A framework for automatic generation of web-based data entry applications based on XML

Volker Turau
University of Applied Sciences, FH Wiesbaden
Department of Computer Science
Kurt Schumacher Ring 18, 65197 Wiesbaden, Germany
turau@informatik.fh-wiesbaden.de

Keywords
Web-based data entry, frameworks, automated prototyping

ABSTRACT
This paper presents a framework for web-based data entry applications. It introduces a method for the conceptional and the navigational design based on a textual specification in the form of an XML-application. This forms the input to a code generation environment allowing for real automated prototyping. The environment produces fully functional skeletons for the web pages. Together with the framework classes they can be utilized for testing and for requirements review. They also form the starting point for the work of the presentation design. The main advantage of the framework is a clear separation between the presentation and the business logic. This allows work on each aspect to proceed in parallel along relatively independent but cooperating tracks. The framework has been implemented using Java Servlets and Java Server Pages.

1. INTRODUCTION
Data entry tasks appear as subtasks in almost every web-based application from E-commerce to clinical data collection. The major subtasks of a data entry task are: collection of the data through a sequence of input masks, validation of the data, generation of status respectively error messages and finally the persistent storage or preprocessing of the data for other back end applications. The ubiquity of web-browsers makes the web an ideal platform for remote data entry applications: There is no need to distribute special software and the data can be stored in a central database on the server. Creating complex web-based applications including data entry tasks is an effort in which developers with diverse backgrounds are involved. There are HTML layout and graphics designers, programmers, application domain experts, and database experts, among others. Very often these groups have communication problems due to their different views of the application. Customers are often more worried about the presentation than about data management issues. As a consequence the realization of the presentation layer consumes a substantial part of the life cycle of the application development. Changes in the business logic can have far reaching consequences, which may lead to aggravating changes in the presentation layer. This in turn can prolong the development time drastically. This calls for a clear decomposition of the application. Ideally the decomposition results in a set of objects that can be assigned to subteams based on their particular skills. This division of labor allows work on each object to proceed in parallel. Therefore, the following requirements are fundamental:

- The separation of stable code from more volatile code. Usually the parts that deal with presentation and user interfaces are subject to rapid and frequent changes. The business rules and the database schemas are less likely to change frequently.
- A simple process of assigning suitably trained human resources to accomplish the development task such that work can proceed along relatively independent but cooperating tracks.
- Data entry applications must be designed and tested, not only for normal situations, but also for as wide a range of anomalous situations as can be anticipated. The development process must allow testing at a functional level (decoupled from the final user interface) early in the development process.

Experience has shown that it is very useful to have rapid prototyping tools; i.e. tools which transform automatically a formal specification of an application into a fully functional version of the web pages of an application with a basic layout. This allows to concentrate on the functional aspects of the application with little or no initial effort on the layout.

In this paper we present a three-tier framework called Wizard for web-based data entry applications which implements all the above mentioned requirements. The formal specification is contained in a single XML file. There are
clearly defined interfaces for the persistency tier and for the implementation of the business logic. The specification is used as input for a code generation environment to generate a complete system prototype (including static and dynamic characteristics). The generated views carry only the functional semantics and have a default design. This prototype can be used for testing the complete cycle of the data entry process including the persistent storage and for requirements review. After some iterations of this process, a clear and stable interface is established, this forms the starting point for the presentation subteam. The automatically generated views form a skeleton for the final views. The framework includes a set of simple tags to access the provided functionality. These tags are loosely modeled after the tags provided by HTML and are therefore easy to use. Hence, there is no need to write any programming code for the design team. Furthermore, designers can use their favorite HTML-editors.

2. REQUIREMENTS

Larger Web-based applications are usually multiter applications, because they provide a lot of flexibility in choosing how to distribute application functionality across the tiers. Commonly three-tier architectures are used: the client tier provided by the browser, the web tier provided by the server and the database tier which holds persistent application data. An important issue is the subdivision of the application into objects and the assignment of these objects to tiers. Today it is common to develop frameworks for applications of this complexity. Frameworks enable the reuse of software designs respectively code and identify the responsibility of each object. They are therefore ideally suited to solve the problem in question.

