Do-It-Yourself Database-Driven Web Applications

Keith Kowalzcykowski
app2you, Inc.
keith@app2you.com

Kian Win Ong
CSE Dept.
UC San Diego
kianwin@ucsd.edu

Kevin Keliang Zhao
CSE Dept.
UC San Diego
kezhao@cs.ucsd.edu

Alin Deutsch
CSE Dept.
UC San Diego
deutsch@cs.ucsd.edu

Yannis Papakonstantinou
CSE Dept.
UC San Diego
yannis@cs.ucsd.edu

Michalis Petropoulos
CSE Dept.
SUNY Buffalo
mpetropo@cse.buffalo.edu

1. INTRODUCTION

UCSD's app2you project [15] (commercialized as app2you.com) and its successor FORWARD project [11][1] belong to the emerging space of *Do-It-Yourself (DIY), custom, hosted, database-driven web application platforms* that empower non-programmer business process *owners* to rapidly and cheaply create and evolve applications customized to their organizations' data and process needs. The hoped-for outcome of DIY platforms is paralleled to the emergence of spreadsheets in the 80s and of graphical presentation tools in the 90s [1]. Before the arrival of tools such as powerpoint, polished presentations had to be prepared by graphics professionals. PowerPoint enabled us to do them ourselves.

Generally DIY application platforms provide an *application design facility* (also called *application specification mechanism*) where the application owner (also called process *owner* and business architect [20] specifies the application by manipulating visible aspects of it or by setting configuration options. A simple early example of DIY creation was form builders, where the owner introduces form elements in a form page and the platform, in response, creates a corresponding database schema.

A DIY platform must maximize the following two metrics: First, how wide is its *application scope*, that is, what computation, collaboration on a process, and pages (presentation) can be achieved by applications specified using the platform's design facility? Second, how *easy* is the specification of an application using the platform's design facilities? When the ease increases the technical sophistication required by the owner decreases, and non-programmer process owners are increasingly enabled. The two metrics present an inherent tradeoff. At the one extreme, building applications using Java, Ajax and SQL provides unlimited scope, but does not provide ease of specification. Platforms such as Ruby on Rails [17] and WebML [3] make specification easier and faster, but still not easy enough to enable non-programmer owners. At the other extreme, creating an application by copying an application template, as done for example in Ning [13], is very easy but the scope of the platform is limited to Ning's finite number of templates. DIY platforms are between these two extremes of the scope/ease trade-off (see Section 4 for a discussion of particular platforms).

This paper focuses on *human-centric* [20] *database-driven web applications*, i.e., applications whose

- entire state is captured by the application's database as opposed to also having out-of-database state that is accessed by the application by interfacing to corresponding external systems.
- the state changes (and correspondingly the business process progresses) in response to user actions on the web pages. We do not consider the possibility that, say, an automated data feed produces data asynchronously and without user request.

Furthermore the paper makes only brief mention of the customization of the front-end's visual and interface aspects.

In an app2you/FORWARD application users with potentially different roles and rights interact on a web-based process. Depending on the state of the process/application, each user has rights to access certain pages, read certain records in them and execute requests, which often pertain to reported records.

The presented *general framework* captures a broad scope of workflows and applications but achieving its full scope requires knowledge of SQL. The *limited framework* is the subset that can be generated by the DIY design facility and is the focus of the paper. Its essential limitation is that the core queries are SPJ queries, including the possibility of EXISTS conditions in the WHERE clause. An overlay of aggregation and calculation, which captures Excel functionality, can also be added on the core limited framework, without disrupting the design facility.

The limited framework presents an excellent scope/ease-of-specification tradeoff point: The DIY design facility, which is tuned to the limited framework, enables the easy specification of applications by "business architects" [20], that is, application owners that are not programmers but have the sophistication to reduce their business process into web pages by specifying, in a WYSIWYG fashion and in response to easy-to-understand prompts, properties of the pages such as who can access each page, what is the page's main function, what happens in response to an action.[3]

[1] Supported by UCSD's von Liebig Center for the commercialization of technology and NSF OCI 0721400 is collaboration with SDSC.

[3] An advantage of supporting both a general and a limited framework is that even when the non-programmer business process owner cannot create or fully customize the entire web application (typically because parts of the application require the functionality of the general framework) app2you still enables a much more efficient collaboration

Submit Startup

submit

Startup Name: Facebook

Logo: Browse...

Business Plan:

Social networking and entertainment!

Founders

Name	Title

add more "Founders"

Figure 1 `Submit Startup` Page

Evaluate Startups

Startup Name	Logo	Business Plan	Founders		Reviews		Solicitations	Advisor Comments			Demo Grades
			Name	Title	Notes		Route to	Comments			Score
Facebook	facebook	Social networking and entertainment!	Mark Zuckerberg	CEO	I found my friend from high school!	solicit	larry@google.com	Very targeted demographics for advertising!	remove	Invite	
			Dustin Moskovitz	Programmer			bill@microsoft.com	We are buying it!	remove		
									submit		
							add more "Solicitations"				
YouTube	YouTube Broadcast Yourself	Videos on the Internet!	Steve Chen	CTO		solicit	larry@google.com		remove	Invite	10
			Chad Hurley	CEO					submit		
							add more "Solicitations"				

Figure 2 `Evaluate Startups` Page

Advisor Comments

Startup Name	Logo	Business Plan	Advisor Comments	
			Comments	
Facebook	facebook	Social networking and entertainment!	Very targeted demographics for advertising!	submit
YouTube	YouTube Broadcast Yourself	Videos on the Internet!		submit

Figure 3 `Advisor Comments` Page

Let us first convey informally the scope of limited scope applications through a redacted version of a real-world app2you application. More real world examples are found in the Appendix. We use a simplified and modified version of the app2you application for TechCrunch50 (TC50) 2008 [19], a conference where ~1000 startups submitted requests, along with information packages, in order to present themselves and their products. The app2you application was used to collect the submissions, review them, schedule multiple rounds of appointments where the candidates met the reviewers online to demo their products, and select the top 50 startups. At page `Submit Startup` (Figure 1) any user with a registered account can prepare and submit information regarding her startup, which includes the name, logo, and list of founders.[4] Every user is constrained to at most one submission. The submitted startups are displayed on the `Evaluate Startups` page (Figure 2), which is accessible by all reviewers, each of whom can execute three *requests* on each one of them: submit a review consisting of `Notes` for each startup; solicit comments from one or more advisors (essentially external reviewers), in which case the startup submission will be displayed on the `Advisor Comments` page (Figure 3) to the particular advisors;[5] invite the startup submitter to schedule an interview. The invitation results to the candidate receiving an email notifying him to visit the `Schedule Appointment` page, which reports available interview slots, submitted by the reviewers on the `Post Interview Slots` page, and lets the invited startups choose one of them (see Figure 4). The submitted choices are reported on the `Grade Demo` page, where the reviewers post their grade for the demo given at the agreed interview slot as part of the second review round. Finally, the submitted demo reviews are reported on the `Evaluate Startups` page, where reviewers can now make an informed decision of which 50 startups are the most promising ones. The actual application evolved during a period of two months, indicating the great value of the Do-It-Yourself approach in allowing applications to evolve as the business process evolves.

