6-9-2016

Uniform: The Form Validation Language

Sawyer Novak
Santa Clara University

Reid Palmquist
Santa Clara University

Douglas Parker
Santa Clara University

SANTA CLARA UNIVERSITY
Department of Computer Engineering

I HERBY RECOMMEND THAT THE THESIS PREPARED
UNDER MY SUPERVISON BY

Sawyer Novak, Douglas Parker, Reid Palmquist

ENTITLED

UNIFORM: THE FORM VALIDATION LANGUAGE

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE
IN
COMPUTER SCIENCE AND ENGINEERING

6/9/16
Advisor Date

N. Ling 6/9/16
Chairman of Department Date

Thesis

Senior Design

Santa Clara University

Santa Clara, California

Uniform: The Form Validation Language

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science and Engineering
Santa Clara University
School of Engineering

Authors:
Sawyer Novak
Reid Palmquist
Douglas Parker

June 9, 2016

Abstract

Digital forms are becoming increasingly more prevalent but the ease of creation is not. Web Forms are difficult to produce and validate. This design project seeks to simplify this process. This project is comprised of two parts: a logical programming language (Uniform) and a web application.

Uniform is a language that allows its users to define logical relationships between web elements and apply simple rules to individual inputs to both validate the form and manipulate its components depending on user input. Uniform provides an extra layer of abstraction to complex coding.

The web app implements Uniform to provide business-level programmers with an interface to build and manage forms. Users will create form templates, manage form instances, and cooperatively complete forms through the web app.

Uniform's development is ongoing, it will receive continued support and is available as open-source. The web application is software owned and maintained by HP Inc. which will be developed further before going to market.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Digital Forms

Forms are a vital component of data transactions. Modern businesses and organizations rely on quick and efficient data flow and forms have long been the primary means by which data is acquired. As data has increasingly shifted to a digital format, growing exponentially in volume, the need for consistent, accurate, and accessible forms has become critical, to the point that there are jobs dedicated entirely to the creation and management of forms. Increasing reliance on automation means that incomplete or incorrect forms, which require manual effort to correct, significantly impact timeframes and quality.

1.1.1 Digital Forms Require Programming Skills

Prior to the digital revolution, forms were, universally, a physical medium, but the advent of the Internet has increasingly pushed forms into a digital medium in order to accommodate for the growing scope and volume of the data businesses and organizations seek to capture. At the same time, however, businesses have struggled to adjust and adapt to the unique problems posed by digital forms. Core amongst these issues is that a physical form would typically be produced by a group of businessmen and women who had a clear understanding of that form‘s purpose and the business needs it would serve. Unfortunately, these same men and women do not, by and large, possess the necessary skills and knowledge to develop digital forms. The insertion of a skilled programmer into the form development process has led to increased cost, lengthened timelines, and an even greater disconnect between the designers of a form and the resulting product.

1.1.2 Logically Simple but Hard to Implement

Moreover, although logically simple, online forms can be surprisingly hard to implement effectively. This is because the tools that support web development are not particularly suited to form development. Currently, most digital forms have their functionality implemented two or three times. There is one implementation on the server, validating request data. A second implementation exists on the client, validating the form when the user presses the “submit” button. Often there is yet one more implementation on the client using an event system to update the form dynamically for the user. These three components can end up using drastically different architectures and languages, but still perform the same *logical* operation, creating a large amount of duplicate code that cannot be easily maintained. The problem is that forms are logical entities which can better be defined by the states of - and relationships between - their underlying components, something that current web-technologies can only address obliquely. Thus, form development has become a messy, inefficient process that forces developers into utilizing toolsets that were designed and developed to solve different problems.

1.1.3 We Need a Solution

Increasingly, the underlying issues of digital form development are being acknowledged, but it is difficult to say what can be done to answer them directly. The structure of the web necessitates that digital forms continue to be implemented using existing technologies and so long as this is true there will still be a need for skilled developers. But this does not mean that a solution is impossible. Recognizing the constraints placed on the form development process, the answer is not to directly attack the source of the problems, but rather to build around it. Form development can be brought back into the hands of business users by providing an abstraction of current web based technologies that eliminates the need for in-depth programming knowledge and experience with web development.

1.2 Emerging Solutions

Currently, a number of platforms have already been established in an attempt to return form development to the hands of business users. These solutions have attempted to solve this problem in a number of different ways and range in their success. This includes Form.com, DocuSign, SurveyMonkey and Google Forms. However, there are three distinct areas where current solutions are still lacking to varying degrees:

- Form logic does not have the depth which many forms require
- Forms lack effective help for confused users
- Forms under-serve mobile use cases

1.2.1 Problem 1: Limited Functionality

The first major problem is that these services are not flexible enough for users to create the forms their businesses need. One of the biggest advantages of digital forms is that they can modify and check themselves automatically. If someone checks a box saying he/she is a U.S. citizen, then any questions for foreigners should be automatically hidden. This simple logic means the form can automatically restructure itself to only ask the questions that are necessary. This minimizes the amount of work done by the user, reduces the chances for an error, and improves the overall experience.

Logic like this is somewhat supported in existing solutions; however, all of them favor accessibility over functionality, reducing features provided to make the interface simpler. DocuSign avoids logic support altogether. Google Forms and SurveyMonkey only allow forms to redirect to different pages depending on the selection of a radio group. Form.com is the only site which has any meaningful logic support, but it has a complicated interface as a result and still does not provide the freedom business users need.

1.2.2 Problem 2: Inadequate Interactive Help

The second issue is that forms do not have any way to effectively help confused users. Forms can become quite complicated, and often the user filling them out is not completely sure how to answer. On physical forms, there is usually someone there who can help. There is a person across the desk who understands the form and can cut through the nonsense and filler to direct the customer to the parts that matter. This person is usually experienced with the form and knows how to handle any edge cases. In a digital environment, this person is not present and it is up to the user to simply “figure it out."

No existing solution has addressed this problem in particular. At most, these solutions can offer technical support if their service malfunctions. The company that built the form may host a support email address or call center if they are large enough, but these are no more helpful. This support has no knowledge of the forms they are supporting. They can help a user who is unable to check a box, but they are useless to someone who needs to know whether they *should* check a box.

1.2.3 Problem 3: Lack of Mobile Support

The third difficulty with digital forms is that they often lack or under-serve mobile use cases. In the modern era, mobile devices are becoming more and more prevalent and these devices need to be supported for businesses to keep up.

Most of these solutions will work in a mobile solution, but only Form.com has prioritized mobile devices in any meaningful way. Form.com directly supports mobile inspections as well as offline mobile submission, but the other solutions simply render the desktop version in a smaller and more minimalist form. This is hardly sufficient, and does not provide the user experience that is expected of these sites.

These three problems are not new, but existing solutions have failed to address them and users are demanding better than what they currently provide.

Our proposed solution has two components, the first being the Uniform Validation Language.

1.3 Uniform Validation Language

1.3.1 Why Uniform?

As stated previously, part of the reason forms are so difficult to implement is due to using programming tools to solve a logical problem. Since no current tool exists, Uniform is being created as the logical tool for this logical problem. It is not difficult to create a basic form; however, if one were to add just a few simple logical rules, the complexity quickly becomes overwhelming. The amount of work does not correlate with the simplicity of the intended action. In a Uniform program, however, statements are written in a one-to-one relationship between the programmer's thoughts and the code they write. Uniform is the medium between logical statements and computer code.

1.3.2 Uniform's Capabilities

The Uniform language allows users to easily set validation rules for forms and validate them. Additionally, the language will allow a means to define relationships between form controls and allow the form to change dynamically based on both user input and the status of other controls. The language will also be executable server-side to allow the same piece of Uniform code to validate both the client-side HTML form and the server-side request data without requiring a customized solution for either.

1.3.3 Powerful, Yet Easy to Learn

Previous solutions give business users an easy to understand interface, but at the expense of logical complexity. Uniform is powerful, yet easy to understand and learn. Ideally, Uniform allows a business level user to control their forms at the skill level of an advanced programmer. Compared to current implementations, Uniform is an elegant and clean solution that abstracts complicated code into a familiar language.

1.4 Web Application

The second component of our solution is a Web Application which utilizes the Uniform language to create and issue forms. The Web Application is the interface and the driver that fully showcases the variety and complexity of forms. In conjunction with the Uniform language, the Web Application makes it easy to create, administrate, and view web forms.

1.4.1 Web Application's Capabilities

This system allows business-level administrators to create digital form templates with a simple drag-and-drop interface and set validation rules written in the Uniform language. Associates, company representatives, can then create new instances of forms from the templates provided by the administrators and distribute them to clients. Clients are then presented with an interface that allows them to easily fill out their form, with the interactive guidance of an associate. This application provides a means of communication between company associates and the customers in real time as they fill out the form together.

1.4.2 Mobility Addressed

Mobility and accessibility are two key features of this software, addressing the concern regarding the lack of mobile support. Uniform works cross-platform on desktop and mobile devices. Forms are relevant and frequent in our lives, and so being able to access form content even from mobile devices is invaluable in todayâĂŹs day and age.

With these three problems addressed, form development can be put back in the hands of business users and enable them to create the forms their clients deserve.

2 Document Layout

This project involved, first, the creation of a logical language, and second, a practical implementation of this language in the form of a web application addressing these issues. Because these are distinct components, this document will be split into two major parts: the "Logical Language" followed by the "Web Application".

Part I
Logical Language

3 The Logical Solution

The goal of the Uniform logic language is accurate, efficient, and accessible digital form validation. This solves "Problem 1: Limited Functionality" in section 1.2.1.

The Uniform language addresses this problem by providing a simple framework which allows forms logic to be built quickly and easily.

4 Design Philosophy

Uniform was built on the following principles:

- Uniform code should follow natural spoken language
  - Think it, don't program it
- Uniform should be accessible
  - Easy to learn and understand, even for non-programmers
- Allow for smarter forms

  - Simple in design, but powerful and versatile

5 Requirements

The Uniform language design can be broken down into functional requirements which state what the system is capable of while the non-functional requirements state the manner in which the functional requirements will achieve their goals. (In order of the first being the most important)

5.1 Functional Requirements

The language will...

- allow form creators to set validation rules for individual form controls.
- allow form creators to define relationships between multiple controls.
- provide a mechanism to validate forms.
- be executable on the server to aid in server-side validation.
- identify the causes of validation failure to the user.

5.2 Non-Functional Requirements

The language will...

- apply validation rules on an HTML DOM page (when executed on the client).
- operate at a logical level of abstraction, rather than a programmatic level.
- be easy to use.
- be well-documented.
- be easy to learn, even for non-programmers.
- run efficiently.
- be scalable and reusable.

5.3 Design Constraints

The language will...

- be compatible with most modern browsers.
- be compatible with modern NodeJS engines.

6 Why Uniform?

6.1 The Problem: Car Form

To better illustrate how the Uniform language is utilized a basic example is below. The following Form asks the user to answer two questions:

1. Do you have a car?
2. If so, what is the make, model, and year?

The following HTML markup constructs a web form for the user to fill out their answers, producing the form seen in fig. 1.

```
<form id="myForm">
  <input id="chkCar" type="checkbox">I have a car</input>
  <div id="frmCar">
    Make:  <input id="make"  type="text" /><br />
    Model: <input id="model" type="text" /><br />
    Year:  <input id="year"  type="text" /><br />
  </div>
  <button type="submit">Submit</button>
</form>
```

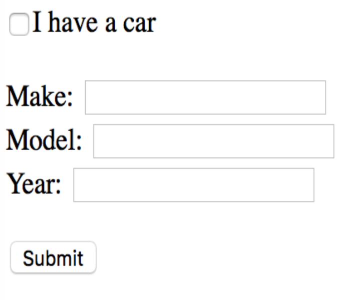


Figure 1: HTML Example

This form can be built quickly and with only a basic understanding of web technologies. The difficult part is the form logic.

