechanism and shifts the forms design role from a skilled developer to a skilled business user.

Operational scenario

To design an appropriate architecture for an electronic forms solution, it is imperative to start with a clear understanding of the problem. There are three primary criteria:

- **Simplify integration with the system environment.** Most enterprises have a software infrastructure that consists of a combination of Web front ends, custom applications, packaged applications, and backend databases. Because these pieces run on a variety of operating systems, use different application development paradigms, and were written at different times, most enterprises have a significant investment in integrating these disparate components. Typically, the integration platforms either use XML natively or have an adapter that supports XML. In some cases, industry or internal standard XML vocabularies may shape the format of integration.

  Support XML as the standard for systems integration

- **Facilitate a smooth and efficient development process.** There are roughly four types of tasks involved in electronic forms development. In increasing order of complexity—and thus cost—they are layout design, Web development, systems integration, and application programming. Therefore, a cost-effective electronic forms solution should maximize the use of visual designers and Web developers and minimize the use of integration developers and application developers.

  Must minimize the number of expensive tasks and provide clean separation of tasks

  Of course, all four types of development tasks will be involved in most forms applications. The form solution can only minimize, not eliminate, the need for expensive systems integration and application development tasks. Therefore, the forms solution must also make it possible to keep the different types of tasks separate. Complicated scripts developed as part of a Web development task should be cleanly encapsulated within the form definition instead of mixed in with the layout design. Layout design must remain insulated from systems integration through an abstraction layer. Applications should interact with the form through published APIs or file formats. These mechanisms enable a clean separation of responsibility for different tasks, smoothing the development process.

- **Support offline work.** Most business processes require a variety of forms at different stages of the process since different people require different information for their role in the process. For example, when applying for a car loan, applicants first fill out a form requesting approval for a loan amount. Once they buy a particular car, they must submit a form with all the detailed information about that vehicle. Finally, after they and the bank agree on the final terms, they have to fill out and sign the actual promissory note. For many people, downloading all the forms at once may be more convenient than repeatedly going back to the bank's Web site to get the next form. Moreover, gathering information for each of these steps may take some research, by making phone calls to an insurance agent or reviewing paper documents provided by the seller. Applicants would probably find it more convenient to fill out individual pieces of information offline as they gathers them, rather than writing them down and submitting them at once via a standard HTML form.

  Enable users to fill in forms at their convenience.

Solution requirements

Given the preceding operational scenario, the minimum requirements for an enterprise-scale electronic forms solution are as follows:

- **Arbitrary XML formats.** Forms will touch upon data managed by multiple core systems. The integration technology that allows these systems to collaborate will at least support and may require XML. Furthermore, the form must support the enterprise's chosen schemas for this XML. Because it is desirable to minimize the amount of work integration developers have to do to process the form XML, the architecture must support data in arbitrary XML formats.
- **Integration with Web services and databases.** The form must be able to integrate with core business systems using standard Web services protocols such as SOAP, HTTP, ADO, and JDBC. In some cases, the form may need to be able to connect to these types of services or data or business logic, which may not be able to be embedded in the form itself.
- **Online and Offline interaction.** Users may not be able to remain online for the entire time required to complete a form. Therefore, the architecture must support online and offline interaction. In some cases, users may work on a form online as well as offline, so the architecture should accommodate the mixture of both modes.
- **Self-contained business logic.** Forms should contain all the information necessary for a user to complete them. This information may include business logic that assists the user in completing the form. It may also include instructions on how to connect to Web services or back-end databases to retrieve any up-to-date information the user needs to complete the form. However, because users may sometimes work on the form offline, this business logic must be self-contained in the form interaction environment.
- **Easy to design and change.** Supporting the breadth of enterprise processes possible with electronic forms may require the design of large numbers of such forms. Consequently, the solution should maximize the amount of work that visual designers can perform. The architecture should include a tool that makes it easy for designers to create and maintain forms.
- **High fidelity visual presentation.** The primary purpose of electronic forms is to support secondary users. Such users are often sensitive to the layout of form, which has led to both legal and internal standards for paper form layouts. Therefore, the electronic forms architecture must offer visual presentations that provide high fidelity to such standards.
- **Final documents.** Many form interactions such as applying for a loan or insurance result in an official document. So while an enterprise may design a set of forms for maximum ease of use in inputting required information, it may still need to provide the user with a final document that conforms to a format mandated by a regulatory body or internal policy. The architecture should make generating a final document as easy as possible.
- **Universal client.** In supporting secondary users of core systems it cannot be assumed that they have any specific client software. Forms must be able to execute within a standard Web browser. However, due to the need to support offline work, only working within a browser is insufficient. The electronic forms architecture must provide an offline-capable universal client as well.

Technical goals

To meet the solution requirements, certain technical goals must be achieved:

- **Data abstraction layer.** Supporting arbitrary XML and making form design easy can be competing concerns. An XML format may not reflect the organization of a form most convenient for users. A designer shouldn't have to redesign the same form layout to accommodate slightly different XML formats. If a particular piece of data gets dynamically retrieved from a database or applications, a form designer shouldn't have to treat it differently. Therefore, the architecture requires a data abstraction layer.