Do-It-Yourself Design Facilities of app2you & FORWARD The general goals of the app2you and FORWARD Do-It-Yourself design facility are typical in easy-to-use systems: (i) WYSIWYG design, where the owner immediately experiences the result of each specification action. (ii) Wizards that suggest to the owner common and semantically meaningful specification options and automate their implementation. (iii) Wizards that explain the specification at a high level where the user does not have to engage in schema design or database queries. Satisfying such ease-of-use specification goals has required the introduction of multiple novel DIY specification techniques, which are briefly described in this paper, and the construction of algorithms that automatically infer the schemas and queries that fuel the pages.

Many of the reported techniques of this paper have been developed by UCSD's app2you project and consequently were used by app2you.com users, while other techniques (see below) are currently under development in FORWARD and resolve fundamental problems faced by owners in their efforts to build applications, as observed by real world experience with app2you.com.

Due to the highly interactive, WYSIWYG nature of the DIY design facility we suggest that the reader watches the 10-minute high resolution video at http://www.vimeo.com/2075363, password app2you.

between the business process owner and IT specialists: The non-programmer owner creates himself the bulk of the application's pages and workflow, which corresponds to the limited framework, while the IT and specialists provide assistance in elaborate graphics, integration with outside services, code for complex functions, complex SQL and other such aspects that belong to the general framework. The business architect can then build the rest of the process.

[4] Users must first pass from a typical login and signup page before they reach the page of Figure 1.

[5] In the actual application there were solicitations to other reviewers.

The first technique, already used extensively by app2you users, is *page-driven design* (Section 3.2), which provides to the owner a WYSIWYG model of the pages. The owner specifies properties of the pages that have immediately visible effects on the page. For this contribution we borrowed techniques from the WYSIWYG/automatic design of database schemas by the creation of respective input forms as done in form-builders even before the web. We expanded with the WYSIWYG specification of forms and requests that operate within the *context of reported records* (for example, every input form, such as `solicit` action and `invite` action on `Evaluate Startups`, operates within the context of a submitted startup). We also launched wizards for specifying properties by answering questions, expressed in easy-to-understand language referring to the pages and the requests that happen on them. app2you in response creates automatically the pages' structure (called *page sketches*, Section 2) and the underlying schemas and queries, therefore relieving the business process owner from designing the database layer, which is one level away from the layers that she understands, namely, the application page layer and the overall workflow. Hiding database schemas, queries, constraints and other low-level details is facilitated by an architecture where high level, easy-to-explain derived properties of the page sketch hide hard-to-understand complex primitive properties (Section 3.1).

By observing the users' design efforts we found out that the inherent difficulty (in comparison to, say, a spreadsheet) in producing a WYSIWYG model for a collaborative application is that the pages typically behave differently depending on which user accesses them and the application state. The resulting enhancement to page-driven design (Section 3.2) allows the owner to experience the page's function as if she were a suggested sample user of it. The FORWARD project pushes this concept further by prompting the user to assume the role of such sample user and to perform particular suggested actions that reveal properties of the pages' operation that would otherwise not exhibit themselves. Forcing the use of the application also leads to collecting sample data, which become useful in exhibiting the operation of other pages also.

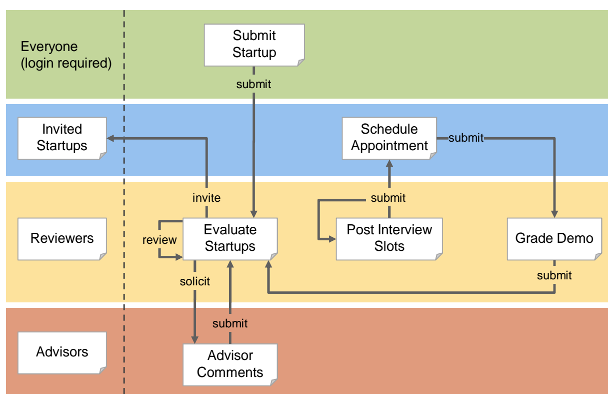

Figure 4 TechCrunch50 Workflow Visualization (in ppt; see video demo for the actual one)

Page-driven design by itself still turns out to be insufficient for allowing the owner to reduce a non-trivial multistep process she has in mind into a working application. In order to appreciate the difficulty that non-programmer owners face, visualize a database-driven application as a workflow. Figure 4 shows the workflow visualization of the functionality of the TC50 application. *User groups* are on the left. The rows in Figure 4 visualize *access rights*, that is, which pages are accessible by which user group. Intuitively, it is easy for the owner to specify in a single specification action what appears as a single transition in the workflow graph, such as the `solicit` edge. Unfortunately, a major shortcoming of DIY online databases, which is not resolved by page-driven design alone, is that they require the owner to decompose a single user action in the process into coordinated activity in two pages. For example, in page `Evaluate Startups` the user submits the solicited advisor's name. Page `Advisor Comments` has a too-complex-for-non-programmers query that filters startup submissions according to whether the currently logged-in user appears in a solicitation related to this startup. The FORWARD technique that will resolve this problem is the *workflow-driven design* extension (Section 3.3) to the page-driven paradigm.

Finally, in Section 3.4, we discuss work-in-progress on a semiautomatic creation of reports. The goal is a report building interface that

- suggests semantically meaningful joins of various data sets; or joins of the currently reported data with other collected data sets of the application
- does not suggest joins that would lead to provably redundant information on the report
- explains to the owner the (potentially nested) involved data sets and joins by referring to names that appear in the application; avoids causing confusion with details of how the pages are normalized in tables
- requests minimal information in the form of plain multiple choice menus
- discovers the best placement of information on the report in order to illustrate associations and constraints between the reported data sets.

In effect the interface must compensate for the minimality of the owner-provided information with algorithms that detect and perform complex nested report creation operations.

Section 2 presents the part of the general framework that pertains to database-driven applications and the limitations of the limited framework. It argues why both the general and the limited scopes can capture many practical applications. Section 3 briefly describes an array of design techniques. Section 4 discusses related work.

2. FRAMEWORK AND SCOPE

An app2you application is described by its *application sketch,* which is defined by the *general app2you framework*. The general app2you framework, which is used in the rest of the paper, captures purely database-driven applications, i.e., it ignores interfacing with external services and systems, which is functionality captured by the under development general framework in the FORWARD project.

The sketch is modified by the owner when the application is in design mode. The sketch consists of *primitive properties* (collectively called *primitive sketch*) and *derived properties,* where the former are more low level (e.g., queries, constraints) and their settings cannot be derived by the settings of other properties. For ease of specification the non-programmer owner typically does not access the primitive sketch aspects directly, since deconstructing a process into primitive aspects tends to require CS sophistication. Rather the non-programmer owner indirectly accesses them via the *derived properties*, which explain at a high level common questions and options, using wizards and other components of the DIY design facility (Section 3).

The primitive sketch consists of *page sketches*, *user group definitions,* a *database schema* and *general* properties, such as the application name and path.