6.1.1 Intended Form Logic

If the user does not have a car, then the following question, asking the make, model, and year, are irrelevant. As the form builder, I would like the form to adjust depending on the user's input, hiding or disabling the following three fields. So I would think to myself to add this rule:

- If the check box is not checked, disable the make, model, and year textboxes.

As this rule is spoken aloud, it sounds very simple, but even something as fundamental as this is surprisingly difficult to do with current solutions. Below is the code necessary to enforce this behavior using an existing solution: JavaScript (including validation for each individual field):

```
$("#chkCar").on("click", function (evt) {
    if (evt.target.checked) {
        $("#frmCar").removeClass("ui-disabled");
    } else {
        $("#frmCar").addClass("ui-disabled");
    }
}

$("#frmCar").on("submit", function (evt) {
    var success = true;
    if ($("#chkCar").checked) {
        try {
            if ($("#make").value === ""
                || $("#model").value === ""
                || $("#year").value === "") {
                    success = false;
            }
            if (parseInt($("#year").value) <= 0) {
                success = false;
            }
        } catch (err) {
            // Handle parseInt failure
            success = false;
        }
    }

    if (!success) {
        // Something failed, don't submit form
        evt.preventDefault();
    }
});
```

While this code does provide the desired functionality, it is surprisingly bulky for a relatively simple piece of logic.

6.1.2 Problems

Evident from this example are several problems with the current means of applying logic to forms.

- JavaScript can be difficult for non-programmers to understand; even this example requires two lambda functions, two event listeners, a try-catch, and five if-statements. It is much too complicated to even describe.
- The simplicity of the rule is not reflected in the relative complexity of the code.
- The logic is separated: part of it is in the form's `submit` event and part is in the checkbox's `change` event.
- It is necessary to implement the same logic on both the client-side and the server-side. There is likely another set of code probably written in a different language which must be maintained separately.

These problems stem from using the wrong tool. Developers are trying to use a programming language to solve a logical problem.

6.2 The Solution: The Uniform Language

6.2.1 The Car Problem Revisited

The Uniform code below provides identical functionality as the JavaScript code written in section 6.1.1 but in contrast to the former, it provides additional benefits.

6.2.2 index.html

```
<script src="/lib/uniform.js"></script>
<script>uniform.options.href("/carForm.ufm");</script>
```

6.2.3 carForm.ufm

```
$("#carForm") {
    valid:
        $("#subForm") is valid or
        $("#checkBox") equals false;
}

$("#subForm") {
    valid:
        $("#make")  is valid and
        $("#model") is valid and
        $("#year")  is valid;
    enabled:
        $("#checkBox") equals true;
}
```

Using Uniform, it quickly becomes apparent that applying the desired logical functionality to the form is significantly simpler, both in terms of readability and the amount of code required.

The Uniform code above:

- Is easy to read
- Is easy to learn
- Is self-documenting
- Allows the user to create logical statements that directly translate into executable code
- Allows the form to easily adjust dynamically to user input
- Minimizes code overhead
- Eliminates the need to understand complex JavaScript

The code can be included in an HTML document with a simple script tag, much like jQuery or any other JavaScript library. Often requiring less than half the lines of code to solve the same problem, it is easy for any entry level programmer to understand what the Uniform language is doing. Even if you are not a programmer, read the code above aloud, and its meaning should be straightforward.

7 Uniform Code Structure

7.1 Blocks

The most fundamental Uniform file is as structured:

```
<selector> {
    <tag>: <statement>;
}
```

This piece of code is called a "block." A block determines the attributes of a particular Document Object Model element.

Below is a non-abstracted example using the code structure above:

```
$("#myElement") {
    valid: 1+3 equals 4;
}
```

This code states the following:

- `myElement` is valid if the following condition is true: 1+3 equals 4

Since 1+3 does indeed equal 4, the condition is evaluated to `true`, and so the valid tag of `myElement` is set to `true`.

7.2 Selectors

Selectors are a jQuery notation for selecting a particular Document Object Model element. Uniform borrows the same notation eliminating any language specific convention confusion. For more information on selectors, visit the jQuery documentation at:

https://api.jquery.com/category/selectors/

7.3 Tags

The Uniform language uses tags to define logic for each form element. As shown in the example above, the valid tag determines if a sub-form is valid. The list below enumerates each keyword and how it affects the form element depending on the conditions inside. Uniform supports the following tags:

- Valid
  - Will determine the sub-form as valid if the valid conditions are true
- Enabled
  - Will disable a sub-form if the enabled conditions are false
- Visible
  - Will hide the sub-form if the visible conditions are false

7.4 Statements

A statement in Uniform must be a boolean expression. This is to say an expression that evaluates to either true or false. These statements are used to validate or invalidate a particular tag which is attached to the block selector. More information on how a statement is constructed will be explained in a later section.

7.5 Advanced Features

7.5.1 Regular Expressions

Regular expressions are used to select a specific set of characters. In a web based environment, they can be used to verify a particular format that information is entered. For example one could use a regular expression to verify that the user's email is valid based on the placement of the `@` symbol and the suffix (.com).

Uniform regular expression notation is identical to the normal convention. To learn more about regular expressions, visit this link:

`http://www.regular-expressions.info/refquick.html`.

The code below shows an example of how regular expressions can be used to verify that a textbox field is not empty.

```
$("#carForm") {
    valid: all $("input") matches /"."/;
}
```

The Uniform notation requires the use of the keyword `matches` to compare a string with a regular expression. A regular expression is defined by wrapping the regular expression in `/" "/`. This syntax was chosen because of a desire to match JavaScript conventions where appropriate. In JavaScript, regular expressions are denoted with `/ /`, which was the intended syntax for Uniform. However this can lead to ambiguity since the slash character can already be used to denote a line comment `//`, block comment `/* */`, and division `(a / b)`. As such, the `/" "/` syntax was chosen to more clearly differentiate it from other tokens beginning with the slash character and make development easier and reduce time spent on minor issues. It is very likely that sometime in the future this syntax will be updated to align with the JavaScript syntax.

7.5.2 Variables

The Uniform language supports the use of variables. Variables allow coders to create a label for a complex expression or a piece of code that is written frequently.

The code below shows an example of how variables can be effectively used.

```
$("#carForm") {
    valid:
        $("#subForm") is valid or
        $("#checkBox") equals false;
}

$("#subForm") {
    valid:
        $("#make") is valid and
        $("#model") is valid and
        $("#year") is valid;
    enabled:
        $("#checkCar") equals true;
}

```

```
// Variable declarations
@filled: /"."/;
@integer: /"^[0-9]+$"/;
@fourDigitInteger: /"[0-9]{4}"/;

$("#make") {
    valid: $("#make") matches @filled;
}

$("#model") {
    valid: $("#model") matches @integer;
}

$("#year") {
    valid:
        $("#year") matches @fourDigitInteger and
        value > 1900;
}
```

Regular expressions are used frequently and thus can be time consuming and irritating to repeat. Putting them in a variable increases the readability of the code.

7.5.3 Applying Rules to Multiple Elements

The Uniform language uses Document Object Model (DOM) Manipulation to access and define the relationships between elements. Rules and relations can be applied across tags, classes, and nodes through nesting. For example, make, model, and year are all "input" tags within the sub form. A user could create a rule which applies to these forms with minimal trouble using the `any` or `all` keywords preceding the selector.

```
// Applies to #make, #model, and #year
@16CharLimit: /.{1,16}/;
$("#carForm") {
    valid:
        all $(".field") matches @16CharLimit;
}
```

8 Examples in the Real World

To see full implementations of the Uniform language see Appendices B and C located at the end of the Thesis.

9 Architectural Design

From a high-level perspective, the Uniform language will perform two functions to accomplish its task.

1. Parse the plain text uniform rules file (.ufm) into event listeners that update the state of the selectors of the HTML web markup.

2. Validate the HTML form based on the values of the HTML that Uniform embeds in the document.

Figure 2 shows the overall architecture of the language processing flow. A plain text rules file (.ufm) is fed into the parser which interprets it and produces a set of events and handlers capable of enforcing the rules given dynamically based on user input. A jQuery plugin was written to allow direct access to this logic and allows users to check the validity of any element on the page.

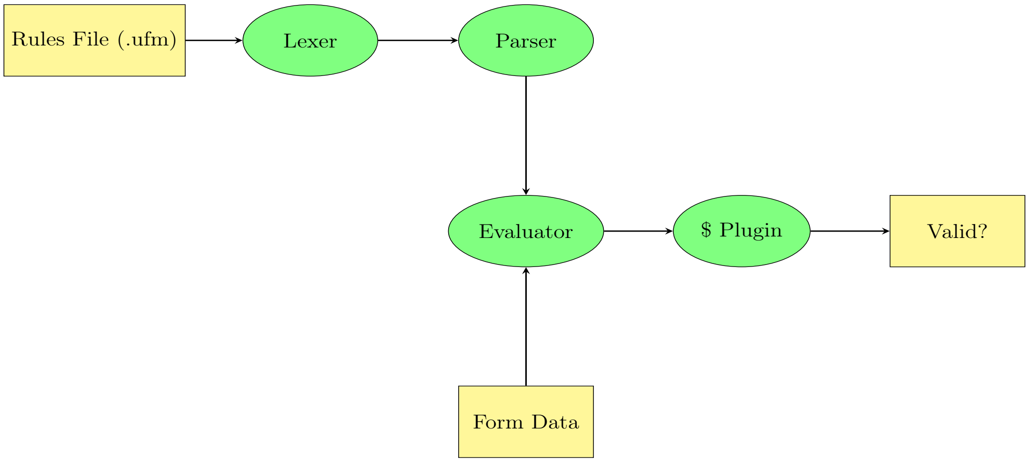


Figure 2: Language Architecture

9.1 Lexer and Parser

The parser is given the URL or content of a rules file written in the Uniform logic language. It reads and parses this file by analyzing the lexical components to determine a ruleset, which is a set of event listeners and handlers capable of enforcing the rules given in the parsed script.

9.1.1 Lexical Components

The lexer will break apart the Uniform file by determining what kind of lexical component each token is. The types of tokens are listed in table 1.

Type	Example	Meaning
Number	4912.30	A JavaScript Number type
String	"Hello \"World\""	A text string
Regex	/"[0-9]*"/	Regular expression
Selector	$("#myId.myClass")	A DOM element of the HTML page
Keyword	valid	Language keywords
Variable	@var: expression;	Stores the expression into a label
Comment	/*Comment*/ //Comment	Ignored Code

Table 1: Lexical Components

9.1.2 Expression Priority

Order of operations is complicated to express in grammar notation and as such the implementation does not need to be discussed here. Table 2 lists the order of operations that one can expect while writing Uniform code.