- **Layout-data binding.** A data abstraction solves the problem of treating different formats and sources the same from a design and execution perspective. However, it raises the issue of making sure that the form information provided by a user finds its way to its intended place. The architecture must have a means to bind data elements from the data abstraction layer to the form fields specified in the layout template. This binding must support bidirectional propagation of changes in one location to the other.

- **Standalone scripting.** Self-contained business logic means that the architecture must include a standalone script interpreter. Different scripting tasks lend themselves to different types of scripting languages. Calculating the value of one field from the values in others and evaluating validation constraints require an expression language accessible to nonprogrammers. More complicated behaviors, such as altering accessible fields based on results from a query to a directory service, require a full-featured programming language. So the architecture must support both types of languages.

- **Paginated layout model.** Paper form layouts are paginated. Maintaining fidelity to such layouts requires that the form architecture have an internal layout model that supports pagination. However, because electronic forms may expand or contract to accommodate specific instance data, this support must be highly sophisticated to ensure that it can render arbitrary user data within the specified pagination constraints.

- **Robust security model.** Interaction with core systems from a universal client that supports offline interaction naturally raises security issues. Moreover, the execution of business process tasks by secondary users has inherent security requirements of its own. Therefore, the architecture must support a robust security model that provides strong authentication, supports digital signatures, and minimizes the opportunities for malevolent code.

Accommodating multiple views

Map different layouts to same data format

People and systems often have different perspectives on business processes. For example, an application that manages research proposals for a government agency may store company information, technical information, and financial information for the same proposal in the same XML format. However, users may want this information presented on different forms because different people in their companies fill out each category. It would be very inconvenient if the agency had to break apart the official XML schema definition into three special definitions just for this purpose. Instead, each type of form should be able to access the part of the official schema that is relevant to its purpose.

Map different data formats to same layout

The reverse is certainly possible as well. For example, customers typically perceive an order as a single transaction. However, different systems usually handle credit checks, order fulfillment, and order shipping, so they expect that single order form to generate multiple XML messages. It would be impractical if users had to fill out three separate forms at the same time for each order just to accommodate the system architecture. Instead, the form should be able to place information from different form fields into different XML documents, each corresponding to the message expected by a particular back-end system. This flexibility is only possible with well-designed data abstraction, binding, and scripting capabilities.

How the Adobe XML architecture works

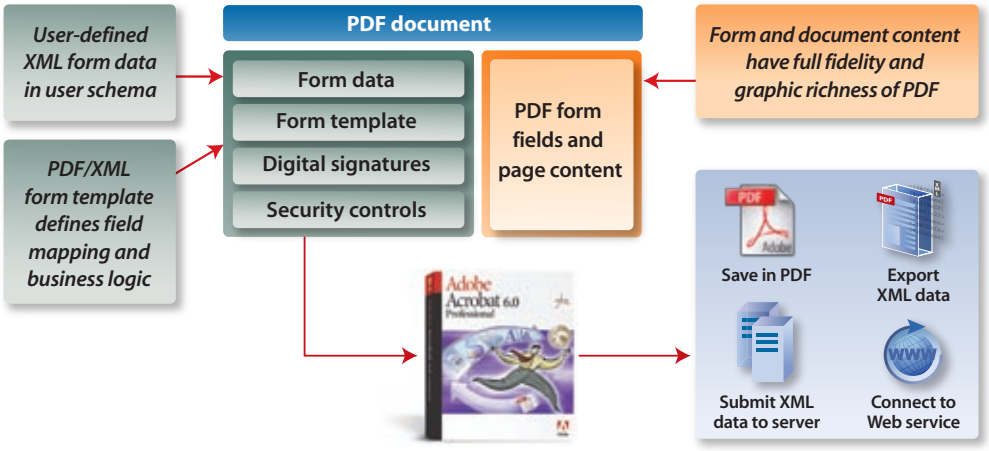
Use native XML container that is interchangeable with PDF

To meet these requirements, Adobe's XML architecture embraces several core design principles. Because it must support both XML data in an arbitrary format and seamless round-tripping to core systems that use XML for integration, it uses a container for document components that is also XML. However, due to the enormous installed base of software that supports PDF, this XML container is interchangeable with PDF. The XML format and its corresponding PDF representation constitute the document perspective on the architecture.

Separate responsibility for processing among internal modules

To accommodate the different perspectives of users and systems, the architecture cleanly separates the form template and form data. While the data may often consist of XML documents, the prevalence of non-XML data sources necessitates a flexible abstraction layer that can seamlessly support both. The architecture combines the separate template and data at runtime to dynamically build the form representation with which users interact. The modules that manage the template, data, and form constitute the behavioral perspective on the architecture (Figure 1).