Each page sketch has a URL, a *page context*, which captures the request parameters (and the types of their values) that are expected upon requesting this page, and a top-level *unit*.

A unit generally has *fields*, a *mode*, *requests* and one *visual template* for each mode. *Atomic fields* generally display data of corresponding parameters of the context.

1.1 Reports

Iterator fields are important for the generation of reports. They have a query, which is typically parameterized by the context c and retrieves from the database tuples $t_1,..., t_n$ that have schema t and correspond to the records displayed by the iterator. The iterator has its own unit, which contains the displayed fields of the retrieved tuples. Such unit operates within context $c+t$, which is the concatenation of c (i.e., the context within which the iterator operates) and t (i.e., the context that the iterator generates). The unit of an iterator field may recursively contain its own *nested iterators*.

For example, the top-level unit of the `Evaluate Startups` page (Figure 2) has an iterator field, whose unit contains the atomic fields `Startup Name`, `Logo`, and `Business Plan`. This iterator runs a query `SELECT * FROM Submit_Startup`, where `Submit_Startup` is the automatically inferred table that collects the non-nested fields of the startup submission form (see Figure 1). It also contains the (nested) iterator field `Founders`, whose unit, in turn, contains the atomic fields `Name` and `Title`. The query of the iterator `Founders` is `SELECT * FROM Founders WHERE Founders.Parent=?` and the parameter (`?`) is instantiated by the `Submit_Startup.ID` of the query result of the containing iterator.

The general framework allows iterator queries to be arbitrary SQL queries over the schema, typically parameterized by values of the context. In this way SQL experts can utilize SQL's full power. The limited framework queries (*lqueries)* are SPJ queries with `EXISTS` predicates), that is, queries of the form `SELECT * FROM` *OuterJoinExpression* `WHERE` *BooleanCondition*, where the condition may be parameterized with values from the context and may also involve `EXISTS(`*SubQuery*`)` predicates where the parameterized *SubQuery* is recursively an lquery. Applications of the limited framework use only lqueries and corresponding constraints, which are generated by the DIY facility. A DIY-built application however may go outside the limited framework and into the general framework by selectively utilizing "manually" written queries and constraints for a few complex functionalities.

App2you will soon also allow *calculated fields* to be associated with queries. In the limited framework such queries will capture the typical functionality of Excel spreadsheets. In particular, a calculated field may:

1. Compute a new "scalar" value from values of the context. For example, if the context has attributes `First Name` and `Last Name` then the calculated field `Name` may be calculated as `concat(Last Name, ",", First Name)`.
2. Compute an aggregate value by applying an aggregate function over a (potentially hidden) nested iterator of the page. For example, the non-programmer owner may include in `Evaluate Startups` a calculated field `Number of Founders` that performs the count function over the `Founders` iterator fields.
3. Combinations of the two above.

Lqueries, scalar calculations and aggregates capture the needs of most typical reporting applications and even the needs of relatively unusual request-controlling constraints, such as "each startup may receive at most 5 advisor reviews". Therefore the limitation leads to small scope loss. At the same time, this limitation enables ease of specification benefits: First, the design facility automates the creation of reports (see Section 3.4) for lqueries. Second, filtering and aggregation uses DIY interfaces that have proven themselves in other settings (e.g, spreadsheets). Third and most important, lqueries enable the easy and efficient computation of the context created by each report tuple, therefore enabling the automatic inference of the database commands associated with contextual requests, such as the `submit`, `invite` and `solicit`, i.e., requests that appear in the context of reported data, as explained in Section 1.2.

Note that for DIY simplicity the design facility focuses on pages with a single iterator at the top level unit of the page. Such pages are called report pages.

1.2 Contextual requests

A unit may also contain zero or more *requests*. Intuitively, a request combines an HTML input form with information on the effects of submitting the form. In particular, a request contains (i) zero or more *input fields* (ii) a mechanism of submitting the form (e.g. a button) (iii) a *constraint*, represented by a yes/no query (see discussion below on representation of queries), whose semantics is that the request is applicable only when the constraint is satisfied, and (iv) a list of effects.

The most common effect of executing a request is an update on the database; this will be the only effect discussed in detail next. In the general framework such effect is captured by an SQL statement, which is possibly parameterized by the context. In the limited framework the database effect is automatically inferred by the DIY design facility: It is an insertion in the database of the values collected by the input fields. It is described in the sketch by (i) naming the database table that takes the insertion and (ii) mapping the input fields to type-compatible attributes of the table. If the form contains repeated nested forms, such as the `Founders` in the `Submit Startup` form that contains `Name` and `Title` pairs, then each nested form is mapped to a corresponding database table. Note that the inserted record also includes *system attributes* such as the auto-generated `ID`, the `submitter` and `creation timestamp` of the record.

Other effects of a request may be (i) sending an email, described by a template (in the style of MS Word mail merge) whose placeholders can refer to both the input fields of the form and the system attributes and (ii) causing a navigation to another page, which can be used to produce confirmation pages and forms submission processes that span multiple pages.

For example, the data submission form of Figure 1 is a request. Its effect is inserting the collected data in tables `Submit_Startup` and `Founders` and sending a confirmation email. It has the constraint that the currently logged-in user has not submitted a startup already. The `solicit` and `invite` of Figure 2 are the buttons of respective requests.

A feature that sets the scope of app2you applications apart from the scope of online databases (see Section 4) is the ability of reports to have nested requests, which operate in the context of the reports. For example, the `solicit` request in Figure 2 operates within the context created by the containing report iterator. Such a nested request is said to be an *annotation* of its report iterator. A nested request differs from a top level request as follows: First, when it inserts in the database it may map values from its context into attributes of the insertion table. For example, when the

solicit request is executed it stores a tuple in a table Solicitation and this tuple has a foreign key attribute that stores the ID of the startup submission within whose context the particular nested request operates. Second, its constraint and its side effects may also utilize the context. For example, the invite action is associated with a constraint that there may be at most one invitation for each startup.

Note the following important interplay between lqueries and the automatic inference of the insertions happening when a request is issued: lqueries enable automatic inference of a compact context for the nested requests that appear within reports fueled by such lqueries. In particular, each record produced by a lquery creates a context consisting of the IDs of the few database tuples that joined together to result in it. This, in turn, enables fully automatic inference of an efficient database insertion performed when the nested request is activated. In particular, the insertion stores the IDs of the compact context along with the input fields of the request and the system attributes. This, in turn, leads to ease of specification since the non-programmer owner does not have to specify what part of the context of a nested request will be stored with the insertion.

Note that the DIY design facility is facilitated by *iterator+request field combos* where the iterator part of the combo ranges over requests created in response to the request part of the combo. For example, the iterator+request field Advisor Comments in Figure 3 combines the submit request with an iterator showing the comments collected by the submit request.

1.3 User group definitions

In the limited framework *user groups* (such as Invited Applicants, Advisors and Reviewers) are identified as a pair consisting of a report page and a field (of such report) whose values are user identities. The submitter is typically such a field.