Priority	Name	Symbols	Associativity	Arity
Highest	Parentheses	(expression)	L	U
	Negation	-	R	U
	Mul Div Mod	* / %	L	B
	Add Sub	+ -	L	B
	Comparators	equals matches is < > <= >=	L	B
	Not	not	R	U
Lowest	And Or	and or	L	B

Table 2: Uniform Expression Priority and Associativity

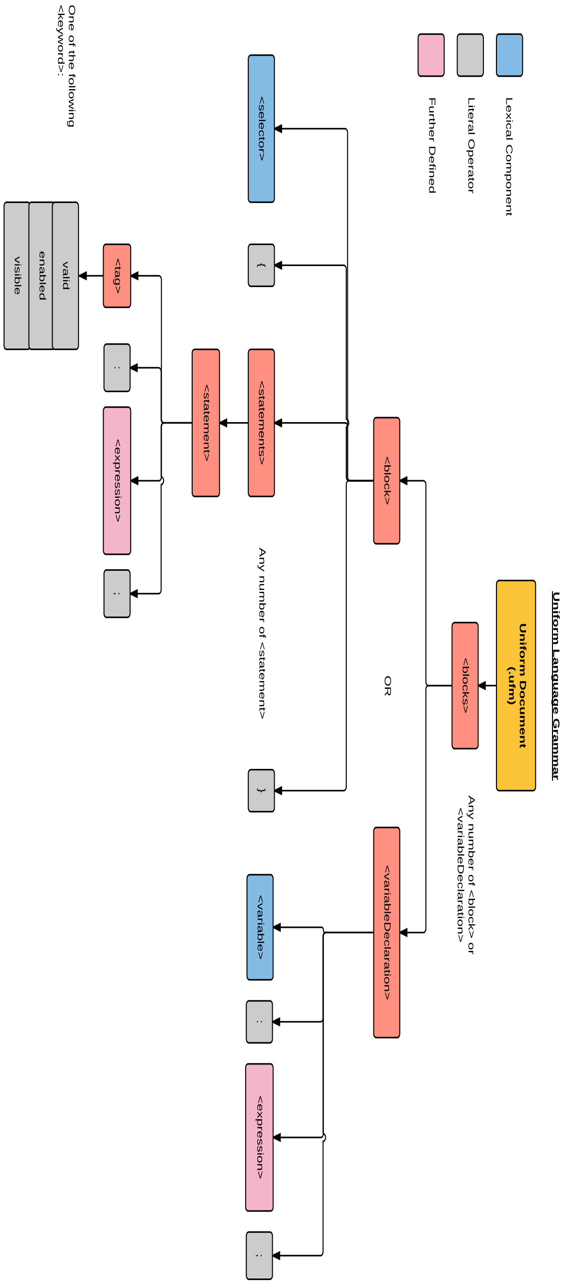


Figure 3: Uniform Grammar

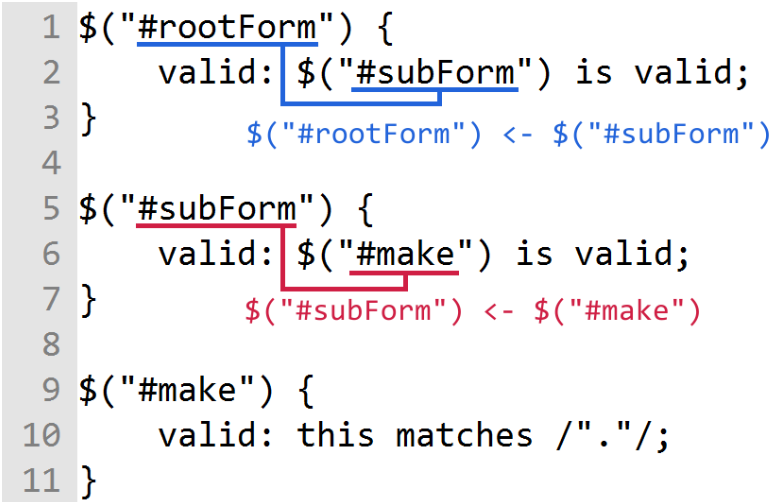
9.1.3 Grammar

The parser determines if a file is syntactically correct using the grammar illustrated in fig. 3.

9.2 Ruleset

The ruleset is the set of events and handlers created by the Uniform library when it parses a file. These handlers listen for the user to edit data on the form and then automatically update the page based on the rules specified in the .ufm file. When the user modifies a form control, the necessary event listeners will be triggered to update the page as necessary with the new data. From the car form example earlier, checking and unchecking the "I have a car" box instantly enables and disables the three text fields which provide the car's information. This ensures that the page is always in sync with any actions performed by the user by dynamically modifying itself and providing instant feedback to the user's actions.

When the user submits a form, Uniform is automatically queried to check if the submitted form is valid and sends the data to the server. In the event that the form is not valid, the data is not sent to the server and user is displayed a message telling him/her of the error. The user can then correct the problem and attempt to submit again.



```
$("#rootForm") {
    valid: $("#subForm") is valid;
}

$("#subForm") {
    valid: $("#make") is valid;
}

$("#make") {
    valid: this matches /"."/;
}
```

Figure 4: Dependency Code Example

The ruleset is implemented by setting up a dependency graph for the page. As the Uniform file is parsed, it identifies dependencies between various selectors and sets up the listeners to notify each other as appropriate. In the example from fig. 4, `$("#rootForm")` is dependent on `$("#subForm")` which can be read as “If `$("#subForm")` ever changes, it will notify `$("#rootForm")`”. There is a similar relationship between `$("#subForm")` and `$("#make")`. This means that if the user edits `$("#make")` and fills in a value, then `$("#make")` will become valid, so it will notify `$("#subForm")` to update itself. `$("#subForm")` then changes from invalid to valid, and so notifies `$("#rootForm")` to update itself and become valid. This allows the entire form to become valid instantly from only one minor change.

This dependency architecture allows any change to update only the validity of selectors that may have actually be affected by it. No time is wasted checking unrelated elements. This dependency graph is built from the Uniform file without utilizing the pages HTML, which means that it will not fall out-of-date as the DOM tree is updated over the page’s life-cycle.

9.3 Evaluator

The evaluator module is responsible for performing the actual computations within each expression. This has a unique problem due to a separation of inputs. The evaluator needs to know two inputs:

- How the data should be evaluated
  - *How* should it add, subtract, compare, etc.?
  - Comes from Uniform script when parsed
- The data itself
  - *What* should it add, subtract, compare, etc.?
  - Comes from DOM tree as user enters it

These two inputs come from very different places at very different times. Consider the Uniform script:

```
$("#rootForm") {
    valid: 2 + 2 equals $("#myNumber");
}
```

After parsing this script, Uniform knows that it needs to determine if `2 + 2` is equal to `$("#myNumber")`, but doesn't yet know the value of `$("#myNumber")`. Due to the nature of web development, the HTML page may not be fully loaded, or `$("#myNumber")` may be dynamically generated by some JavaScript code at a later time. As such, Uniform has no guaranteed knowledge of the DOM structure at parse-time. This expression needs to be parsed in some meaningful way, but cannot yet be evaluated for a result. To address this, Uniform utilizes currying.

Currying is a technique where a function is only partially evaluated and rather than returning a result, it returns another function that can finish its evaluation at a later time. Every evaluator function in Uniform generates a new function that can evaluate its result once given the appropriate inputs. A simplified example of this is shown below for addition and equality.

```
function add(leftExpr, rightExpr) {
    // Code here is executed at parse-time
    return function () {
        // Code here is executed at run-time
        var leftVal = leftExpr(); // Evaluate left operand
        var rightVal = rightExpr(); // Evaluate right operand

        return leftVal + rightVal; // Return add result
    };
}

function equals(leftExpr, rightExpr) {
    // Code here is executed at parse-time
    return function () {
        // Code here is executed at run-time
        var leftVal = leftExpr(); // Evaluate left operand
        var rightVal = rightExpr(); // Evaluate right operand

        return leftVal === rightVal; // Return equals result
    }
}
```

`add()` and `equals()` are called at parse-time, but the functions they return are evaluated at run-time. This creates an interesting distinction where code in the actual `add()` and `equals()` functions is executed at parse-time, when they are called. However code in the anonymous functions they return is evaluated at run-time, thus deferring the actual evaluation until Uniform has the data required to do so.

This also applies to the inputs `leftExpr` and `rightExpr`, which are functions that return the evaluated results of the left and right operands. Instead of directly putting the input of `2`, Uniform passes a function which returns `2`. This is a little strange for the data itself, but it makes it possible for the various evaluator operations to be chained together at parse-time. This chaining allows the parser to build the expression tree at parse-time and then evaluate it at run-time as data is entered by the. The expression tree for `2 + 2 equals $("#myNumber")` is displayed in fig. 5.

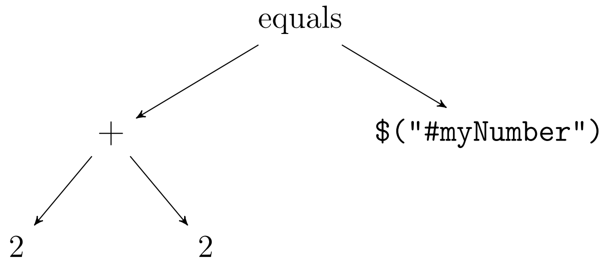


Figure 5: Expression Tree

When the expression is parsed, Uniform would create a getter for both of the constant `2` values, as well as a getter for `$("#myNumber")`. It would then use `add()` to create a new function which will evaluate `2 + 2`. Finally it would call `equals()` with the addition function it just created and the `$("#myNumber")` getter to make a function that can perform `2 + 2 equals $("#myNumber")`. A simplified example of this is below.

```
// Create getters for leaf nodes
var twoExpr = function () { return 2; };
var myNumberExpr = function () {
    return $("#myNumber").val(); // Get $("#myNumber")
};

// Generate expression for 2 + 2
var fourExpr = add(twoExpr, twoExpr);
console.log(fourExpr()); // Prints 4

// Generate expression for 2 + 2 equals $("#myNumber")
var equalsExpr = equals(fourExpr, myNumberExpr);

// Can call equalsExpr() to get current result
$("#myNumber").val(3); // Set $("#myNumber") to 3
console.log(equalsExpr()); // Prints false
$("#myNumber").val(4); // Set $("#myNumber") to 4
console.log(equalsExpr()); // Prints true
```

By using this architecture for evaluation, the parser is able to create the tree for each expression at parse-time, utilizing all the information it has from the Uniform file. Any time after that, as the user enters information into the page, the evaluation functions automatically read the new input and provide new results.

9.4 jQuery Plugin

In order to provide a usable and convenient interface, a plugin for jQuery is included in the Uniform library which provides easy and direct access to object state. This is used by the Uniform code internally and is also exposed to user code as well. The plugin is defined as `ufm`, and is accessed by taking any jQuery selector and executing the `ufm()` function.

```
$("#mySelector").ufm();
```

This `ufm()` plugin attaches several Uniform functions to the selector, exposing useful information about it. Most of this information is useful only to the Uniform library code, but the most notable and useful features of the plugin are the state getters.

```
$("#mySelector").ufm().valid();
$("#mySelector").ufm().enabled();
$("#mySelector").ufm().visible();
```

These functions return whether or not a given selector is valid, enabled, or visible based on the parsed Uniform script. This can be useful to some clients which may want to check if a particular selector is valid before performing an action. This would allow application developers to specify custom error text and provide more complex user interfaces.

Applications can directly access the Uniform state information for any selector, bypassing the need for validation code to be duplicated in JavaScript. Without this capability, displaying a custom error message would require the application developer to include additional JavaScript logic to identify the error which should have been handled by Uniform. This interface servers that need and allows JavaScript code to be added on top of Uniform where it is appropriate.

```
if ($("#mySelector").ufm().valid()) {
     alert("Looks good!");
} else {
     alert("Something's wrong.");
}
```

Since Uniform uses custom jQuery events to notify selectors of changes on the form, these events can also be used by user code to listen for updates to selector validity. The "ufm:validate" event is triggered on a particular selector when its validation state changes. This is used for internal purposes, but it also allows application developers to listen in and build dynamic interfaces on top of their Uniform codebase without having to recycle pieces of the validation logic.

```
$("#mySelector").on("ufm:validate", function () {
     console.log($("#mySelector").ufm().valid());
});
```

The "ufm:validate" event is triggered when the selector's state updates, meaning it may not necessarily have changed since the last time the event was triggered. If an application developer needed such a system, then he/she would need to track such state externally.

9.5 Server-Side Validation

A major requirement for the language was to allow Uniform to be executable server-side to validate client data automatically. This allows developers to use the same code on both the client and the server, saving developers from creating inconsistencies between the two systems or forgetting to update one over the other.

To allow Uniform to execute server-side, a few changes were made. Uniform utilizes Node's V8 engine to emulate the JavaScript environment present in a browser. The basic inputs and outputs have not changed, but the context is completely different on the server. On the client, data is pulled from the HTML DOM tree using the jQuery operator and output is displayed there as well. On the server, no such tree exists. Input data comes from an HTTP request sent by the client while the output is simply a true/false value representing the validity of the submitted form.