Figure 1: PDF acts as a container for multiple types of data and content



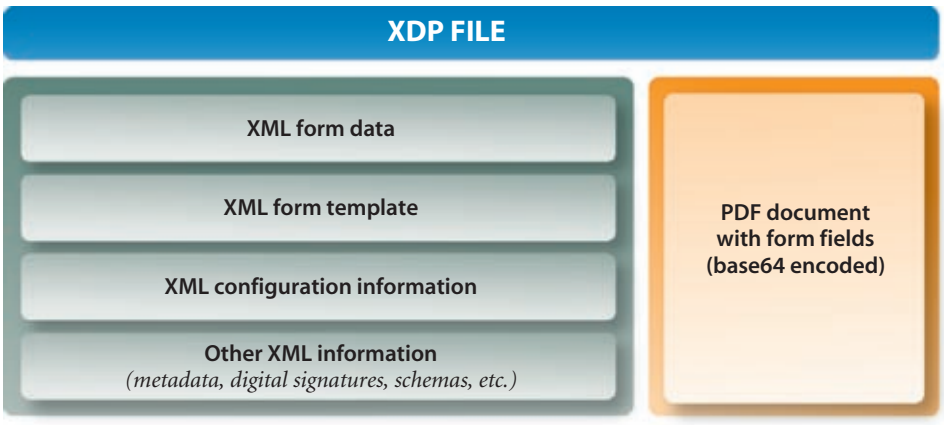
XDP and PDF file formats

XDP contains five types of subassemblies

An XML Data Package (XDP) file is simply an XML file that encodes a PDF file in XML, along with the form data and a form template. Because an XDP file is an XML file, standard XML tools, system interfaces, and Web services can work with it, and the XML data is directly accessible. An XDP file contains five types of subassemblies, as shown in Figure 2.

- **XML form data.** This component is the user data encoded according to an arbitrary XML schema chosen by the form developer during the design phase. The schema can be an industry standard, the enterprise's standard, or completely customized.
- **XML form template.** This component contains all the form intelligence, including the mapping of XML form data to PDF form fields as well as all the business logic that controls the interactive behavior of the document, such as calculations and data validations.
- **XML configuration information.** The XML form template uses this component as a global reference for database and Web services SOAP connections.
- **Other XML information.** XDP files can include custom XML information such as a schema file to facilitate validation, XML digital signatures, content metadata to facilitate archiving, or data used by a custom digital document application. The ability to include the XML schema in an XDP is a key capability in ensuring that other systems can easily process the XML data contained in a form.
- **PDF file.** XDP files provide all the traditional PDF benefits of precision document layout and high-fidelity printing by embedding the PDF in an XML element as base64 encoded data.

Figure 2: Anatomy of an XDP file



XDP and PDF are convertible at the client and the server.

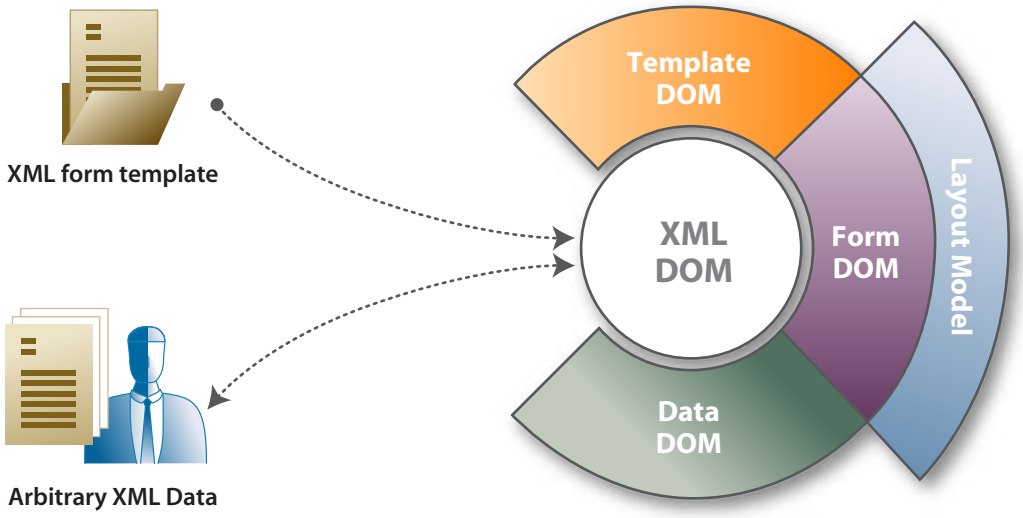
Because PDF and XDP are equivalent and interchangeable representations of the same underlying electronic form, enterprises can choose which one to use depending on the specific requirements of the process. PDF offers advantages when files size is important, documents are large, or forms contain supplementary data such as images. XDP offers advantages when forms must travel in XML workflows, systems need to manipulate form data with off-the-shelf XML processing components, or forms must reside in an XML repository. Adobe provides server software components for converting one representation to another.

Template DOM

Uses form-specific DOMs on top of W3C DOM

Whenever a piece of software such as a form client accesses an XDP or PDF document, it loads the persistent information into a set of runtime modules. The Adobe XML architecture includes a number of different modules, whose programmatic interfaces are called document object models or DOMs. As Figure 3 shows, the Adobe DOMs partition the operations specific to electronic forms into several pieces. However, at the lowest level, they manipulate XML data using the standard W3C DOM so that the in-memory representation of data remains synchronized with the XML document representation.

Figure 3: Anatomy of an XDP file



Template DOM is responsible for layout, binding, and scripts .

The Template DOM manages information related to visual layout, data binding, and interaction rules. This information follows a declarative model, defining form components and their relationships but not how to render them. The template includes static and dynamic components. Static components include boilerplate text, drawing objects, and areas that contain other static components. Dynamic components include fields and subforms that contain other dynamic components. Static components are the same for every instance of a form that uses a template, but dynamic components change depending on the data contained in the instance.