1.4 Visual Templates

The visual template of a unit uses placeholders [10] that refer to its fields. During runtime, such placeholders are replaced by actual values. App2you provides a list of built-in visual templates that are automatically revised during design time to capture changes on the structure of the page, the forms and the reports. For example, when a new field is added on the report, the visual template of the report is automatically revised to display the new field. Due to space limitations we will not discuss visual templates in further detail.

3. DO-IT-YOURSELF DESIGN FACILITY

We focus on three key DIY-enabling techniques of the design facility and the architecture that enables them: page-driven design (Section 3.2), workflow-driven design (in progress, Section 3.3) and automatic creation of complex reports (in progress, Section 3.4). We use the following principles as a scorecard for the DIY design facility.

- Prefer to provide concrete explanations of sketch properties using WYSIWYG feedback and verbalization of prompts and options that refers to pages, requests and other highly visible properties of the page; rather than being abstract and making references to database terms.
- Prefer to provide a high-level specification from which primitive properties can be generated, rather than a low-level specification of primitive properties that requires the owner to deconstruct high level concepts into low level concepts.
- Prefer to summarize and enumerate design options to focus on common cases, rather than provide an unstructured, high degree of freedom. "Advanced user", less prominent interfaces should cater to the less common cases.

3.1 Derived Properties

Often an important combination of primitive properties must be explained to a non-programmer owner at a high level, which is close to the non-programmer's understanding of the workflow and the function of the pages. Therefore the *derived properties interface* reads the primitive sketch and exports *derived properties* and corresponding common options (called *derived* options) for their settings. When the owner chooses an option the derived properties interface translates it back to the primitive sketch. We describe next a simple example of a derived property, exemplifying the concept. Derived properties become paramount in the following sections.

For example, recall that a user of the Submit Startup page may submit only one startup. Once she makes her submission, the form of Figure 1 disappears. At the primitive sketch level, this behavior is achieved by a non-obvious primitive property: The constraint associated with the form checks that the set of startup submissions of the currently logged-in user is empty. Understanding the behavior of the Submit form at this level is fairly complex. Therefore the page wizard offers a derived property asking the much more obvious question of Figure 5.

The combination of a primitive sketch with a derived properties interface produces many benefits on scope and ease of specification:

- It enables the incremental addition of derived properties in the platform, as common cases that lend themselves to higher level explanations emerge, without disrupting existing applications. Indeed, applications created before the introduction of a new derived aspect in the platform can benefit from its introduction: The derived properties interface reads their primitive sketch and exposes a high level derived property.
- It enables a 90/10 rule where the design facility first poses common questions, often relying on derived properties and derived options in order to express them. At the same time, the wide scope enabled by the primitive sketch is available.

3.2 Page-Driven Design

The first step towards providing a high-level specification is to allow the process owner to design her application through the WYSIWYG model of pages, as opposed to engaging in low-level web and database programming. Various properties of pages are either specified by direct visualization on the pages, or via answering simple questions about the page. The design facility in response automatically creates the page's form/request and iterator structure, underlying schemas and queries.

Through the high-level specification, page-driven design relieves the owner from specifying data structures in the abstract while en route to construct pages. Moreover, explaining the design options available at the page level promotes easy comprehension, especially if they are explained directly in terms of the application layer that are easily perceived by the owner such as what is the report/form structure of the pages. Lastly, page-driven design facilitates immediate feedback on whether a design satisfies the owner's requirements, since the owner can both inspect and experience the page directly.

3.2.1 *Page Wizard*

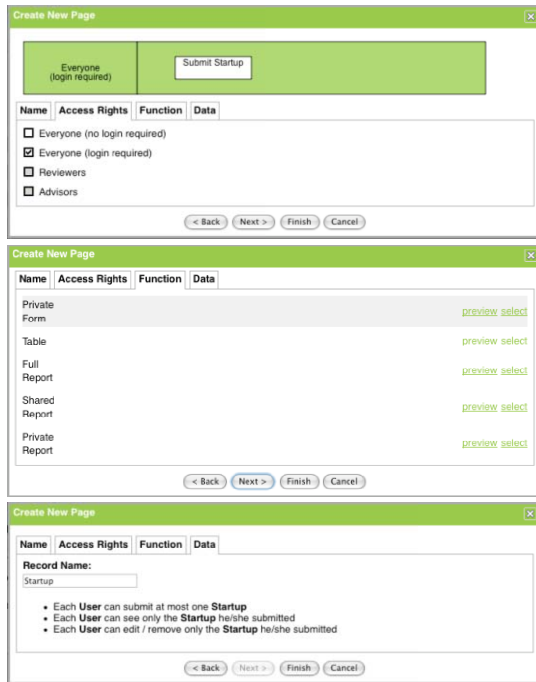

Figure 5 Page Wizard for `Submit Startup` Page

The page wizard is the starting point of page-driven design. It prompts with simple questions about page-specific information, such as the page name, URL, and the groups that are authorized to access the page. For example, access to `Submit Startup` is granted to system-defined group `Everyone (no login required)` (Figure 5), whereas access to `Evaluate Startups` is granted to custom group `Reviewers`. Allowing a page to be accessed by a group is also visualized on the workflow diagram by placing the page in the appropriate swim-lane (row).

The page wizard prompts for the main function of the page by enumerating a list of templates, where each template bundles a commonly occurring combination of page properties including presentation format and action rights. Templates are provided to speed up the design of common cases. Such common cases may include forms that allow each user to submit at most one record, and tabular reports where each user sees all records but can only edit/remove the records she submitted, etc. Where the common case is not fully applicable to the scenario at hand, the owner can always customize the page by overriding individual properties independently.

Figure 5 shows the `Private Form` template used in creating the `Submit Startup` page. The template provides the following defaults for the following derived properties:

- The `submit` property of the page's form is set to `on, but max one per user`. (Each applicant can only submit one startup.)
- The `display` property of the page's iterator is set to `on if user has submitted the record, off otherwise`. (Each applicant can only see the startup info she has submitted.)
- The `edit` and `remove` properties of the page's iterator are also set to `on if user has submitted the record, off otherwise`. (Each applicant can only edit or remove the startup info she has submitted)

Whenever the `submit` aspect is `on`, the page wizard also prompts the owner to optionally assign a name to the records collected. The record name helps the system phrase questions and options more specifically. Figure 5 shows the wizard for `Submit Startup`. It starts with a system-proposed default of `Record submitted at Submit Startup`, which is later set by the owner to `Startup`.

3.2.2 *WYSIWYG Design*

After the basic properties of the page have been specified through the page wizard, the owner can customize the form of the page in a WYSIWYG fashion. To create new input fields, the owner drags-and-drop input components such as text boxes, image upload prompts, dropdown boxes, check boxes etc. into the request form of the page (Figure 6).

For each input component dragged into the form, a corresponding field is added to the request, and a corresponding attribute is added to the schema of the database table where the records corresponding to the request are inserted. The input component determines the data type of the field. For example, the `Logo` field is created through an image upload component, therefore storage is allocated for binary data, data can be submitted through an HTML file input form element, and submitted data are displayed as images.