The planned framework shall support the entry, validation and persistent storage of data utilizing web browsers as front ends. The framework must be independent of a particular application. The data entry will be performed through a sequence of web pages, similar in style to wizards known from personal computer applications. Users will be guided by the application. The order of the web pages is not defined a priori, it is driven by the user’s input. At some points the successor pages depend on the data entered, hence there are several paths through the application. For each possible path a different set of data entries may be collected. Users should never be confronted with questions that are not applicable. The following high level objectives for data entry applications must be obeyed [9, 11]:

- **Minimal input actions:** Making a choice by selection from a menu, or from radio buttons, rather than by typing in a lengthy string of characters, is preferred and eliminates the possibility of typographic errors.
- **No redundant data entry:** Redundant data entry is perceived as a waste of effort and is an opportunity for errors and inconsistencies.
- **Compatibility of data entry with data display:** The format of data entry information should be linked closely to the format of displayed information.
- **Informative feedback:** There should be a response by the system for user actions which influence the persistent storage.

- **Error Prevention and simple error handling:** The system should detect all errors made by users and offer simple, constructive, and specific instructions for recovery. For example users should not have to retype all input, but rather should need to repair only the faulty part. Erroneous actions should leave the system in a consistent state.

- **Easy reversal of actions:** As much as possible, actions should be reversible. The unit of reversibility may be a single data entry or the complete transaction.

Validation through the application on the server is mandatory. In addition a client-side application (e.g. through JavaScript) may perform validation in order to reduce response time. The complete application must not rely on the usage of Cookies for session management. The application must ensure that only complete and valid data are stored. A simple interface to access permanent storage systems which is independent of any particular storage system is required.

The knowledge of the system (i.e. how objects interact and which tasks they perform) should be kept in one central place. This makes a framework highly reconfigurable in order to accomplish changes or to adopt new requirements without programming (or minimal programming). There is a current trend to use plain text as a format for storing knowledge persistently. Hunt and Thomas call this the power of plain text [5]. The Extensible Markup Language (XML) is ideally suited for such a format, due to the fact that tools and parsers for XML are widely available.

The work to adopt the framework to a concrete data entry task should be accomplished without greater effort. The necessary programming should be minimal. The ideal case requires apart from the design of the web pages and the implementation of the storage interface only entries in textual configuration files. The design of the web pages should not involve any programming tasks (e.g. for server-side applications such as CGI-scripts). The information about the data entries, their grouping and ordering on web pages as well as the specification of the validation operations should be declared largely in a declarative style. It must be possible to include complex validating mechanisms (e.g. performed by external applications).

The framework should make the localization of applications (i.e. the adoption to an individual language) a simple task. The major requirement is a clean separation of the tasks for the different developer groups, enabling an independent development.

3. ARCHITECTURE

The framework is modeled according to the Model-View-Controller (MVC) design pattern. This pattern divides an application into three logical categories of objects: objects that deal with

- presentation
- business rules and data
- mapping user requests to actions on the business objects.

A view renders the contents of a model. It accesses data from the model and specifies how the data is presented to the user. Another task of the view is to forward user
commands to the controller. In a Web application a view corresponds to a web page and user commands appear as GET and POST HTTP requests transporting data entered by users into HTML-Forms.

The model represents the application data and the business logic that governs access, modification, validation and storage of data. A controller defines the application behavior by applying the business logic which in turn changes the state of the model. The controller also selects a view to display the state of the model. Whether the business logic is part of the controller or belongs to the model can be debated. We prefer to regard the business logic as a separate entity, this reduces the model to a structural representation of the applications state. Figure 1 shows the flow of control and data in the MVC architecture. The encircled numbers show the order of the actions.

![Figure 1: Flow of control and data](image)

The framework includes three business rules or actions:

- **Collect**: This action is responsible for processing the data entered by a user through a single page. This processing includes validating the data, updating the model and selecting the next view to be presented.

- **Store**: This action is responsible for the persistent storage of the data. Before the data is stored it is checked for completeness (users can manually manipulate URLs and parameters, thereby leave out individual steps) and a global validation is performed.

- **Navigate**: This user can always go back to a previous view and change the entered data. In some cases this may lead to a different path through the application. Navigate takes care that the application is still in a consistent state in this case.

Figure 2 shows how the MVC pattern is used in the Wizard architecture. The actions are further structured, such that parts of the business logic can be encapsulated in Enterprise Java Beans. This increases the reusability of these components and further isolates the back-office applications from the rest of the application.

4. SPECIFICATION

A central requirement upon the framework is a declarative specification of the application. This is realized through a XML-document `wizard.xml` which describes:

- all data entries with their value ranges
- how to validate the data
- a grouping of data entries and a mapping of the groups to web pages (the views)
- the ordering of the collecting steps
- the necessary actions.

Extracts of the document type definition for this XML-document are shown in Listing 1. The `wizard`-element is the root of the application. The attributes define a default language and the prefix of the available resource bundles for localization. The body of `wizard` consists of three sections specifying the data entries, the validation process and the views. In the following the elements of the DTD are briefly described.