Provides default binding rules based on name and hierarchy

Dynamic components are bound to this instance data so that fields specified in the template reflect this data. The architecture includes default binding rules that map XML data elements to form template fields based on similarities in their respective structural hierarchies as well as element and field names. These default rules simplify the job of template designers because they only have to provide a specific mapping when structure and naming aren't sufficient to calculate the binding. Once binding is complete, changes to the fields or the data are synchronized through cooperation with the Data DOM discussed in the next section.

The Template DOM also manages scripts in either FormCalc or ECMAScript. FormCalc is an expression language used primarily to calculate the values of one field from the values of other fields, much like a spreadsheet formula language. A FormCalc script attached to a field automatically creates a dependency on other fields, and the script automatically executes every time those fields change or whenever the Template DOM receives a global recalculation request. ECMAScript supports a much more general set of operations such as popping up interaction dialogs and even directly accessing the different Adobe DOMs. ECMAScript enables programmers to use the architecture to turn electronic forms into powerful applications that can even look up a user's security role in an enterprise directory and then suppress access to certain form fields based on this role. By providing both FormCalc and ECMAScript, the architecture accommodates the different requirements for layout designers and Web developers.

Two scripting languages for two different purposes

Data DOM

As discussed in the previous section, the Template DOM binds template fields to instance data. The Data DOM manages this instance data by providing a data abstraction layer. Most instance data will reside in XML documents, in which case the Data DOM bridges high-level form data manipulations to low-level W3C DOM operations. However, certain types of instance data may represent very dynamic data managed by back-end applications, so the Data DOM must also support these other sources. To this end, the Data DOM can also synchronize form data with information in any ODBC-accessible database or any SOAP-accessible application.

Data DOM is responsible for abstraction of user form data.

The Data DOM dynamically synchronizes the instance data with the form representation managed by the Form DOM discussed in the next section. So if a user changes the value of a field, the Data DOM receives the event from the Form DOM and propagates the change. Depending on whether the underlying data source is in an XML document or a dynamic data source, the Data DOM will make the change to the in-memory W3C DOM representation or flush it to a back-end data source. Similarly, if a script changes data in the Data DOM or if a trigger fires to update data from a back-end system, the Data DOM will notify the Form DOM of this event.

Synchronizes display data with underling XML data

Form DOM

The Template DOM manages form definitions for all instances of the same type, and the Data DOM manages the data specific to each instance. They come together to support runtime form interactions in the Form DOM. The information managed by the Form DOM is completely derived from the combination of information in the Template and Data DOMs. It is the runtime representation of the form as a whole. While applications and scripts can directly manipulate the Template and Data DOMs, the most common case is for an application to use the Form DOM.

Form DOM constitutes runtime form representation.

For example, suppose a user interacts with a form through Adobe Acrobat. Whenever a user clicks in a region corresponding to a form field, types in data, and hits return, Acrobat calls an operation in the Form DOM to change the data. The Form DOM then collaborates with the Template DOM to execute any scripts that depend on the value of that field and then collaborates with the Data DOM to implement the change in the underlying data representation.

Collaborates with Template and Data DOMs

Layout model

The layout model controls the page layout of forms. Layout model information includes document characteristics as page size, headers and footers, and pagination. Usually, this type of information is derived completely from the Form DOM and the target rendering device. However, it is possible to exercise some control over the layout model. Programmers can include scripts in the form template that directly manipulate the layout model, but usually specific applications include logic for exercising such control. Adobe Form Server and Adobe Document Server are examples of such applications.

Possible to directly manipulate layout at runtime

Leveraging XML and PDF

Three types of organizations use the architecture

It's important to remember that the sophistication of the internal architecture does not translate into complexity for developers, integrators, and designers. In fact, the very reason that this architecture is so sophisticated is to shield them from the underlying technical complexity of the electronic forms problem. These different stakeholders can choose to work at the level of sophistication necessary to solve a particular problem, for example:

- In-house forms developers can work with easy-to-use visual tools and off-the-shelf XML processing components.
- Systems integrators (SIs) can directly manipulate XDP files.
- Independent software vendors (ISVs) can write code to the various DOM APIs.

Enterprises use visual layout tools and standard XML components.

End-user organizations will, of course, be the most common users of the architecture. They will also be the ones that have to know the least about how it works. From their perspective, they only need two types of software to work with it. First, layout designers will need the visual layout tool to create form templates. Second, server developers will need off-the-shelf XML processing components to populate and extract form instance data. Because most designers are already comfortable with similar types of layout tools and most developers are already comfortable with particular XML processing components, it takes very little effort to get started with electronic forms applications.

SIs can manipulate XDP files to support industry frameworks.

SIs may want to manipulate XDP files as part of distributed application processing. A common situation in electronic forms application development is for an SI specializing in a particular industry to have its own framework that makes it easier to process that industry's forms. Such an SI could leverage the Adobe architecture by developing very generic templates and then programmatically customizing them by altering the XDP files to change default values, change validation constraints, customize calculation formulas, suppress certain subforms, or modify the security permissions for different user roles. Because XDP is a documented XML format, Adobe can easily help such SIs work with the portion of the XDP schema that is relevant to their requirements. They can then use off-the-shelf XML processing components to perform the desired operations.

ISVs can uses DOMs directly for highly customized form applications .