The majority of the Uniform codebase is unchanged between the client and server. All of the lexing, parsing, evaluation, and so on are all identical on both systems. Since the only differences are the inputs and outputs, that is where the changes were made.

To receive input from the request data easily without modifying the existing functionality and potentially creating bugs, the jQuery operator is overridden with a completely different implementation. This new jQuery function takes the same inputs, but instead of accessing the DOM tree, it reads the data inside the HTTP request and returns a mock jQuery object which is used by the same Uniform code as handles the real jQuery on the client. The rest of the codebase runs exactly as it does on the client but using these mock objects instead of real jQuery objects. This ensures that only a minimal number of changes to the proper Uniform code were required, with most of the work happening in the jQuery mock. This significantly reduces the chances of a bug which will cause the server-side and client-side validators to disagree about the validity of a particular form.

9.6 Install and Usage Guide

9.6.1 Client-Side

The client-side validation can be used as a plug and play system. Simply include the two lines

```
<script src="path/to/uniform.js"></script>
<script>uniform.options.href("path/to/script.ufm");</script>
```

Form Submission

Form submission is handled the same way it is in standard HTML. Uniform will automatically check that the submitting `<form>` tag is valid, and if so, it will submit the data in a format that the server validator expects. In the example below, when the user clicks the "Submit" button, the system automatically checks that `$("#myForm")` is valid, and if so, submits the checkbox (and any other inputs) as a `POST` request to `/submit`.

```
<form id="myForm" method="POST" action="/submit">
    <input name="chkBox" type="checkbox" />
    ...

    <button type="submit">Submit</button>
</form>
```

AJAX Submission

In single-page applications and often many other use cases, it becomes necessary to build an HTTP request directly in JavaScript and send it asynchronously without navigating the browser to a different page. To support this requirement, Uniform can be submitted as an AJAX request. This can be done using the `uniform.submit.ajax()` function.

```
var options = { method: "POST", url: "/submit" };

uniform.submit.ajax(options).then(function (res) {
    // Handle success
}, function (err) {
    // Handle error
});
```

The AJAX function is given an options object which should align to the format of the options in jQuery's `$.ajax()` call. Behind the scenes, all the `uniform.submit.ajax()` function does is replace the `options.data` value with Uniform data to submit, and then calls `$.ajax()` to send the request.

This does **not** check if a form is valid. This decision was made because AJAX requests are often built ad hoc for a particular kind of action which was probably hidden from the user and may not have any kind of `<form>` tag existing on the page to validate. Because of this, the `uniform.submit.ajax()` function does not check if anything is valid on the client. If the developer wanted such behavior, then he/she could simply check before sending the request like so:

```
if ($("#rootForm").ufm().valid()) {
    var options = { method: "POST", url: "/submit" };

    // Send AJAX request
    uniform.submit.ajax(options).then(function (res) {
        // Handle success
    }, function (err) {
        // Handle error
    });
} else {
    // Handle invalid form
}
```

It is considered a best practice to check the form for validity before submitting it, as it provides a better user experience for the client and allows them to fix the error more easily. However, in many scenarios it is not always practical to do so, therefore the client will have to depend on the server to notify it of any validation errors that have occurred.

9.6.2 Server-Side

The server-side validation module has been built to be used by a NodeJS server running the Express framework. Eventually, other frameworks and servers may be supported, but Node was chosen as the first because it is the easiest environment to execute Uniform from since it can already run native JavaScript code. Express was chosen because it is one of the most popular Node frameworks and supports plug-and-play middleware, making input parsing much easier. This platform was chosen for its simplicity and ease of development. Other platforms are expected to be supported in the future.

To install Uniform on the server and use it to validate client data, use Node Package Manager (NPM). Install NodeJS (NPM should come with it) and then open a terminal and navigate to the root directory of the server code. Then simply run

```
$ npm install uniform-validation
```

This will install the server-side package in the current directory. It can now be used by any Node server in the same directory. In order to run the validator, it must be required and called. The following code sample shows a simple blueprint for running the Uniform validator against a request.

```
var express = require("express");
var validate = require("uniform-validation");
var app = express();

// Generate middleware for UFM script and main selector
var validator = validate({
    path: <ufm-script>,
    main: <main-selector>
});

// On POST request
app.<verb>(
    // To this url
    <submit-url>,
    // Use the file at <ufm-script> with <main-selector>
    validator,
    // If <main-selector> was valid, run this code
    function (req, res) {
        res.end("Valid!"); // Successful
    },
    // If <main-selector> was NOT valid (or had an error),
    // run this code. MUST define all four params for Express
    // to run this correctly
    function (err, req, res, next) {
        res.end("Invalid!"); // Failed
    }
);
```

A filled-in example of this blueprint follows. When a `POST` request is sent to `/car/submit`, it is validated against the Uniform script residing at `car/car.ufm` to see if the `$("#rootForm")` selector is valid. If so, it will respond with the text "Valid!", otherwise it will respond with "Invalid!"

```
var express = require("express");
var validate = require("uniform-validation");
var app = express();

// Generate middleware for UFM script and main selector
var validator = validate({
    path: "car/car.ufm",
    main: "#rootForm"
});

```

```
// On POST request
app.post(
    // To this url
    "/car/submit",
    // Validate against "car/car.ufm" with "#rootForm"
    validator,
    // If $("#rootForm") was valid, run this code
    function (req, res) {
        res.end("Valid!"); // Successful
    },
    // If $("#rootForm") was NOT valid (or had an error),
    // run this code. MUST define all four params for Express
    // to run this correctly
    function (err, req, res, next) {
        res.end("Invalid!"); // Failed
    }
);
```

A complete example, showing both server and client code is shown in appendix D.

9.7 Security

Security is a top priority in this system. In a perfect world, the server would be able to trust any and all information given from the client. Unfortunately, in today's world there are always malicious users who will break and steal anything they can. Since any client could send a request, not necessarily a browser which has run Uniform validation, it is possible and expected that malicious users try to send invalid data to the server. Uniform must be able to validate these requests and dismiss malformed ones while accepting valid ones.

9.7.1 Uniform Ambiguity

The Uniform server-side validator checks the input data to the best of its abilities. Unfortunately, due to the current design of the language, it is effectively impossible to securely validate Uniform code on the server. Consider the following script:

```
$("#myForm") {
    valid: $("#foo") is valid;
}

```

```
$("input") {
    valid: true;
}

$("div") {
    valid: false;
}
```

When the server must validate data from a client, it is asked “is `$("#myForm")` valid?” which is a trickier question than it may seem on the surface. The server actually has no idea whether or not `$("#myForm")` is valid. This is because that information is not present in the Uniform file, but was actually in the HTML used by the client. Adding the following HTML to the previous Uniform code should make the solution clear.

```
<form id="myForm">
    <input id="foo" type="text" />
</form>
```

Considering this block on HTML as well, it is now clear that `$("#foo")` is an `<input />` tag and would therefore be considered valid due to the second Uniform rule. Consider if the HTML had been the following:

```
<form id="myForm">
    <div id="foo"></div>
</form>
```

Now it is clear that `$("#foo")` is a `<div>` tag and would therefore be considered *invalid* due to the third Uniform rule. This means that a single Uniform script can have multiple meanings based on the HTML which is being used. This is not a problem on the client since there is a single DOM tree to draw from. The server however does not have such a luxury and lacks this critical information.

9.7.2 Server Data Format

When data is sent from a client to a server using Uniform, the library automatically reformats the request to be more friendly to validation. The data is encoded in JSON and passed under the “ufm” form parameter to the server. This means that submitting the below car form looks like the following:

HTML Form

```
<form id="carForm">
    <input type="checkbox" id="hasCar" name="ownsCar" />

    <input type="text" id="make"  name="carMake"  />
    <input type="text" id="model" name="carModel" />
    <input type="text" id="year"  name="carYear"  />
</form>
```

Uniform Script

```
@filled: /"."/;
@integer: /"^[0-9]{4}$"/;

$("#rootForm") {
  valid: $("#make") matches @filled and
      $("#model") matches @filled and
      $("#year") matches @integer
    ;
}

$("input") {
    enabled: $("#hasCar") equals true;
}
```

POST Request Data

```
ufm={
    "#hasCar": [ { "value": true,    "type": "boolean" } ],
    "#make":   [ { "value": "make",  "type": "string" } ],
    "#model":  [ { "value": "model", "type": "string" } ],
    "#year":   [ { "value": "year",  "type": "string" } ],
    "input": [
        { "value": "make",  "type": "string" },
        { "value": "model", "type": "string" },
        { "value": "year",  "type": "string" }
    ]
}
```

The JSON data sent over the network is formatted as a map of jQuery selectors to the values they return. These jQuery selectors are all the ones which appeared in the Uniform script which causes some interesting behavior.

The first thing to notice is that the make, model, and year information is actually sent twice. Once under `$("#make")`, `$("#model")`, and `$("#year")`, and once more under `$("input")`. This can cause security issues as the client could lie about part of the data, giving different values for data which should be the same. Beyond this, there is a larger issue of lack of names.

In the HTML and HTTP world, a form is submitted according to the `name` attribute, *not* its ID or selector. If Uniform were not present on this system, the request would have looked like:

```
ownsCar=on&carMake=make&carModel=model&carYear=year
```

Note that the label for each parameter uses the `name` attribute from the HTML. This is how data is submitted and it is how the server identifies each input and uses them. On a Node server this format can easily be used to echo back the make like so (some setup and middleware omitted for brevity):

```
app.post("/submit", function (req, res) {
    res.end(req.body.carMake);
});
```

Trying to do the same task *after* the data has been formatted for Uniform is significantly harder and less intuitive.

```
app.post("/submit", function (req, res) {
    var data = JSON.parse(req.body.ufm);

    res.end(data["#make"][0].value);
});
```

This requires the server to parse the body, find the selector with the relevant data, and take the value of the first item. There is actually a lot of error checking that needs to happen here. The data may fail to parse into JSON. `$("#make")` may not exist in the request or may not be an array. The array could have zero elements or multiple, and it may not have a `value` key. This is a significant amount of validation to manually check for and results in specialized validation code being written on the server, something this system was specifically designed to avoid.

Beyond the security and validation issues, it is also very counter intuitive. These selectors come from the Uniform script, so only a selector which appeared *exactly* in the Uniform file will work. Searching for `$("input#make")` would come up empty, despite the fact that both `$("input")` and `$("#make")` appear. It also doesn't make sense to refer to elements by jQuery selectors when the server has no concept of what they are. The client works at a level of abstraction which lends itself to jQuery selectors, but the server works at a completely different level of abstraction which lends itself to individual argument names. Because of this, it is difficult, complicated, and unintuitive and consume the request data on the server and process it.

These are significant problems which prevent server-side validation from being a viable solution. However, there are two obvious ways of addressing these problems, unfortunately neither of them are feasible.

9.7.3 Extraneous Client Information

The first thought likely to jump into anyone's mind is to simply have the client send more information. If the client is able to validate without issue, then it clearly has all the information it needs, so just set the client up to send what it knows to the server which can then validate the data similarly.

This would address the problem of server-side validation, and give the field names that the server needs, however it would completely break the security of the system. Since the client may be malicious and therefore cannot be trusted, any extraneous information the client sends to help the server could be a lie. From the above example, the client could send its request the extra bit of information that `$("#foo")` happens to be an `<input />` tag. There is no easy way for the server to verify that statement and it must trust the integrity of the client. The server would then have the information necessary to validate the form and accept the user's input. However, if it turns out that the client lied and `$("#foo")` was actually a `<div>` tag, then the server should have denied that request, and has therefore let an un-validated request be executed. Such a situation is a total failure of the system and cannot be allowed under any circumstances. Because of this security issue, the client cannot send any more information than the actual data it is trying to submit. The rest *must* be known to the server via secure channels.

9.7.4 Server-Side HTML

The other obvious solution to the missing information on the server is to simply have it load up the same HTML that was given to the client and use that for validation. Aside from the significant performance hit, this would actually solve the problem and still be secure, but is technically infeasible on modern web systems for two reasons.