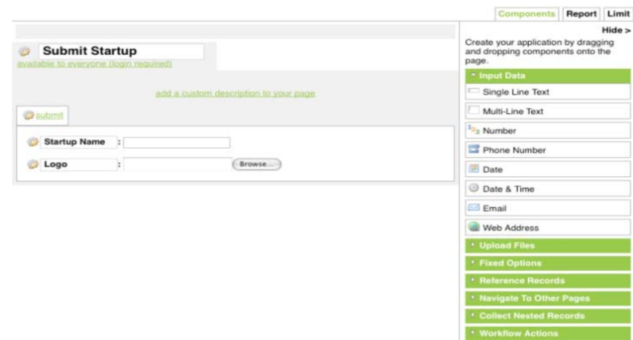

Figure 6 WYSIWYG Page Design

The owner may also introduce repeating nested data by creating nested tables, such as the `Founders` on the `Submit Startup` page.

3.2.3 *DIY creation of nested requests*

While existing form builders and online databases employ WYSIWYG design for (pure) input forms, app2you advances page-driven design to also encompass the WYSIWYG specification of nested requests that operate within the context of reported records. On the `Evaluate Startups` page for example, the `Reviews`, the `solicit` and `invite` nested requests all operate within the context of a startup. A nested request is also called *annotation* by the DIY design facility if it is a `submit` requests, which is associated with input forms, since the data it collects intuitively annotate the data of the report.

An annotation is created by the drag-and-drop of an input component into a report.[6] For example, the `Reviews` annotation is created when a multi-line textbox for `Notes` is dropped into an area corresponding to a startup, excluding the area corresponding to `Founders`. In this way the owner visually specifies the context of the `Reviews` annotation to be a startup. Had she accidentally

[6] As we will see in Section 3.3.1, more generally, nested requests are created by the introduction of a workflow action (such as the `solicit`, `invite`) into a report.

dropped it into the area for Founders, she would have seen a multi-line text box for each founder, which creates an immediate visual indication of the mistake. Recall that the design facility automatically infers the database insertions that will be issued when a review is submitted. For example, the insertion of a review will lead to inserting in the underlying table Reviews a record that contains the values collected by the input fields, system attributes and a foreign key that refers to the startup that provides the context for the particular review. The updates issued when a review is edited are computed similarly.

3.2.4 Experiencing the page

WYSIWYG design is not sufficient since there are properties that are not immediately evident from the page's visual appearance. For example, how many submissions can a user make? Can a user see which other users have submitted?

The inherent difficulty faced by an owner of a collaborative application (as compared to an owner of a spreadsheet) in comprehending the behavior of an application and verifying it against her requirements, is that pages typically behave differently depending on what data has been submitted and who accesses the data. The design facility takes a number of steps towards resolving this problem. First, it makes every feature that is available in use mode also available during design mode. The fact that the page sketches are interpreted, instead of requiring a design-compile cycle, facilitates this. Second, it always prompts the owner to submit sample data and make requests so that corresponding records can be shown on report pages. The third step is to prompt and help the owner assume the role of particular sample users in order to visualize the behavior of properties that would otherwise be hidden.

The system suggests to the owner to experience a page as a sample user if it recognizes that certain properties of the page cannot be explained by the owner's current WYSIWYG experience. For example in Submit Startup, the system suggests the experience submit as a sample user in order to explain to the owner the following properties:

- The display property of the page is set to on.[7] The owner understands this when she sees that the startup info record submitted by the sample user is displayed on the page.
- The submit property of the page's request form is set on, but max one per user. The owner understands this when she sees that the request form and button disappears once she submits a startup info record.
- The edit and remove properties of the page's iterator are set to on.[8]

Note however, that the experience of the first sample user does not fully explain whether the display, edit and remove properties are unconditionally or conditionally on. For example, does the iterator display all records submitted, or only records submitted by the current user? Therefore, the design facility subsequently engages the owner to experience as a second sample user. The experience shows that in this the page, each user can only see, edit and remove records she has submitted. If this is contrary to requirements, the owner can then either select another template, or customize the individual properties defaulted by the template.

When the records displayed by iterators and the requests that are available are controlled by complex conditions, it is harder to reason about what sample data and sample users are needed in order to experience a page. For example, obtaining the experience of a solicited advisor at the Advisor Comments page requires that (i) at least one (sample) solicitation has been made and (ii) the owner uses the Advisor Comments page as if she were the solicited advisor. When the conditions have been introduced in response to workflow-driven design, as described next, it is easier to reason about such sample users and data.

Note that in practice sample data are not needed when the first pages of the application have actually gone in use and have already obtained actual data.

3.3 Workflow-Driven Design

In the workflow visualization of an application (see Figure 4), which is under design and development in the FORWARD project, edges (also called transitions) capture requests that happen on the page at the source of the edge and affect the experience and rights of a user on the page at the target of the edge. The starting points of a workflow are data collection pages, such as Submit Startup and Post Appointment Slots that provide requests collecting new records without implicit or explicit references to other records. The records may be reported on the data collection page itself, or appear on reports that combine data collected from one or more pages. Reports, such as Evaluate Startups, may allow their user to act on individual reported records (review, solicit or invite). Formally, there is an edge from page P_1 to page P_2 labeled with request a_1 if executing a_1 on P_1 may change

1. the *read rights* of a user u on P_2, that is, u can read on P_2 a record r as a result of a_1. For example, the submit edge from page Submit Startup, accessible to Everyone (login required), to page Evaluate Startups, accessible to reviewers, denotes that reviewers gain read rights to a startup once the request is submitted.
2. the *action rights* of a user u on P_2, that is, u can perform a request a_2 on P_2 as a result of a_1. For example, the solicit edge indicates that upon executing the solicit on Evaluate Startups a user (in this case the solicited advisor) can read and comment on a startup submission at the Advisor Comments page.
3. the *access rights* of a user u on P_2, that is, u gains access on page P_2. For example, the invite edge of Figure 4 indicates that upon executing the invite request on Evaluate Startups a user (in this case the startup submitter) gains access to the Schedule Appointment page.

An implementation that visualizes the workflow also allows drilling down into the nature of the edges so that the owner can tell which type of right is affected by the edge, why it is affected, etc.

Some workflow transitions correspond to application functionality that is easily built using page-driven design. For example, the submit edge from Submit Startup happens because the owner ordered at the page wizard that the Evaluate Startups reports the data collected on Submit Startup.

However, process owners often want to capture more elaborate workflow logic, which leads to application functionality that cannot be easily-built in page-driven design. Consider in Figure 4 the solicit edge from page Evaluate Startups to page

[7] The display aspect of a page is a derived aspect that asks whether a page that has a form also has a report iterators that displays the data submitted at the form.

[8] The edit and remove aspects of a page are derived aspects that ask whether the report iterator of the page provides the built-in actions edit and remove.