A `field`-element describes a single data entry. The attribute `name` denotes the name of the entry. The name must be unique in the document. The optional attribute `range` references a `range`-element characterizing the legal values of this entry. Different fields may reference the same `range`-element. Ranges are primarily used for validation. If an attribute has a finite range of values, these values can be presented in a selection list. This reduces the probability of errors. The boolean attributes `implicit` and `multiple` have both default value `false`. If the value of `implicit` is `true`, then the value for this entry can be deduced from other entries and users will not be asked for the value (e.g. the name of a town can be inferred from its zip code). The attribute `multiple` signals that an entry is multivalued.

A `field`-element may optionally have a `rule`-element as a subelement. This element describes rules for the validation of the values and consists of a sequence of `rule`- and `rules`-elements. The data supplied by users is transported using the HTTP protocol. Since HTTP is a text-based protocol, all data is of type String. Ranges and validation are formulated based on this representation. It is the responsibility of the persistency layer to perform necessary conversions. A single rule is described by a `rules`-element. The required attribute `message` contains a logical key which refers to an error-message to be used when the application of this rule fails. The usage of logical keys provides a hook for internationalization. The attribute `class` `Name` refers to a class which performs the actual validation. All classes implementing the interface `Checkable` (part of the framework) are legal. The default class `RuleRule` tries to match the value with a regular expression specified via the attribute `pattern` or `patternRef`. In the first case a regular expression using the

![Figure 2: MVC pattern in the Wizard architecture](image)
Listing 1: Excerpts of the Wizard DTD

Perl 5 syntax is expected. In the latter case a regular expression in a predefined pool of expressions is referenced (examples are email, letters, notempty). Additional attributes can be introduced on demand. They are interpreted by the validating class. Different rules and rule-elements can be combined. The attribute relation describes the semantics of their combination. Legal values are or, and, all. The first is a disjunction and the other two are conjunctions. If the value is and then the validation stops after the first rule fails and only this error message is returned. If the value is all, all rules are applied and all error messages are returned.

Listing 2: Example declaration of four fields

is used to automatically generate a range class to be used at run-time. Listing 3 shows the declaration of three ranges.

Listing 3: Example declaration of three ranges

Every view-element represents a view of the application and is identified by a unique name. It has an additional display name to be used in hyperlinks referring to this view. Views are materialized as Java Server Pages. The name of this page is specified by the attribute jsp. The code generation component will generate a JSP-page with this name. The attribute fields contains a list of data entries (e.g. names of field-elements) to be collected by this view and the attribute action references a action-class to be executed on behalf of this view. Normally a view has a fixed successor view. Occasionally it is useful to define the successor view based on the value of a collected data entry. For this purpose a view-element can have select-elements in its body. Each such element defines a rule to determine a successor view. The first applicable rule defines the successor view, if no rule is applicable the view specified in the view-element is visited. Listing 4 shows two view definitions. The successor view of payment depends on the value of the data entry payment.

An advantage of using a DTD is that it can be used to validate the specification. For example the usage of the types ID, IDREF and IDREFs guarantees that there are no dangling links. The usage of default values for many attributes makes the specification documents rather concise.

5. THE FRAMEWORK

As depicted in figure 1 the framework consists of four parts which are either included in the framework (controller,
Listing 4: Example declaration of two views

basic actions), automatically generated from the specification (model, views) or supplied by the application provider in form of classes implementing fixed interfaces (persistent storage, rule classes). The code generator environment generates from the file wiz.xml the model in form of a Java Bean. Basically there is a private property of type String for each field and public read/write methods. Furthermore, for each view element a JSP-page is generated. The details are presented in the next section.

The initialization of the controller is performed by a dedicated servlet. This servlet reads the file wiz.xml and builds an internal representation of the specification. It initializes helper classes (e.g. range and rule classes), reads the resource bundles needed for localization etc. This class must be extended only if external classes are required for ranges or actions. To do so, users must override the method load-ExternalResourcex. In this method all external classes are initialized and made available to the web application. The controller itself is part of the framework. It receives all requests and dispatches them to the corresponding actions. These update the model and inform the controller about the successor view.

As mentioned earlier the framework contains three action classes: Collect, Navigate and Store. While the first and the second are ready to use, application providers must override the method makePersistent of Store to store the data persistently. Implementors are responsible to keep the storage system in a consistent state (e.g. by using transactions). The framework contains an implementation of the Store action accessing databases via JDBC. Additional action classes can be implemented and used in the specification. Such classes must implement the interface Action, which declares one method called upon dispatching an action.