The first reason is that HTML is no longer a static structure. A few decades ago, most sites were built with plain static HTML, but this simply isn't the case anymore. Almost every modern server incorporates some kind of HTML preprocessing, whether its ASP, JSP, PHP, Ruby on Rails, or any number of Node preprocessors. Almost every web page involves some kind of dynamically generated content, a simple user name in the corner, region and analytics information, even advertisements. As such, every client receives different content from the server in a very difficult-to-predict manner. To avoid bugs from inconsistent HTML, the system would need to guarantee that the server will receive the same content that was given to any particular client with no meaningful differences. There is no way Uniform can help in this regard, since the overall server architecture is completely outside the scope and knowledge of this one small library, so it is up to application developers to perform this task. The killer here is that it would take more time and effort to build a system that can consistently return matching HTML to the client and server than it would take to simply validate the form data on both systems manually. Because of that, no developer would spend the effort building such a system in order to use Uniform over their existing solution.

Even ignoring that problem, there is a second issue in dynamic clients. Modern web sites use JavaScript to create and alter web content directly on the client as the user navigates the site. As such, the HTML the client received from the server and the page the user views and interacts with may be completely different from each other. The form which was submitted by the user may not even exist in the HTML visible to the server. That form may have been dynamically created by client JavaScript based on user actions. In order to validate data from such clients, the server would need to know exactly what actions the client performed and then replicate them server-side before validating the data. Unfortunately, this falls under the previous problem of being unable to trust the client to provide any extraneous information, since they could easily lie by saying that one action was performed when in reality a different action was, causing a security vulnerability. Due to these constraints, it is not feasible to validate data by using HTML content accessible to the server.

9.7.5 Language Redesign

The problem of validating on the server is really the result of a language design error. The current design of the language makes it impractical to validate on the server, so the design will have to change. It is unclear exactly how it will need to change to be viable on the server, but mostly likely the jQuery functionality will need to be replaced. Any jQuery selector that appears in a Uniform script is an ambiguity that can only be solved with knowledge of the HTML structure. As such, a more consistent and stable system of referring to data must be developed. One solution would be to replace jQuery selectors with direct name references. A reworking of the previous car form under this new system is below, with the original HTML and Uniform code reproduced for convenience.

HTML Form

```
<form id="carForm">
    <input type="checkbox" id="hasCar" name="ownsCar" />

    <input type="text" id="make"  name="carMake"  />
    <input type="text" id="model" name="carModel" />
    <input type="text" id="year"  name="carYear"  />
</form>
```

Current Uniform Script

```
@filled: /"."/;
@integer: /"^[0-9]{4}$"/;

$("#rootForm") {
    valid: $("#make") matches @filled and
        $("#model") matches @filled and
        $("#year") matches @integer
    ;
}

$("input") {
    enabled: $("#hasCar") equals true;
}
```

Proposed Uniform Script

```
1  @filled: /"."/;
2  @integer: /"^[0-9]{4}$"/;
3
4  // Page is valid when...
5  valid: make matches @filled and
6      model matches @filled and
7      year matches @integer
8  ;
9
10 // Define inputs as make, model, and year
11 @inputs: [ make, model, year ];
12
13 // All inputs are enabled when checkbox is checked
14 @inputs {
15     enabled: hasCar equals true;
16 }
```

Under this new system, there is no way to refer to non-input fields (since they do not have names). As such, it is impossible to actually define any logic for a form since it is not an input. Lines 5-8 define the valid condition for the entire script and is what the server would attempt to validate. Line 11 defines a new variable `@inputs` as the set of `make`, `model`, and `year` inputs. It is then used on lines 14-17 to define a single enabled rule for all three elements. This is a different way of approaching the same validation problem which can be validated on the server.

There is no ambiguity in this script because the relationships between elements are explicitly placed in the Uniform code rather than hidden in the HTML. Previously, the fact that `$("input")` was the same as `$("#make, #model, #year")` required knowledge of the HTML structure. In this new format, it is explicitly defined defined in the Uniform script in the `@inputs` variable.

This also solves server data usage issue because the data does not need to be reformatted between the client and server, meaning it can remain it its normal URL encoded state with input names preserved. After validation, the server would be able to use these values in the format which makes sense to it, independent of the client.

This does have downsides however. Uniform is not able to take advantage of the structure already defined in the HTML and forces developers to keep a strong consistency between the Uniform code and the HTML form it validates. This is still far less effort than maintaining two or three form validators on the client and server, so it is not a deal breaker, but an annoying imperfection that can never be fixed. The larger downside is that Uniform can no longer take advantage of the power, simplicity, and convenience of jQuery's selectors. The language is now much less versatile because it can no longer work on arbitrary jQuery selectors, and is worse for it. This is unfortunate, but it may be the price for security.

None of the syntax or features discussed in this section are finalized and nothing here has been implemented yet. This was merely a possible solution to the problems that were encountered in server-side validation. The rest of the document will refer to the language as it exists today.

Uniform can do a lot for the security and maintainability of a system, but it must be used correctly or vulnerabilities can occur. There are two significant ways in which the system can be misused to create vulnerabilities.

9.7.6 Cross-Script Validation

Consider the following code:

```
var validate = require("uniform-validation");
...
app.post("/submit", validate({
    path: req.query.script, // Uniform script path
    main: "#rootForm" // Element to validate
}), function (req, res) {
    ...
});
```

Note that the Uniform script path is `req.query.script`, meaning that it is derived from the request and therefore provided by the client. The server is probably expecting `req.query.script` to point to a file that looks like:

www/myStuff.ufm

```
// Validates /api/myStuff
// Form is valid if all inputs are valid
$("#rootForm") {
    valid: $(".myStuff") is valid;
}
```

However, the client could provide a different URL to a script that reads:

www/otherStuff.ufm

```
// Validates /api/otherStuff
// Form is valid if all other inputs are valid
$("#rootForm") {
    valid: $(".otherStuff") is valid;
}
```

The server would then validate data intended for /api/myStuff against the Uniform script intended for /api/otherStuff. The client could easily add attributes which satisfy the script for /api/otherStuff, while leaving bad data in attributes that are used by /api/myStuff.

This has been dubbed “Cross-Script Validation,” as the client has tricked the server into validating the data with a different Uniform script than it should have. In order to avoid this problem, the server cannot trust the client for the location of the validation script, so a specific API should connect with a specific Uniform script which validates it known directly to the server. A better implementation of this would be:

```
var validate = require("uniform-validation");
...
app.post("/submit", validate({
    path: "www/submit.ufm", // Uniform script path
    main: "#rootForm" // Element to validate
}), function (req, res) {
    ...
});
```

Note that "www/submit.ufm" has now been directly written into the server's code, which means the server will *always* use the correct script, regardless of what data the client sends.

9.7.7 Main Misdirection

A similar vulnerability involves the main selector. The validator must be told which selector needs to be validated so it knows what must be valid to proceed. Consider the following code:

```
var validate = require("uniform-validation");
...
app.post("/submit", validate({
    path: "www/submit.ufm", // Uniform script path
    main: req.query.main // Element to validate
}), function (req, res) {
    ...
});
```

Notice that the element to validate is `req.query.main`. This value is derived from the request object and is therefore provided by the client. Similar to Cross-Script Validation, the client could easily lie about this particular value with something more favorable to the attacker. Consider the script:

www/submit.ufm

```
@filled: /"^[0-9]{4}$"/;

$("#rootForm") {
    valid: $("#make") is valid
        and $("#model") is valid
        and $("#year") is valid;
}

$("#make") {
    valid: this matches @filled;
}

$("#model") {
    valid: this matches @filled;
}

$("#year") {
    valid: this matches @filled;
}
```

The server was probably expecting the client to tell it the main selector was `$("#rootForm")`. However, the client could easily say that the main selector is actually `$("#make")`. The server would then check if `$("#make")` is filled, and if so, the request would be considered valid. Meanwhile, the client could have easily slipped invalid data into `$("#model")` and `$("#year")`.

This has been labeled “Main Misdirection” as the client has tricked the server into validating against the wrong main selector. To avoid this problem, the server should not trust the client to tell it what main selector to check, but it should rather be specific to the API that is being called, as shown below.

```
var validate = require("uniform-validation");
...
app.post("/submit", validate({
    path: "www/submit.ufm", // Uniform script path
    main: "#rootForm" // Element to validate
}), function (req, res) {
    ...
});
```

Note that `"#rootForm"` has now been directly written into the server's code, this means that the server will *always* use the correct main selector, regardless of what data the client sends.

10 Design Rationale

10.1 Usability

The more complex a system gets, the harder it becomes to use. If the customer or staff cannot figure out how to operate the system, its design becomes meaningless. This language will focus on ease of use, independent of existing programming skills. Business level users will be able to drive the form creation process instead of relying on programmers. Of course there is still a learning curve for non-programmers, but learning Uniform is a significantly easier to learn over the alternatives. Several design decisions were made to create the ideal user experience.

10.1.1 Logical Design

Uniform's design is structured around the programmer's intuition. While it can be very easy to create logical statements, it is not always easy to write them as code. Uniform abstracts the programming in a way that allows the user to focus on the logic rather than how it will be implemented. The language is structured such that, if code is read aloud, one would be able to understand the logic through natural spoken language without having to decrypt complicated syntax.

10.1.2 CSS Format

CSS is a styling language that is a staple in any web designer's arsenal. The language’s syntax mirrors this style, taking advantage of the familiarity of CSS to become easier to read and learn for business people and other non-programmers. CSS follows the standard Document Object Model to manipulate elements in HTML. Uniform uses the same selectors, but instead of changing the style, it changes the logic. This concept is intuitive to understand and lowers the learning curve.

The language structure also takes inspiration from some features of the LESS language. This includes variables and nesting, two features easy to understand which provide a lot of power and scalability to the language.

10.2 Efficiency

As with all layered architectures, adding yet another layer of abstraction results in a performance hit. The following measures were taken to ensure an acceptable performance time.

10.2.1 Nested Form Evaluation

The validator evaluates forms recursively bottom up by determining if the lowest level form is valid, and using that result to determine if the next level is valid, until finally reaching the outer-most form to determine if it is valid. Being tree-like in structure, if one sub-form is not valid, it can already be determined that the whole form is not valid. This short circuit allows validation to end early on incorrect forms and pinpoint the source of the error.

This structure also allows us to save time when making changes to the form. For example, if there are three main sub-forms, A, B, and C, the user fills each of them out, and the validator determines that the form is valid. If later, sub-form C is changed, the system would not necessarily need to validate A and B again depending on their relationship with C.

10.3 Design Patterns

On the client, whenever a form is submitted, it is checked to see if it is valid. This has a lot of power and a couple major benefits.

- Decentralized: Every element is responsible for its own validity. Individual complexity does not build as forms get bigger, each element only cares about itself.

- Multiple forms: A single page can have multiple forms, a user could fill out one form to do one action and fill out another form to perform another action. These two forms may or may not be related and may or may not share logic.

On the server, every exposed API has a set of data it will accept in a particular format that must be verified. This means it has a single set of rules to determine if a request is valid. Some APIs may accept multiple formats, but that is rare and usually has the same logical structure but a different physical one (such as JSON vs XML, same data, different format). This view contradicts the decentralized nature that the client sees the Uniform architecture as.

To address the discrepancy, the server has the concept of a "main selector." This is the "entry point" for the validation and is the one selector which must be valid for the request to be considered valid. All other selectors are irrelevant to the server, and only present to help determine the validity of the one main selector.