`Advisor Comments`, accessible to advisors. Here, the reviewers may solicit reviews for each startup from a subset of the advisors. Using page-driven design, the owner has to add an annotation (request) to `Evaluate Startups` so that reviewers can choose the advisors to solicit reviews from. Then she needs to create the `Advisor Comments` page, for the advisors to submit their comments, by initially report all the startups from the `Evaluate Startups` page, and then keep only those where the currently logged-in user is one of the advisors chosen to solicit a review from; not a simple condition to state regardless of how friendly the query building GUI is. Indeed, the query in SQL is:

```
SELECT *
FROM Submit_Startup
WHERE EXISTS ( SELECT *
               FROM Solicitations
               WHERE SS_ref = Submit_Startup.ID
               AND route_to=<current user>    )
```

The `Solicitations` table folds the advisors chosen (`route_to` column) by the reviewers for each `Startup` (`SS_ref` column). The `SS_ref` column is a foreign key referring to the `ID` of a `Startup`. The `route_to` column is a foreign key referring to the `ID` of an advisor and the condition makes sure that the solicited advisor is the currently logged-in user. No matter how user friendly the query building of the design facility becomes, the above query is too hard to be conceived by a non-programmer.

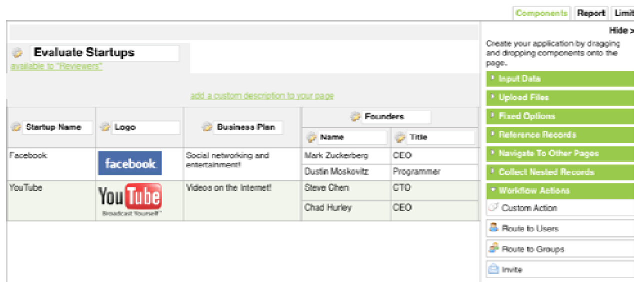

Figure 7 Workflow-Driven Design

Deconstructing a single workflow transition, which corresponds to a single user request, into the above design steps is not a trivial task for the process owner. For that reason, FORWARD will enhance the design facility's page-driven design with *workflow-driven design* where all of the above design steps are integrated into a single DIY task performed on the starting page of a workflow transition.

Workflow-driven design is initially experienced by the owner as a set of `Workflow Actions` components, shown on the right side of Figure 7, which can be dragged-and-dropped on a page as any other component. For the `solicit` example, the process owner decides to drop the `Route to Users` component on the `Evaluate Startups` page, which triggers the `Create New Action` wizard. The wizard saves the owner from having to formulate queries like the one above.

3.3.1 Workflow Wizard

The requirement for the workflow wizard is to either automatically infer or ask the process owner about one or more of the following properties of the workflow action (request) and correspondingly of the transition that appears in the workflow visualization:

1. The action on the current page that corresponds to the workflow transition.
2. The type of record involved in the workflow action, which can typically be automatically inferred from the context in which the request was introduced
3. The user group that will be affected by the action.
4. How exactly the action will affect the rights of the user group on the corresponding records. That is, will the action change the access rights, the read rights and/or the action rights of the affected user group on the target page. In most cases this is implied by the choice of the action and no additional information is needed.
5. Depending on the answer to the above question, additional questions about the exact implementation of the rights become relevant. For example, if the workflow action makes the record readable by users of the affected group, which is the page where the users will read the record?
6. How the affected user group will be notified of the action?

As an example, let us consider what the workflow wizard for the `Route to Users` action should do, in the spirit of the above properties, while the owner customizes the request to solicit comments from advisors.

Property 1 is answered purely by the fact that the owner drags-and-drops the `Route to Users` action from the `Components` list (see Figure 7) into the page. Property 2 is inferred by the fact that the owner dropped the workflow action in `Evaluate Plans` page; therefore the type of record involved in the workflow action is a `Startup` record. The wizard can proceed in a series of questions. Property 3 comes from asking the owner to decide which user group to route `Startup` records. The answer in the running example is `Advisors`. Property 4 is implied by the choice of the `Route to Users` action, whose effect is that the involved record (`Startup` record) becomes readable by users of the chosen group (`Advisors`). Property 5 comes from asking the owner which is the page where `Advisors` will read `Startup` records; the owner will answer that is a new page, named `Advisor Comments`. Property 6 is addressed by a last question, where the owner chooses to send an email to the relevant users of the `Advisors` group.

Once the owner exits the wizard, the system automatically places a `solicit` request in the context of each `Startup`, along with a drop-down box that references the advisors, as shown in Figure 2.

3.4 Automated Report Creation

Section 3.2 has demonstrated how the high-level specification of pages can generate a database schema, while Section 3.3 has shown how raising the specification level to that of application workflows can ease the design of requests. In keeping with this high level of specification, it is desirable for owners to design reports powered by complex queries, without having to specify low-level implementation details of queries such as projections, join conditions and selection conditions.

Since report pages are created after data collection pages the automated report creation can leverage semantic information previously specified by the owner. Consequently, the design facility is able to provide the owner a minimal interface for designing complex reports, while compensating for this minimalism with algorithms that offer semantically meaningful options and automate implementation details.

Let us consider how the owner can extend the `Evaluate Startups` page with the comments that advisors have submitted on `Advisor Comments`. Such an augmented `Evaluate Startups` is shown in Figure 8b.

Figure 8a shows the WYSIWYG design of `Evaluate Startups`, during which the owner selects the `Report` tab and sees options for extending the page. For example, the first option corresponds to extending `Evaluate Startups` with data on `Advisor`

`Comments`. The intuitive understanding is that selecting an option will cause the system to produce a more complex report, which is an amalgamation of both pages.

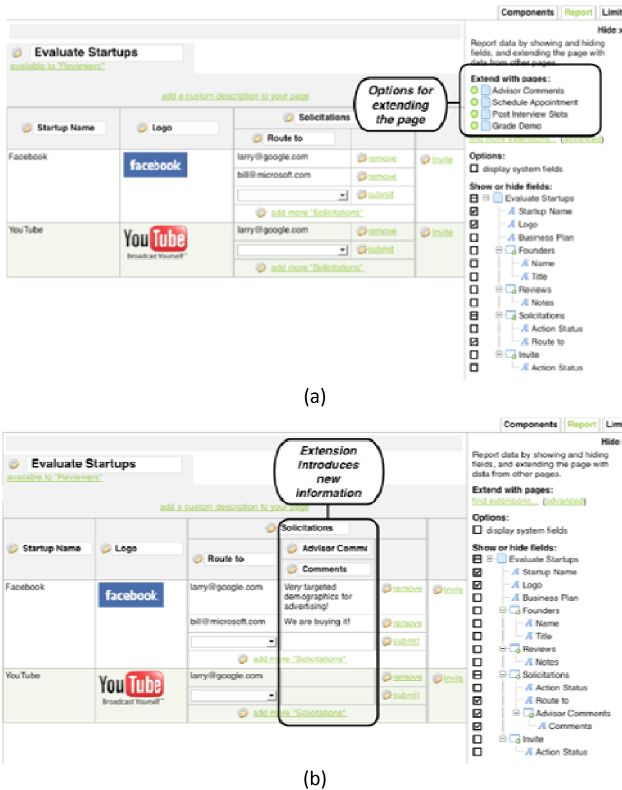

Figure 8 Automated Report Extension on `Evaluate Startups` Page

Figure 8b shows that after the above-mentioned option is selected, the system has introduced advisor comments by extending the `Solicitations` iterator's unit with data collected by `Advisor Comments`. Notice that the extension was placed at an optimal point, next to corresponding solicitations.