ISVs may want to use this architecture to embed advanced electronic forms capabilities in their products. Packaged application vendors may want the ability to flexibly generate forms from within their packages and could use the various DOMs to customize one of Adobe's server products or build their own modules. Because Adobe's own product developers have used these APIs to build Adobe's form-related products, they serve as both a proof point and technical resource for ISVs that want to engage in this level of customization.

E-gov case study

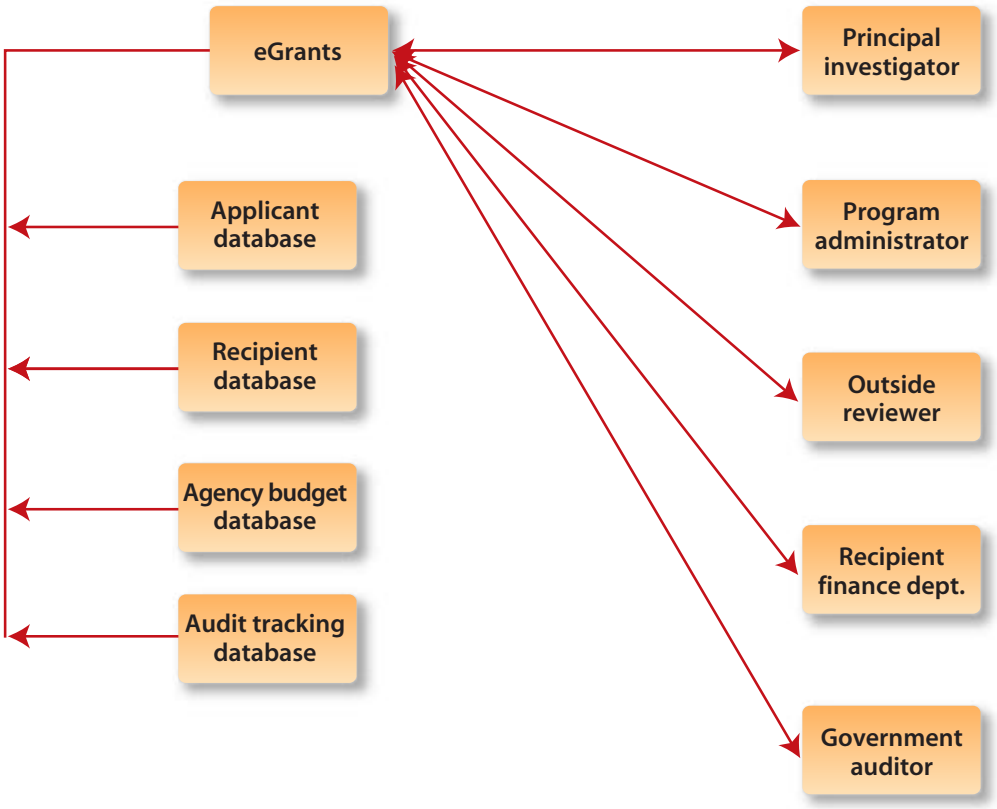
E-gov is a U.S. electronic government initiative.

One of the earliest end-user organizations to adopt electronic forms technology is the U.S. government. It has a broad initiative to improve the efficiency of its processes, generally known as electronic government or e-gov. The government is one of the largest consumers of information technology in the world and therefore has evolved a diverse set of back-end systems and integration technologies to connect them. As most citizens have experienced, most government tasks executed by people involve the use of forms. Therefore, it should be no surprise that the U.S. government is a leading proponent of electronic forms.

eGrants standardizes the process of government grants.

The government has identified 24 different e-gov initiatives. One of these initiatives is eGrants. The goal of this initiative is to improve the efficiency of grant making by providing a single online portal for grant customers that speeds grant applications, facilitates the review process, eliminates the submission of redundant information, minimizes the administrative burden on recipients, and standardizes the collection of financial report data. A large subset of such grants includes those for research and development, administrated by such agencies as the Defense Advanced Research Projects Agency (DARPA), the Homeland Security Advanced Research Projects Agency (HSARPA), the National Institutes of Health (NIH), the National Institute for Standards and Technology (NIST), and the National Science Foundation (NSF). Figure 4 illustrates the key role forms play in this initiative. Adobe produced a prototype for a pilot project sponsored by the Federal CIO Council Web Services subcommittee to demonstrate interoperability between electronic forms and Web services for this application.

Figure 4: eGrants application topology



In this scenario, a principal investigator (the term for a scientist involved in a research project) who is a potential recipient of a government grant would use the eGrants portal to download, fill out, and then submit the eGrants form. Part of the data in this form goes to the applicant database. The data from the proposal gets converted into two types of review forms. One containing all the information goes to the program administrator at the grant making agency, the standard term for a government organization that administers grants. Another containing only technical proposal information goes to outside reviewers who provide their expert opinions. The program administrator receives all of these outside review forms and makes a decision. In practice, many agencies have multiple rounds in their awards processes, so this cycle may repeat.

Proposal submission and review involves many individuals

If accepted, the principal investigator and the finance department of the company receive additional enrollment forms. These forms go to the program administrator for review and upon acceptance, some of the data gets stored in the recipient database and some goes into the agency's budget system. As a project continues, the principal investigator files forms with periodic reports that go to the program administrator, while the finance department files forms with spending data that go to the program administrator, and the agency budget system. At certain milestones, a government auditor must review the project finances and file forms that go to the finance department, the program administrator, and the audit tracking system.