It is of course possible for two APIs to use the same script with different mains, and to use the same main for different scripts. A suggested best practice would be to equate one script and main to one API. If a particular web page could access multiple APIs, then it would download multiple Uniform scripts which are both present for the client. When they invoke a particular API, then that API can be validated against the one script that it depends on while staying separate from the other script. An example of this is below:

/index.html

```
<html>
    <head>
        <script src="/jquery.js"></script>
        <script src="/uniform.js"></script>
        <script>
            // One script dedicated to sending
            uniform.options.href("/send.ufm");
            // One script dedicated to receiving
            uniform.options.href("/receive.ufm");
        </script>
    </head>
    <body>
        <!-- One form for sending -->
        <form id="send" method="POST" action="/send">
            <input id="sAmount" name="amount" type="text" />
            ...

            <button type="submit">Submit</button>
        </form>

        <!-- One form for receiving -->
        <form id="receive" method="POST" action="/receive">
            <input id="rAmount" name="amount" type="text" />
            ...

            <button type="submit">Submit</button>
        </form>
    </body>
</html>
```

/send.ufm

```
// One script for sending
@filled: /"."/;

$("#send") {
    valid: $("#sAmount") matches @filled;
}
```

/receive.ufm

```
// One script for receiving
@filled: /"."/;

$("#receive") {
    valid: $("#rAmount") matches @filled;
}
```

server.js

```
var validate = require("uniform-validation");
...
// One API for sending
app.post("/send", validate({
    path: "www/send.ufm", // Uniform path (sending)
    main: "#send" // Element to validate (sending)
}), function (req, res) {
    ...
});
...
// One API for receiving
app.post("/receive", validate({
    path: "www/receive.ufm", // Uniform path (receiving)
    main: "#receive" // Element to validate (receiving)
}), function (req, res) {
    ...
});
```

10.3.1 Shared Logic

This design pattern reduces general complexity and divides the Uniform logic where it is appropriate while still allowing the components to be combined onto a single web page. This is likely to be applicable in a majority of situations, but the pattern does begin to break down when multiple forms share logic between them. At that point, separating the Uniform files can start to cause difficulties. There are two obvious options in that particular scenario.

One option is to combine the two scripts into one .ufm file containing both sets of logic. The client could then use the one script rather than the original two files, while the server could have two APIs that both reference the same Uniform file, but use different main selectors to ensure that the right values are valid. This option would intrinsically mix the two together and make it much harder to maintain and reason about, but it is a valid option. This would likely be preferable if the two scripts are relatively small and strongly related to each other. In such a case, combining the two scripts would have a minimal increase in complexity while maximizing the benefit of the integration between them. Below, the previous example has been modified to illustrate this.

/index.html

```html
<html>
    <head>
        <script src="/jquery.js"></script>
        <script src="/uniform.js"></script>
        <script>
            // One script for sending and receiving
            uniform.options.href("/sendAndReceive.ufm");
        </script>
    </head>
    <body>
        <!-- Body is unchanged and omitted for brevity -->
    </body>
</html>
```

/sendAndReceive.ufm

```
// One script for sending and receiving
@filled: /"."/;

$("#send") {
    valid: $("#sAmount") matches @filled;
}

$("#receive") {
    valid: $("#rAmount") matches @filled;
}
```

server.js

```javascript
var validate = require("uniform-validation");
...
// One API for sending
app.post("/send", validate({
    path: "www/sendAndReceive.ufm", // Same script path
    main: "#send" // Main selector (sending)
}), function (req, res) {
    ...
});
...
// One API for receiving
app.post("/receive", validate({
    path: "www/sendAndReceive.ufm", // Same script path
    main: "#receive" // Main selector (receiving)
}), function (req, res) {
    ...
});
```

The other option is to factor out the shared logic between them into a third .ufm file. The original two scripts would then be dependent on the new one. A client would need to reference all three, while the server would then have each API reference the script it needs along with its new dependency. This would likely be preferable if the two scripts are modular and large enough to justify adding another script explicitly for logic shared between them. In such a case, the new script would have its own internal logic and consistency which others could utilize for their own purposes. This is the more scalable and maintainable solution of the two. Below, the previous example has been modified to illustrate this. It may seem trivial and wasteful in such a small example, but in larger pieces of code this can easily be the better option.

/index.html

```
<html>
    <head>
        <script src="/jquery.js"></script>
        <script src="/uniform.js"></script>
        <script>
            // One script for shared logic
            uniform.options.href("/shared.ufm");
            // One script for sending logic
            uniform.options.href("/send.ufm");
            // One script for receiving logic
            uniform.options.href("/receive.ufm");
        </script>
    </head>
    <body>
        <!-- Body is unchanged and omitted for brevity -->
    </body>
</html>
```

/shared.ufm

```
// One script for shared logic
@filled: /"."/;
```

/send.ufm

```
// One script for sending logic
$("#send") {
    valid: $("#sAmount") matches @filled;
}
```

/receive.ufm

```
// One script for receiving logic
$("#receive") {
    valid: $("#rAmount") matches @filled;
}
```

server.js

```
var validate = require("uniform-validation");
...
// One API for sending
app.post("/send", validate({
    // Uniform script paths (sending)
    path: [ "www/shared.ufm", "www/send.ufm" ],
    main: "#send" // Main selector (sending)
}), function (req, res) {
    ...
});
...
// One API for receiving
app.post("/receive", validate({
    // Uniform script paths (receiving)
    path: [ "www/shared.ufm", "www/receive.ufm" ],
    main: "#receive" // Main selector (receiving)
}), function (req, res) {
    ...
});
```

11 Technologies Used

To implement the Uniform language as it has been described, the following technologies were utilized:

- HTML: Markup language used for displaying content in a standardized way across web browsers.

- JavaScript: Client-side scripting language.
    - jQuery: JavaScript library which allows easier access to HTML DOM elements.
    - Jasmine: JavaScript test framework and assertion library.
    - Node JS: JavaScript runtime
        - Express: Node framework for building easy and scalable web APIs

12 Testing

A testing procedure for a language verifies that there is no ambiguity and that each case can be resolved consistently. The grammar and syntax for a language must be accurate in a way where the programmer knows exactly how his/her code will be executed. As such, exceptions, error handling, and correct parsing must be considered in the procedure.

12.1 Unit Testing

Unit testing has be performed throughout the development process as part of utilizing a test-driven development mentality. These unit tests were the majority of tests written during the implementation phase and have be useful to track down bugs that surfaced during development. Full test case implementations can be found in the Uniform source code in the "tests" folder.

12.2 User Testing

The system has been shown to multiple users and their feedback has been used to improve it. The primary concern for user testing was learnability and ease of use. A group of seven students, three of which were beginner programmers, one was an intermediate programmer, and the final three were advanced programmers.

Each tester attended a one hour workshop which the language was taught through several lessons and exercises. At the end of the workshop a quiz and usability survey was given to the testers. The primary goals for testing was to determine:

- The correlation between programming skill level and correct Uniform code.
- How quickly Uniform can be learned.
- How difficult it was for users to be engaged in learning Uniform.
- If any bugs were present in the code.
- What improvements could be made to further benefit the user.

12.2.1 Quiz Results

Out of all the testers, everyone was able to generate the correct response for the quiz questions given various amounts of time to complete it. After the quiz, talking individually with the testers has shown that beginner and intermediate programmers had a very difficult time learning the basics of programming, syntax, and how the code affected the web form on the screen. Though each beginner tester stated that given enough time to learn the language, they would be confident in using it. Advanced programmers on the other hand were almost instantly able to use the language as intended. This shows that the syntax follows general convention and is quickly learned on the higher skill brackets.

12.2.2 Improvements Suggested

What would make Uniform easier to understand?

Here are some of the responses given when asked the question:

- “If you know CSS to begin with”
- “More time to study the language”
- “Nothing that I can see. The language enables power users, I like it!”
- “Would be understandable if classes are taken or I had more time to study it”
- “Easier syntax”
- “jQuery notation and such”

- “I thought it was fairly easy to understand but if someone is not used to jQuery then it might be difficult.”

These responses highlight the learnability gap between beginners and advanced users. Adept users generally enjoyed the simplicity of the language whereas novices struggled with fundamentals such as if the dollar sign or the parenthesis goes first.

Would you be interested in using Uniform in the future?

100% said yes. Here are some of the responses given when asked why:

- “Very straight forward and intuitive. Especially since I have programming knowledge”
- “Easy and cost-efficient ”
- “Better cause I can actually do things programmatically instead of using the wiziwigg stuff with other forms.”
- “makes [form validating] simpler”
- “It was very similar to jQuery so it made it easier for me.”
- “It was straightforward once it was explained to me”
- “Simple”

It is important to highlight here that everyone including beginners saw the potential and power of using Uniform regardless of their personal ability to use it. Skilled programmers generally enjoyed the freedom and ease of use, which non-skilled programmers appreciated its simplicity.

Further Testing Analysis

Besides the various bugs that were discovered during the testing event, testing has raised some very interesting arguments that were not considered before. This testing session was the first time anyone other than the developers were able to use and provide feedback on the language. Therefore it was eye-opening to be given an outside perspective. Given that the three developers were indeed skilled senior computer engineers, we had forgotten the experience of learning to program for the first time and the challenges it brings.

To truly make Uniform both a powerful and simple language that can be used effectively by both beginners and veterans, some design decisions are to be considered.

The biggest change concerns syntax. While we agree that there will be an inevitable learning curve to learning what goes where, the learning experience can be improved upon. One suggestion that came out of testing was that one user found that the wording “valid if checkbox equals true;"" was not intuitive. A normal person would be tempted to think “valid if checkbox equals checked”. Several aliases could be made to compensate for the different ways people think. One of Uniform’s core design philosophies is “think it, don’t program it”.

12.3 Browser Testing

Uniform has been tested and proven to work properly on the following browsers:

- Safari
- Google Chrome
- Mozilla Firefox
- Opera

12.4 jQuery Support Testing

jQuery support is tested to work on version 1.8 and above.

13 Development Problems and Solutions

13.1 Detecting Circular Dependencies

13.1.1 What is a dependency?

A dependency is updating elements that relate with one another. If A validates B, if A changes, B needs to be updated as well. A dependency is an indirect connection to elements. For a form to be fully complete, every dependent needs to be updated when its child is updated.

13.1.2 What is a circular dependency?

A circular dependency is when A validates B and B validates A. If Uniform tries to update either A or B, it will endlessly try to validate both, since they are changing each other over and over. This is a simple example, although a dependency graph can get quite complicated and it is not as simple as solving edge cases.

13.1.3 Can we detect at compile time?

To understand why the problem is unfeasible to resolve, the structure of Uniform must be understood. So how does Uniform update its dependents? It does so by using an event based architecture. Going again on a simple example: C validates D and D validates E, when C is updated, it sends an event that says "Myself, the C element, was updated, anyone who is dependent on me please update". Meanwhile D and E are constantly listening for C to say this, and E is listening for D to say this. It is important to note that C has no idea who its dependents are. Therefore our program has no idea who is listening and cannot ensure that every element is updated until in a specified amount of time, the dust settles and no one is shouting for an update.

13.1.4 Why it needs to be this way

Simply put, the page can update at any time. An event based architecture listens all the time. A change event, or user input, is what begins the dependency update call.

13.1.5 The seemingly easy solution

It may seem like a simple task to just implement code that, simply put, tells the element not to update and not to send a dependency update if it hears from the same person twice. This leads to an issue with resetting the code. What if a user changes a textbox field twice? Then the dependency graph needs to fire another update request. If each node only listens from a dependent once, then the page can only load once. How do we reset the page? We would have to recompile and rerun the Uniform code, which is not a scaleable solution.

13.1.6 The surprisingly hard solution (still doesn't work)

We could then potentially build a dependency graph. There are many algorithms that can be run on the graph to determine if there are any loops, however, because of the nature of our system, the graph cannot be deleted. Every change event will simple append new nodes. If a form is large enough, is open for a long enough time, and is constantly being updated, we would be generating large overhead. This could lead the browser to overload its memory and crash.

13.1.7 Decision: Ignore the problem

Given the size and complexity of this issue, it is not worth our time and resource in a short one-year project to resolve this. If a circular dependency does occur, it is the Uniform programmer's fault, not the form filler. A circular dependency will crash the web browser. A simple page refresh will break the crash. The penalty for ignoring the problem is very minor. The programmer will see that his code will crash the page, and this is the browser crash is the best error message we are able to generate with the limited resources we have.