Through the WYSIWYG interface (and appropriate sample data), the owner receives immediate visual feedback of the extension. She can then perform further customization, such as hiding extraneous fields and iterators, deleting the extension and starting over, or repeat the design activity of extending the page.

This minimal interface is intended to capture the common case of designing reports. Sophisticated owners may choose to obtain explanatory details for an option in order to customize join conditions.

To enable this high degree of ease for the owner, the system has employed various DIY features and heuristics. The technical challenge lies in intelligently restraining the infinite space of all possible joins, to produce a summarized enumeration of options for the common case.

3.4.1 Generating Joins

When the owner selects the `Report` tab, the system produces the list of options by first generating a (finite) list of join paths. This is the core mechanism by which ultimately the owner chooses from enumerated options, rather than specify join conditions using arbitrarily complex Boolean conditions.

For each pair comprising an iterator b of the base page, (that is, the report page to be extended - `Evaluate Startups` in the example) and an iterator or a request e of any extension page, app2you attempts to find join paths that connect b and e. A join path is a left-deep relational join of the form:

$$FC(b) \ldots \bowtie_{c_n} FC(i_n) \bowtie_{c_{n+1}} \ldots FC(e)$$

where $FC(i)$ is the *flat context* of an iterator i.

Some example join paths between base iterators on the `Evaluate Startups` page and extension iterators on the `Advisor Comments` page are the following. For the sake of example, assume the `Advisor Comments` page also shows the `Founders`.

1. *FC(Evaluate_Startups)*

   $\bowtie$*[lhs.startup_id = rhs.startup_ref]*
   FC(Founders)

2. *FC(Evaluate_Startups)*

   $\bowtie$*[lhs.startup_id = rhs.startup_ref]*
   FC(Advisor_Comments)

3. *FC(Solicitation)*

   $\bowtie$*[lhs.startup_id = rhs.startup_ref*
   AND lhs.route_to = rhs.submitted_by]
   FC(Advisor_Comments)

The flat context of an iterator i is its corresponding non-parameterized query. If i is the top-level iterator of the page, then $FC(i)$ is simply the query producing the records displayed in i. If i is nested within iterator h, then $FC(i)$ is $FC(h)$ appropriately joined with the query producing the records displayed in i.

The join conditions c_n are conjunctions of equalities between attributes. Currently, the system considers two types of attribute join-pairs: (1) between id attributes, and corresponding foreign key attributes (2) between email attributes corresponding to user groups, and `Submitted By` / `Edited By` attributes of records accessible by said user groups. This reflects the common intuition where the majority of join conditions involve unique identifiers, be they surrogate keys generated by the database or natural keys such as email addresses.

Note that the generated join paths do not contain cycles (i.e. an iterator can only occur once in the path), otherwise there can be an infinite number of paths. The exception is that b and e can be the same iterator, so that the owner can make arbitrary self-joins by choosing the same e for subsequent extension rounds.

3.4.2 Detecting Redundant Joins

The list of join paths generated is finite, but not all join paths are useful enough to present as options to the owner. For each join path, app2you makes a hypothetical extension of the base iterator, and uses view equivalency to test whether the extension adds only redundant information on the page.

For example, join path 1 is provably redundant and can be discarded, since there is already a `Founders` iterator on the base page `Evaluate Startups`.

A conservative definition of redundancy is the following: A join path is redundant if it leads to a new iterator x, where there is already an iterator y such that for all possible database instances that satisfy the schema and its constraints, each tuple $t_x=(v_1, \ldots, v_n)$ in $FC(x)$ has a corresponding tuple t_y in $FC(y)$ that has $v_1, \ldots, v_n$ and vice versa. We currently investigate additional definitions of redundancy.

Note that such a definition does not prevent self-joins or, more generally, reports where a database table occurs multiple times as a result of different join conditions. The redundancy test is accomplished by essentially reducing all constraints into

embedded dependencies, asserting the existence of t_x in *FC(x)* and running a chase procedure (similar to [15]) that deduces tuples that must exist in the flat contexts of other iterators on the page.

3.4.3 Optimizing Join Placement

Given two generated join paths where the extension iterators are the same, one join path may be strictly better than the other. For example, contrast Join path 2 with 3. Extending `Evaluate Startups` with 2 will place advisor comments on each startup, whereas 3 will place advisor comments on each solicitation. Intuitively, 3 is preferable to 2, as only the former visualizes the existing association between a solicitation to a specific advisor, and the corresponding comment.

This intuition can be expressed as functional dependencies between records. A startup can be routed at most once to each advisor, and an advisor can comment at most once on each startup. Therefore, a comment functionally determines a solicitation, which in turn functionally determines a startup. Since app2you relies heavily on WYSIWYG visualization to assist the owner in making design choices, it is important that wherever possible, functional dependencies and other constraints in the schema be visualized with the appropriate placement and nesting of iterators. Extending with 3 will produce a more accurate visualization of the functional dependencies.

Implementation-wise, running the chase procedure in Section 3.4.2 has the side benefit of also producing the necessary functional dependencies.

3.4.4 Bundling Additional Joins

After discarding pruned join paths, the surviving ones are aggregated by the pages of the extension iterators, and presented as a list of options as in Figure 8a. This achieves the minimal interface with a corresponding high-level of specification, as the owner only needs to comprehend pages (and not join paths) to start creating complex reports.

Note that the system uses the page rather than the iterator as the level of summarization. This comes from the observation that due to the parameterization between nested iterators, the standalone functionality of an iterator is harder to perceive than that of a page. Moreover, the existence of a report page is a strong hint that its structural organization is useful. Therefore, bringing in the entirety of the page en masse as part of the extension and allowing the owner to later hide extraneous fields and iterators provides better visual cues, than allowing the owner to extend one iterator at a time.

For an example, consider an alternate scenario where startups can provide rebuttals to advisor comments. There will be a page `Rebut Advisor Comments`, that reports comments and annotates them with a `Rebuttal` iterator. If `Evaluate Startups` were not extended with `Advisor Comments`, but were instead extended with `Rebut Advisor Comments`, the bundling of additional joins will introduce both comments and rebuttals with a single round of extension.

3.4.5 Visualizing Projections

Iterators and fields can be easily shown and hidden with checkboxes (Figure 8b). For example, each iterator has a few hidden-by-default system fields, such as `Submit Timestamp`. The owner can easily customize the new `Advisor Comments` iterator to display when each advisor submitted her comment. From the DIY perspective, it is far preferable for the owner to toggle visibility of iterators and fields through an enumerated list, than to manually specify a projection list of attributes (a la query languages such as SQL).

4. RELATED WORK

The time is opportune for Do-It-Yourself database-driven applications for two reasons. First, they leverage the emergence of hosted applications (software as a service) and Web 2.0 Ajax-based interfaces that allow application page design from the comfort of one's browser, while providing the richness of desktop interfaces. The two aspects combine to remove the hassles of (i) downloading/installing software in order to create an application and (ii) deploying/exporting an application on the web. But the Do-It-Yourself ability presents a larger, qualitatively-different challenge: How to disrupt conventional database-driven web application programming by providing brand new models of specifying database-driven web applications so that non-programmer business owners can build their own applications.