Administering accepted proposals involves additional processes

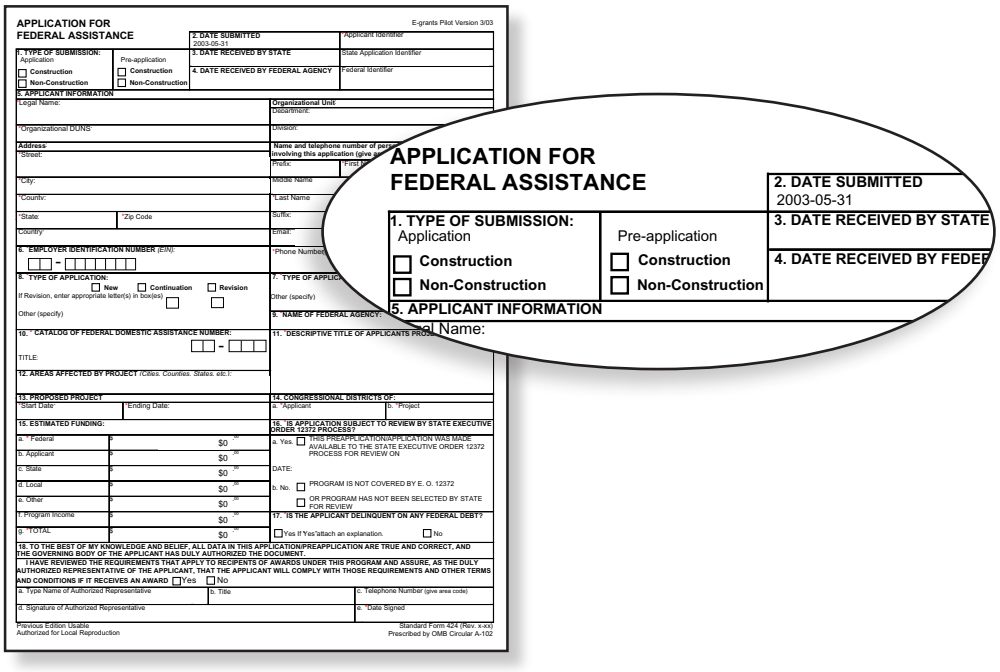
Form design process

While this process seems complicated, much of the software infrastructure for automation is already in place. Because of the general e-gov directive to use XML for integration, XML formats exist in many cases. Moreover, the infrastructure for adapting existing systems to use these formats is, to a great extent, already in place. With paper forms already established for this process, automating it is really an issue of bringing all the pieces together. Figures 5 through 12 illustrate the five steps necessary to accomplish this task using a forthcoming Adobe form design tool:

Design process has five easy steps

1. **Import existing form.** The existing standard for grant applications is SF424. As Figure 5 shows, importing the form layout is simply a matter of loading a PDF file. If this form were generated by scanning in a paper form or outputting a file through a printer driver, the designer would have to overlay PDF form fields. If it came from an existing PDF form, these fields would already exist.

Figure 5: Existing form layout



2. **Import existing schema.** There is a corresponding XML schema for SF424 as shown in Figure 6. Using a standard file open dialog, the designer can load this schema as shown in Figure 7. Then all the data definitions from the schema appear in a list on the left-hand side of the screen as shown in Figure 8.