13.2 Server-Side HTML Knowledge

The server-side validator's inability to validate various scripts due to a lack of knowledge of the accompanying HTML is thoroughly discussed in section 9.7. This was a major design flaw which was encountered far too late in the development cycle. Throughout the process there was an emphasis of working on features which could be easily demoed and presented. Server-side validation did not fit those criteria and was delayed as a result. Development on server-side validation did not begin until the last quarter of the total development time allotted, most of which was lost to documentation, presentations, and other schoolwork.

If the server-side module had been started earlier, the hidden dependency on the HTML structure would have been identified in Winter quarter and there would have been time to fix the design flaws in the language prior to completely implementing it on the client. Due to time constraints, it is impossible to fix in the current timeframe and will need to be remedied in future development work on the language.

Part II
Web Application

14 Solving the Human Element and Mobile Use Cases

Uniform is targeted at enterprise users, that is, businesses and individuals who rely on forms as critical sources of data. Although the Uniform language is designed to be an easy-to-pick-up abstraction of current web technologies, the language on its own is best described as middle-ware. To shine, Uniform needs to be embedded in a platform or application. To that end, a Web Application was built that integrates Uniform to drive form creation, completion, and management while showcasing the advantages Uniform offers over current solutions.

15 System Components

15.1 Actors

- Administrator
  - Responsible for template creation.
  - Creates form templates, sets form validation rules, shares associates' capabilities.
- Associate
  - Assists clients in filling out the form.
  - Instantiates form instances, can provide access to owned form instances, can perform general user tasks.

- Client
  - Any individual who wants to fill out out the form.
- User
  - Anyone accessing the web application.
  - Fills out forms, communicates through the web application, saves and submit forms.

15.2 Form Terminology

- Form Template
  - A template containing the form appearance, controls and their metadata, and validation logic.
- Form
  - An instance of a form template.
- Sub-Form
  - A form that is also the child of a form element.
- Control
  - Form elements that require input from the user such as dropdowns, radio selectors, checkboxes, or textboxes.

16 Requirements

The Uniform web application can be broken down into the following requirements and constraints. Functional requirements state what the system is capable of while the non-functional requirements state the manner in which the functional requirements will achieve their goals. A critical requirement is a core part of the systemâĂŹs functionality, a recommended requirement is a feature that will enhance the systemâĂŹs overall usability and usefulness, and a suggested requirement is something that is useful, but not essential, to the system.

16.1 Functional Requirements

16.1.1 Critical

- The system allows
  - administrators to create form templates.
  - administrators to apply form validation logic to templates.
  - associates to instantiate instances of form templates.
  - associates to set who has read/write access to form instances.
  - users to fill out forms collaboratively.
  - users to submit forms.
  - users to communicate with each other within the system.
- The system stores form data in a repository.

16.1.2 Suggested

- The system allows
  - associates to define permissions for different sections of a form.
  - associates to forcibly submit invalid forms.
  - administrators to modify existing form templates.
  - forms to be audited.
  - changes to be logged.

16.2 Non-Functional Requirements

16.2.1 Critical

- The system will be mobile friendly:
  - The system interface will be friendly to mobile devices.
  - The system will be easily usable from within a native application.
- The system will be accessible:
  - The system will be intuitive to use for non-programmers.

  - The form instances will be easy to fill out.

- The system will be maintainable:
  - The system will be easily maintainable by other developers.
  - The system will log and report exceptions appropriately.
  - The system will be able to recover from fatal exceptions.

- The system will be responsive:
  - The system will provide quick and responsive feedback to user interaction.

- The system will be scalable:
  - The system will be able to handle large numbers of users, stored forms, and templates.

16.2.2 Recommended

- The system will be efficient in both response time and save time.

- The system will be well-documented.

16.3 Design Constraints

- The system must conform to Hewlett-Packard's User Experience standards

16.4 Post Implementation Evaluation

Following the final implementation of the Web Application but prior to the turnover to Hewlett-Packard, our conclusions regarding the degree to which we have met these requirements follow.

16.4.1 Functional Requirements

Fundamentally, all critical requirements were satisfied, resulting in a suitable product. That said, the requirement that associates are able to provide read/write access to form instances could be characterized as having been only partially met as associates are not able to limit certain user access to read only.

Suggested requirements, by comparison, were largely left out. Only the third suggested requirement, that administrators be able to modify existing form templates, was satisfied in the final implementation.

16.4.2 Non-Functional Requirements

Generally, these requirements have been satisfied. However, milestones to measure the achievement of these requirements were never explicitly set so this conclusion is largely a matter of opinion. Without external developer input, for example, it cannot conclusively be stated that the system is maintainable other than to say it has been implemented largely in keeping with industry best practices.

16.4.3 Recommended Features and Design Constraints

Overall the system performs adequately when accessed via an external server, to the extent that response time was not considered an issue following user testing, but again, no explicit milestones were set to measure this achievement.

Similarly, the system makes use of Hewlett-Packard's User Experience standards, but does take some creative license outside of these standards, so this Design Constraint was only partially satisfied.

17 Use Cases

There are a number of ways that users can interact with this system, so a use case diagram was created in fig. 6 to graphically illustrate the different ways the system will be used.

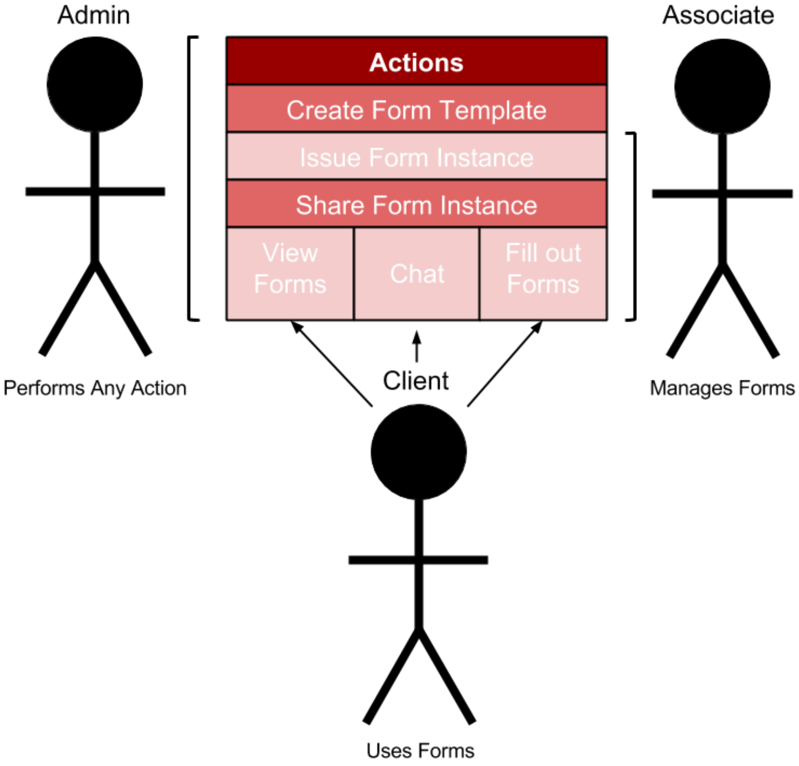


Figure 6: Use Case Diagram

17.1 Create a Form Template

Goal: Create a new form template
Actors: Administrator
Pre-Conditions: Logged in
Post-Conditions: Template is created
Steps:

1. Select “Build Template”
2. Name template
3. Generate Elements
4. Create Validation Script
5. Save template

Exceptions: None

17.2 Issue a Form Instance

Goal: Create a form instance for a customer
Actors: Associate or Administrator
Pre-Conditions: Form template exists. Associate/Administrator is logged in.
Post-Conditions: Form instance is created
Steps:

1. Access Forms
2. Select “Issue Form”
3. Name Form
4. Select base template
5. Provide list of authorized users

Exceptions: None

17.3 Provide Access to a Form Instance

Goal: Modify permissions of a form instance
Actors: Form Issuer
Pre-Conditions: Form instance exists. Associate/Administrator is logged in and owns the Form
Post-Conditions: Permissions have been changed on form instance
Steps:

1. Select form instance
2. Select “Share”
3. Provide list of authorized users

Exceptions: None

17.4 View a Form Instance

Goal: View data entered in a form instance
Actors: User
Pre-Conditions: User has permission to access the instance. User is logged in
Post-Conditions: User can view the form instance
Steps:

1. Select form instance
2. Select “View”

Exceptions: None

17.5 Fill out a Form

Goal: Fill out information on a form instance
Actors: User
Pre-Conditions: User has permission to edit the form instance. User is logged in
Post-Conditions: User has partially or completely filled out the form instance
Steps:

1. Select form instance
2. Select “View”
3. Fill out information
4. Save or submit form instance

Exceptions: Cannot submit due to validation errors

17.6 Communicate through Web App

Goal: Communicate a question or comment to other users about the form instance
Actors: User
Pre-Conditions: User has permission to view the form instance. User is logged in
Post-Conditions: User has commented on a form control
Steps:

1. Select form instance
2. Select “View”
3. Select Comments
4. Use chat interface

Exceptions: None

18 Activity Diagrams

The following diagrams specify the actions of users navigating the Uniform web application to manage forms and templates.

18.1 Client Workflow

Client Workflow (fig. 7) represents the relatively straightforward task of accessing a form instance and inputting data and commenting as necessary until the form can be validated and submitted.

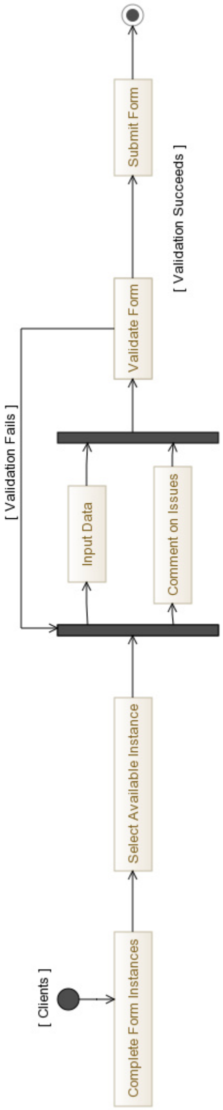


Figure 7: Client Workflow

18.2 Associate Workflow

Associate Workflow (fig. 8) resolves into two main tasks focused on form management. On the one hand an associate will issue new forms and share them with clients, while on the other hand they oversee existing forms, assisting clients as necessary and adjusting access as necessary.

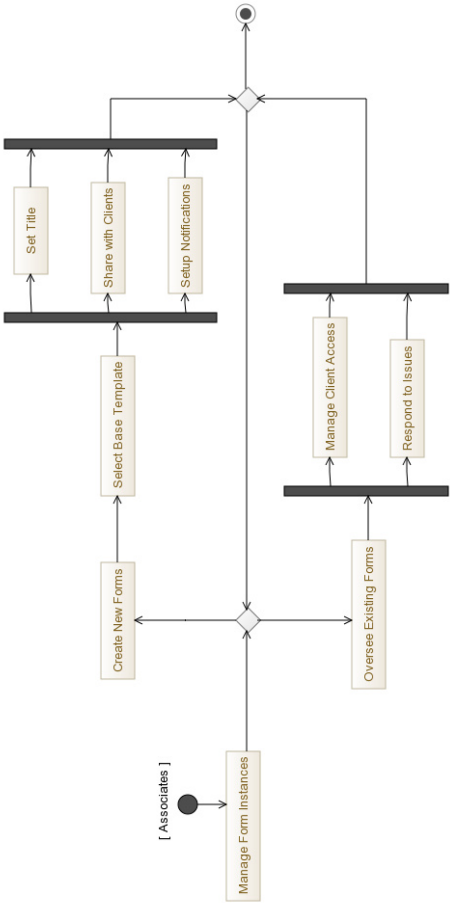


Figure 8: Associate Workflow

18.3 Admin Workflow

Admin Workflow (fig. 9) is entirely focused on the creation of new templates and management of elements within the template.