Multiple systems support the fast creation of custom web applications by removing the need to program in a complex Turing-complete programming language, such as Java. WebML [3] is a prime example of *schema-driven application creation* (also see DeClarit [6], Oracle Express [14]). The creator starts by designing the Entity-Relationship data model for her application. Then it is easy to specify pages by putting together units that accomplish typical functionalities of Web applications. For example, a unit may report the data of an entity and utilize the relationships of the data model to navigate to related entities. It is reported [22] that the development and maintenance of WebML applications led to 30% increased productivity with 46 distinct applications maintained by 5 part-time, junior developers.

The emerging Do-It-Yourself custom application platforms primarily target non-programmer process owners. A common theme is that the owner does not need to create a database schema in the abstract. Rather she builds forms, which automatically lead to corresponding tables that are typically reported on the same page. Such systems tend to be *online databases* [4][5][7][9] for easy information sharing and collaboration, often delivering great advantages over online spreadsheets, which are their main competitor for structured information sharing[9]. However, the resulting applications have a very limited scope (and business logic): Users simply post and read structured data in the shared space.

A next generation of Do-It-Yourself systems promises to go beyond information sharing and to enable users to capture their business processes by web applications. At a high level, these enablers are either "MS Access online" [4][2] or customizable vertical templates [18]**.**

The "MS Access online" enablers allow users to create multiple Do-It-Yourself online tables (having forms and reports to give access to them). In the same spirit with MS Access, the reports have to be fueled by queries where the user has explicitly specified joins and selections. Finally, business logic and flow of data from table to table is offered in the form of scripting programming languages [12] or graphical languages [4] that allow the user to describe series of insertions, deletions and updates and the conditions under which they should happen. The adherence to tables with separate forms and reports creates problems at both the scope axis and the easy specification: The web applications we are dealing with day-to-day are not mere collections of tables with a report and a form for each table. A typical case is that the input forms of a page typically operate within the context of reported dynamic data and even within the context that prior pages create,

[9] Yahoo Pipes [21] and IBM's QEDWiki [8] represent high end versions of the information sharing space, where data from multiple sources and RSS feeds can be automatically integrated and presented online.

i.e., there is no artificial divide of "input only" and "report only", as is clearly evidenced by pages such as `Evaluate Startups` and `Advisor Comments`. In addition to app2you, AppForge [2] also solves this problem by allowing input forms in the context of reports.

Another scope problem of "MS Access Online" is the inability to capture that access rights to a page may depend on the business logic itself. For example, in the TC50 application the group "Invited Applicants" is derived automatically and controls access to "Schedule an Appointment".

The "MS Access online" class is problematic in creating workflow application since the business process owner needs to reduce the collaborative process she has in her mind into normalized tables and into sophisticated queries and updates. For example, we showed in Section 3.3 how hard it is to explain using a query that the `Advisor Comments` should only show startup submissions that have been passed to the currently logged-in user. This raises the bar of sophistication needed by the builders towards the level of sophistication that programmers have, therefore seriously limiting who can create and evolve applications. The anecdotal evidence behind this thesis is plenty: Instructors of undergraduate database classes know the difficulty that, even computer science students, have in designing appropriate schemas and writing non-trivial queries. Furthermore, despite the best efforts of tools, such as the tools of the Microsoft Office Access and Microsoft InfoPath, to make database schema design and query writing approachable by the masses, the general public has found it hard to engage in those activities. The above evidence is not surprising since database schemas and queries are abstract structures that have no immediately visible connection to the web application and workflow aspects that the non-sophisticated designer can immediately associate with, which are the Web pages with which the users of the application will be interacting.

Applications with fixed workflow and database table structure and customizable input form structure (i.e., one can change the attributes of tables as long as the tables and their interactions remain fixed) have been a great success [18]. We believe that customization does not need to stop at that point since, by doing so, the scope of available applications is limited by the available initial templates.

5. REFERENCES

[1] Jeanette Borzo: Do-It-Yourself Software, Wall Street Journal, 9/24/2007. http://online.wsj.com/article/SB119023041951932741.html#articleTabs%3Darticle.

[2] Chavdar Botev, Nitin Gupta, Jayavel Shanmugasundaram, Fan Yang: A WYSIWYG Development Platform for Data Driven Web Applications. VLDB 2008.

[3] Stefano Ceri, Piero Fraternali, Aldo Bongio: Web Modeling Language (WebML): a modeling language for designing Web sites. Computer Networks 33(1-6): 137-157 (2000).

[4] Coghead. http://www.coghead.com

[5] DabbleDB. http://www.dabbledb.com

[6] DeKlarit. http://www.deklarit.com

[7] eUnifyDB. http://www.eunifydb.net

[8] IBM QEDWiki. http://services.alphaworks.ibm.com/qedwiki/

[9] Intuit Quickbase. http://www.quickbase.com

[10] JavaServer Pages Technology http://java.sun.com/products/jsp/index.jsp

[11] K.W. Ong, Y. Papakonstantinou, K.K Zhao: Do-It-Yourself Forms-Driven & Workflow Database-Driven Applications. Provisional patent submitted by University of California at San Diego, December 2008.

[12] LongJump. http://longjump.com

[13] Ning. http://www.ning.com

[14] Oracle Application Express. http://www.oracle.com/technology/products/database/application_express/index.html

[15] Y. Papakonstantinou, I. Katsis, K. Ong: Creating Hosted Web Application and database. Utility patent submitted by University of California at San Diego, April 2007

[16] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, Val Tannen: A Chase Too Far? SIGMOD Conference 2000: 273-284.

[17] Ruby on Rails. http://www.rubyonrails.org/

[18] Salesforce.com. http://www.salesforce.com

[19] TechCrunch50 (TC50). http://techcrunch50.com

[20] Colin Teubner and Ken Vollmer: BPMS Revenue To Reach $6.3 Billion By 2011. Forrester Research, 2007. http://db.ucsd.edu/app2you/2009-www/2007-forrester-bpms.pdf

[21] Yahoo! Pipes. http://pipes.yahoo.com/pipes/

[22] Piero Fraternali, Stefano Ceri, Massimo Tisi: Developing eBusiness solutions with a Model Driven Approach. IMP 2006.

Appendix

More than twenty forms-driven applications have been built and used in 2008 on app2you.com. For example, a recruiter has collected job openings from its customers. A wide group of users, defined and controlled by the recruiter, sees selected fields of the job openings' records and is invited to recommend individuals, who are notified about the positions, provide their level of interest and proceed to exchange information with the customer and the recruiter if interested.

In another example, the United Cerebral Palsy non-profit organization maintains an online loan library of toys, keeping track of who currently holds a toy and who has requested it.

In multiple variations of classroom management applications students submit their projects, often after a phase where they have teamed up in project teams. The TAs and instructor provide feedback and grade. Variations include setting up appointments for project presentations and rehearsals, voting for the best project etc.

In multiple variations of reviewing applications, candidates submit material that is pushed thru a review process with various rules and steps.