The path through the views is driven by the specification and the user data. At any time users can undo actions and change the data. In some cases this can lead to a new path and parts of the data collected may become obsolete. The class WizardHistory keeps track of the course of the application. Basically it is a linked list of the visited views. The Collect action updates this object. To access this object from the views a set of tags is defined. These enable views to present at any time a list of links to all previously visited views in the actual path and to manoeuvre in this path. The framework contains a wide range of rule classes (e.g. DiscreteValueRule, NumberRule, DateRule), range classes and many helper classes (e.g. implementing tags).

6. THE VIEWS

The code generator generates a JSP-page for each view tag in the file wiz.xml. These pages contain a form-element with subelements for each field. In case the range of a field contains only finitely many values, the subelement is presented as a menu, as checkboxes or as radio buttons depending on the size of the range and the value of the attribute multiple. Otherwise, normal text fields are used. In case the validation fails, the same view is presented again. The states of the fields remains unchanged, i.e. text entered and options selected are still present. In addition the error messages are shown. This functionality is provided by the form-tag included in the framework. In the generated HTML-forms, the attribute action references the name of the current view. The run-time system delegates all request to views to the controller, providing the name of the view as a parameter.

Furthermore each page contains links to all predecessors views, an undo and a forward button. This functionality is also provided by dedicated tags. All Wizard tags only access information from the model or the History-object, they do not contain any layout information.

Listing 5: Excerpts of a generated JSP-page for a view collecting data about credit cards

Listing 5 shows a section of a generated JSP-page demonstrating the usage of the wizard-tags. To distinguish these tags from ordinary HTML-Tags, they have the prefix wiz. Figure 3 shows the resulting HTML-page which collects data about a credit card. The range of credit card type, expiration year resp. month are finite, hence they are displayed as menus. At the right of the first field there is an error message, indicating that the credit card number is missing. At the bottom there are links to all views in the current path. They can be used to check or change data already entered. Designers can start from scratch or use the generated pages to build the real pages using any editor supporting JSP-Tags or using a template technique.

7. THE IMPLEMENTATION

The Wizard framework is completely implemented in Java and utilizes Java Servlets for the controller and Java Server Pages for the views. It can be used with any Web-server sup-
8. RELATED WORK AND CONCLUSION

Work on conceptual modeling of e-commerce applications is naturally a relatively new field. It builds partially on the work on hypermedia applications such as the Object-Oriented Hypermedia Design Method (OOHDM) [10] and the Relationship Management Methodology (RMM) [6]. These are mainly restricted to the presentation of content (from files, databases or other resources) and focus on the navigational design. They do not consider data entry and access to external applications. Currently there is a growing number of projects providing a solution for the creation of web applications within a well defined software production process (ADM [7], OOWS [8]). In most cases extensions of formal specification models such as UML or ER-diagrams are introduced to describe the navigational and conceptual design. These projects have a relatively broad focus and the possibilities for automatic code generation are in most cases rudimentary. In contrast the Wizard framework focuses on data entry applications and has a strong support for the automatic generation of the application. Web Modeling Language (WebML) is a notation for visually specifying complex Web sites at the conceptual level including data entry and operation units [2]. WebML is also based on XML and allows automatic code generation. Compared to the Wizard framework it is a general modeling language missing specific modeling primitives of data-entry applications such as validation, ranges and menu generation. A project with similar goals but using a completely different approach is FAR, which is an end user language for programming web services [4]. FAR (Formulas and Rules) allows users to offer and deliver e-services without having knowledge of the underlying software and protocols. The FAR prototype is built on top of E-speak, a product that provides the infrastructure for e-commerce applications. The Telemachus tool developed in the ADM project also allows rapid prototyping of web applications [7]. But instead of producing a default layout the tool tries to produce presentation templates by analyzing sample pages and based on these templates all web pages of the application are generated. They do not consider to automate data entry tasks.

The main contributions of the Wizard framework are: a method for a high level specification of data entry tasks in form of an XML-document which can be validated against a DTD, a code generation environment that allows automated prototyping for testing and requirements analysis and finally a clean separation of the tasks for the different developer groups, enabling human resources to work in their trained field independently from each other.

Current research is focusing on how to automate the testing of the generated applications. In addition the views are extended by a second layer. The original views no longer generate HTML but rather XML without any layout information. The additional layer transforms the XML-views into the final device and language dependent form by utilizing XSLT stylesheets.

9. REFERENCES