Figure 6: Existing schema

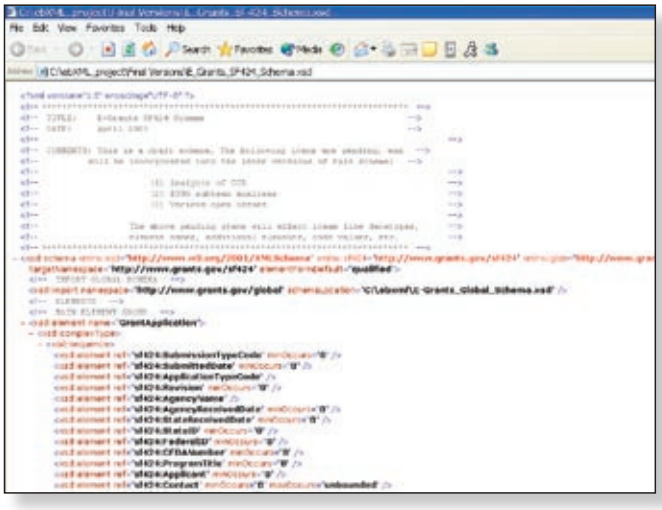


Figure 7: Loading the schema

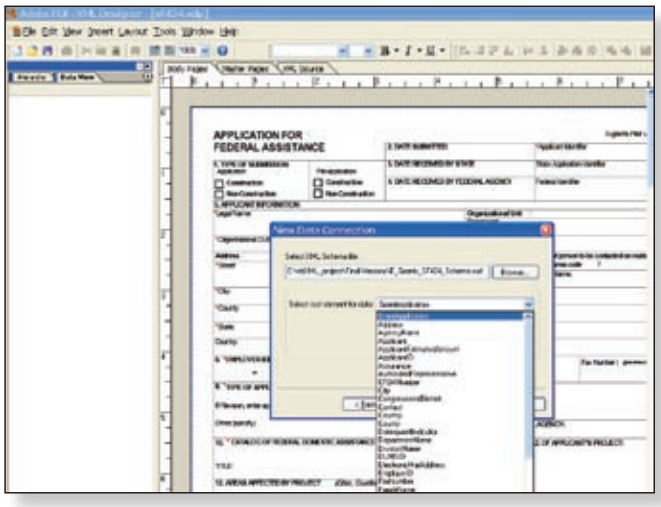
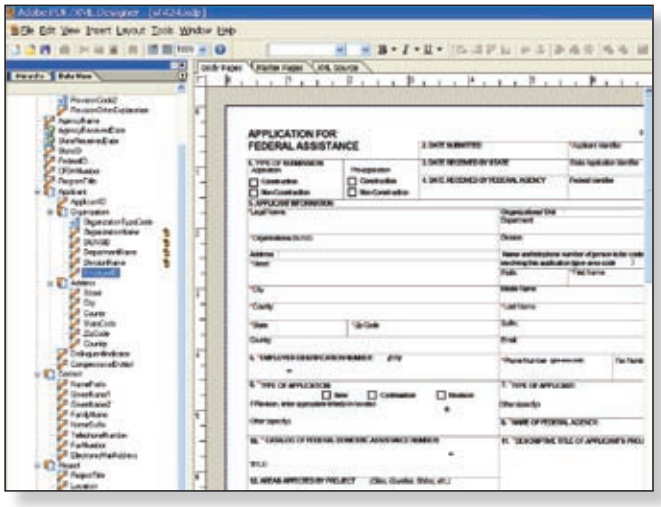
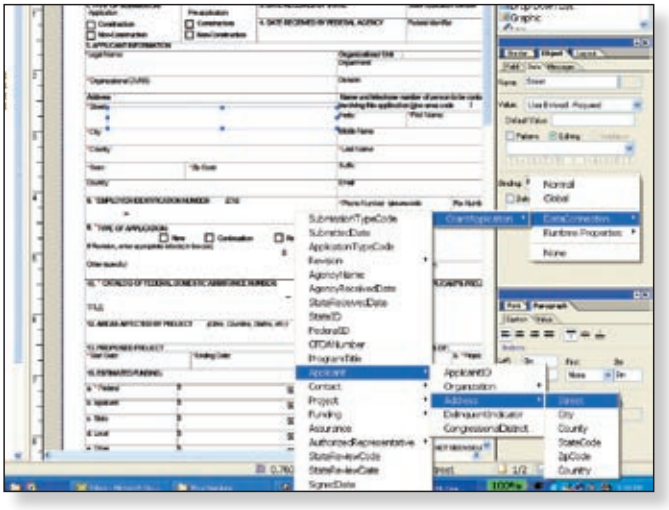


Figure 8: Schema data definitions



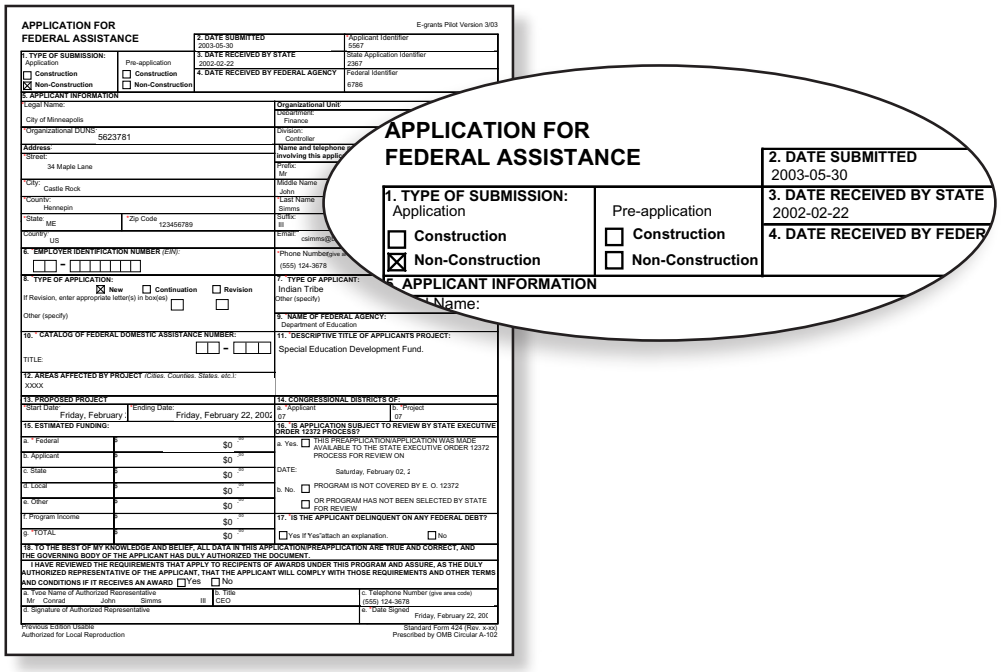
3. **Bind form fields to schema data.** Mapping a form field to a data definition is as simple as right clicking a form field and finding the correct element in a list. Figure 9 shows this action for the Street field. The designer would map all the other fields in a similar manner.

Figure 9: Mapping form fields to schema



4. **Fill out form.** Once the design is completed, tested, and deployed, users can access it through any electronic channel. They can download it from a Web site or receive it in their e-mail inbox. Then they use Acrobat or Acrobat Reader to fill it out as shown in Figure 10.

Figure 10: Filling out form



5. **Generate validated XML output.** Once a user fills out the form, there is a fully validated XML document compliant with the SF424 schema. Figure 11 shows how to save as XML, and Figure 12 shows the resulting document.

Figure 11: Saving as XML

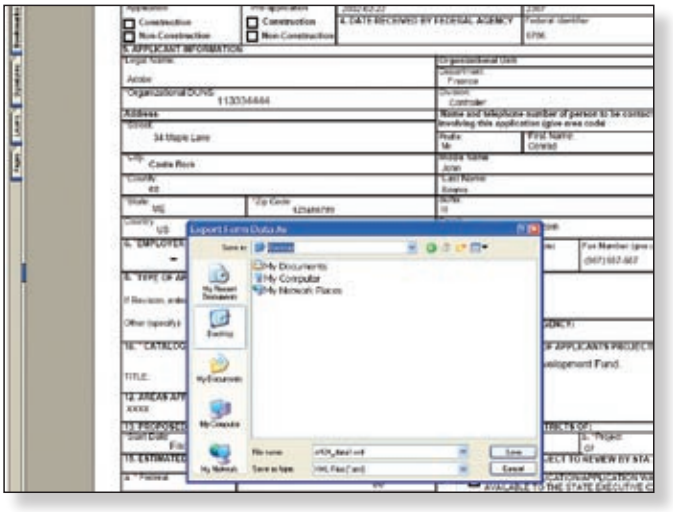
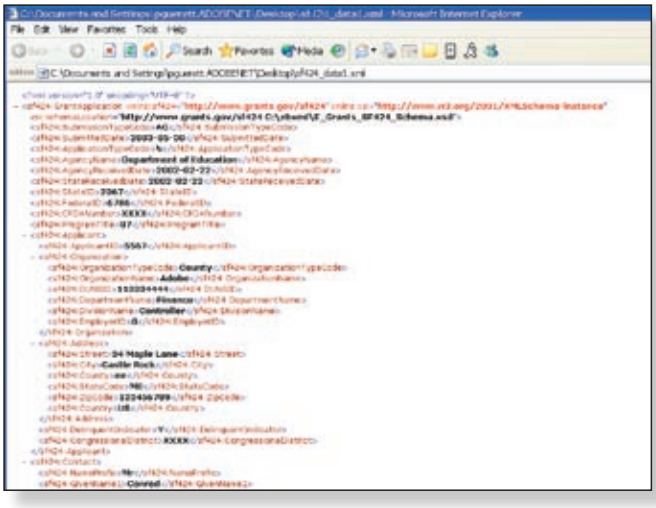


Figure 12: Resulting XML document



In practice, of course, most users would not generate a separate XML file. Instead they would submit the entire form PDF file via the Web or e-mail to the eGrants portal. A small piece of server software generic to all XML/PDF forms would extract the XML file and route it to the appropriate back-end applications. So by leveraging the existing form layouts and data formats, a visual design tool and small piece of server software are all that's necessary to finish automating a reasonably complicated process.

The application would submit XML via Web or Web service

The ease of this development process is by no means unique to e-gov. Many other industries have adopted standardized XML schema, have existing form layouts, or both. Manufacturing has been using the XML-based RosettaNet standard for years. The insurance industry has the standard ACORD schema and associated form layouts. The financial services industry has numerous XML standards. For all the examples of applications built on top of Adobe's XML architecture, go to *http://partners.adobe.com/asn/tech/pdf/samples.jsp*.

Many industries have standard XML vocabularies like E-Gov

Conclusion

The Adobe XML architecture separates visual layout from data schema definition. This gives enterprises an extremely flexible tool for creating and extending business processes that work seamlessly with core systems. An enterprise can start from an existing form layout and work towards core systems or start from an existing XML format and work towards end users. It can adopt electronic forms as a top-down strategic initiative or as a series of bottom-up tactical projects. No matter what the starting point or adoption approach, the Adobe architecture provides the catalyst necessary to marry user and system representations so that the enterprise can achieve a seamless flow of process data.

The Adobe XML architecture enables a seamless flow of data.

This carefully designed architecture addresses the issue inherent in extending enterprise processes. While users and systems may agree on individual data elements, they almost never agree on how to organize them. Users adopt a point of view based on the flow of their job tasks. Systems adopt a point of view based on the schemas of their databases. The sophisticated factoring of data and assignment to one of the DOMs combined with event-driven synchronization is what makes it possible to integrate these competing points of view.

Use of multiple DOMs integrate user and system points of view

By addressing this complexity within the underlying architecture, Adobe shields enterprise designers and developers from it. Designers simply import, modify, and design visual elements using a high-productivity layout tool just like the ones they've used for years. Developers simply populate, extract, and process the underlying XML data using the standard software components they've used for years. Together, they give users electronic forms that mirror the popular usage model of paper forms, but with all the additional intelligence and efficiency possible with software. As a result, enterprise business processes can extend farther and execute faster.

Therefore, layout design and systems integration are both easy

FOR MORE INFORMATION

about the Adobe XML architecture, please visit *www.adobe.com/xml.*

Adobe Systems Incorporated • 345 Park Avenue, San Jose, CA 95110-2704 USA • *www.adobe.com*

95002660 11/03

Adobe

Tools for the New Work™