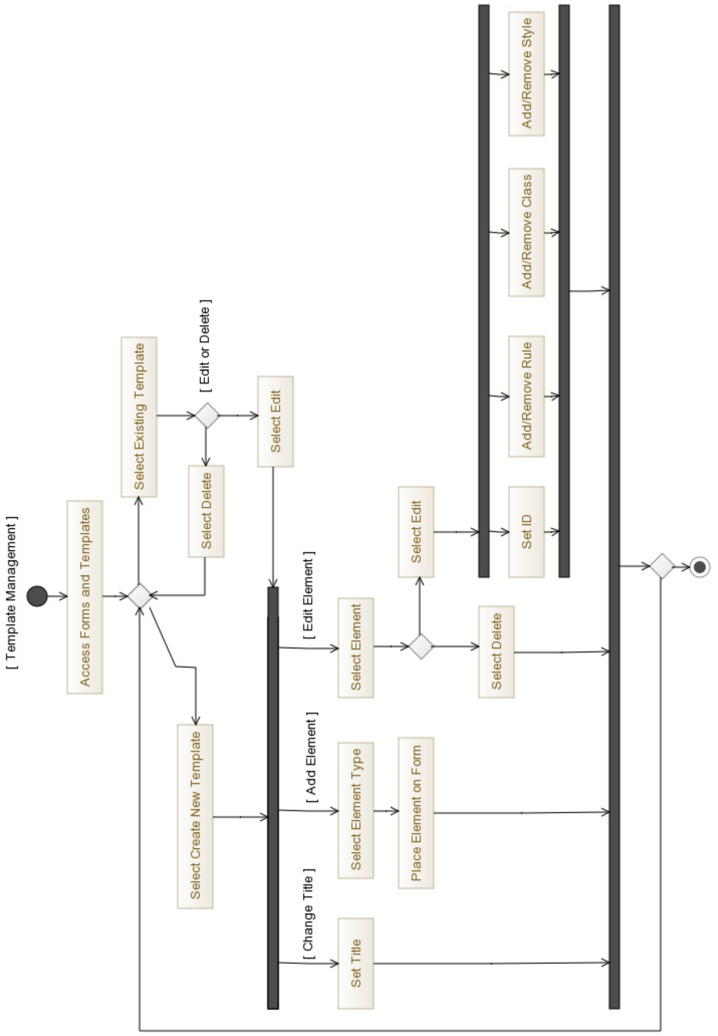
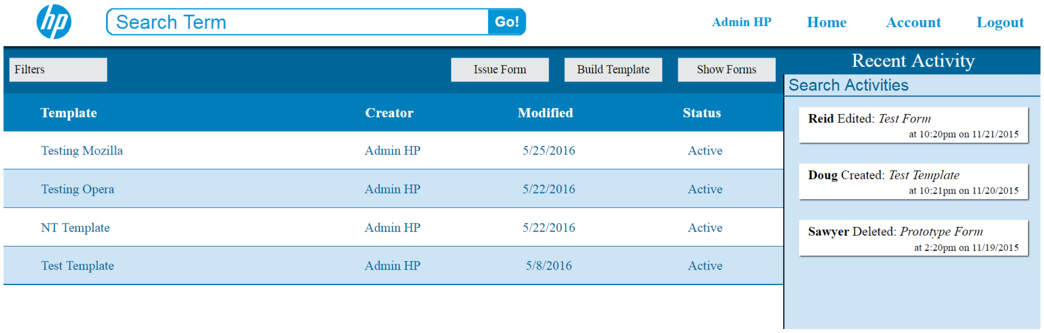


Figure 9: Admin Workflow

19 Conceptual Model

The following models illustrate how the Uniform web application will display content to users and provide tools and mechanisms to create, maintain, and complete forms and templates.

19.1 Landing Page



Template	Creator	Modified	Status
Testing Mozilla	Admin HP	5/25/2016	Active
Testing Opera	Admin HP	5/22/2016	Active
NT Template	Admin HP	5/22/2016	Active
Test Template	Admin HP	5/8/2016	Active

Reid Edited: *Test Form*
at 10:20pm on 11/21/2015

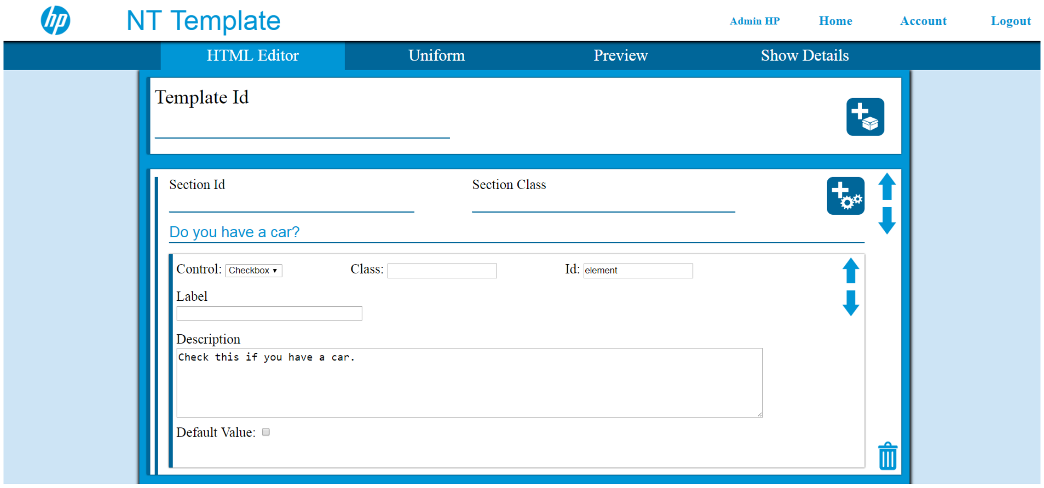
Doug Created: *Test Template*
at 10:21pm on 11/20/2015

Sawyer Deleted: *Prototype Form*
at 2:20pm on 11/19/2015

Figure 10: Landing Page

Forms and templates can be viewed through the landing page. Users will be able to apply filters on issuer, base template, modification date, status, and name. New instances and templates can be created assuming a user has the necessary permissions. Users will also be able to search directly for certain forms and view recent activities related to their account.

19.2 Template Creation



Template Id

Section Id

Section Class

Do you have a car?

Control: Checkbox ▾ Class: Id: element

Label

Description

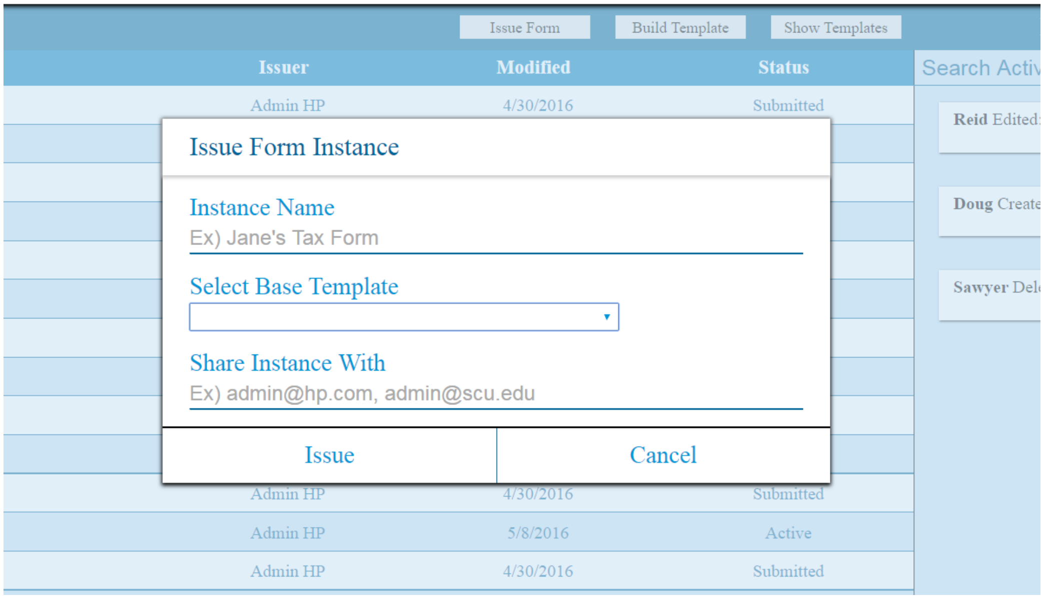
```
Check this if you have a car.
```

Default Value: □

Figure 11: Template Creation

Templates are modified through the Template editor. Template creation is managed via the construction of rows and containers which render as appropriate HTML elements. The type of control, headings, and values can be set within the template. Individual rows can be re-positioned or deleted.

19.3 Instance Creation

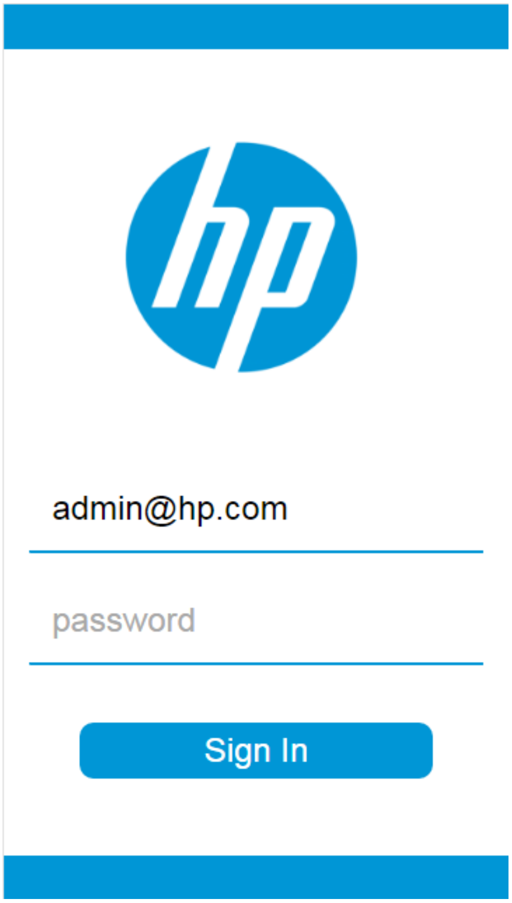


Issuer	Modified	Status
Admin HP	4/30/2016	Submitted
Admin HP	4/30/2016	Submitted
Admin HP	5/8/2016	Active
Admin HP	4/30/2016	Submitted

Figure 12: Instance Creation

Form instances can be generated through the landing page by selecting issue form to bring up the creation overlay. Admins and Associates can define instance specific information such as the name, base template, and list of authorized users.

19.4 Mobile Implementation



admin@hp.com

Figure 13: Mobile Implementation

The mobile implementation of the web application was aimed primarily at providing easy access and usability to client and associate users. This largely came in the form of a simplified, fluid interface that triggered as a result of view-port queries which tracked the size of the screen, altering active CSS styling as appropriate for the current dimensions of the user's device. Mobile menus in the upper right-hand corner of the screen became host to the majority of the actions and buttons that the desktop view hosted, leaving the majority of the display free for content to be highlighted.

On the admin side, while it was certainly conceivable that these users may wish to use their mobile devices to access a form template, the complexity of the editor far outstrips the other components of the web application and it was ultimately deemed unrealistic. Investigation into similar solutions, such as Google Forms, confirmed this was not an out of the ordinary design decision.

20 Architectural Design

To construct the Uniform web application as it has been described, the following architecture has been designed to address the major technical challenges present in this system.

20.1 Data Flow Architecture

Figure 14 shows the data flow of a form throughout its lifecycle. It is first created as a template by the system administrator, who specifies the controls on the form and the the validation logic associated with them. Once this template is ready, associates can instantiate a form template into a form instance for a certain customer. The associate would add the various users involved in this instance, such as the prospective homeowner, building inspector, selling homeowner, etc. Once the form instance is filled out and agreed upon by all parties, it is submitted to the back-end database.

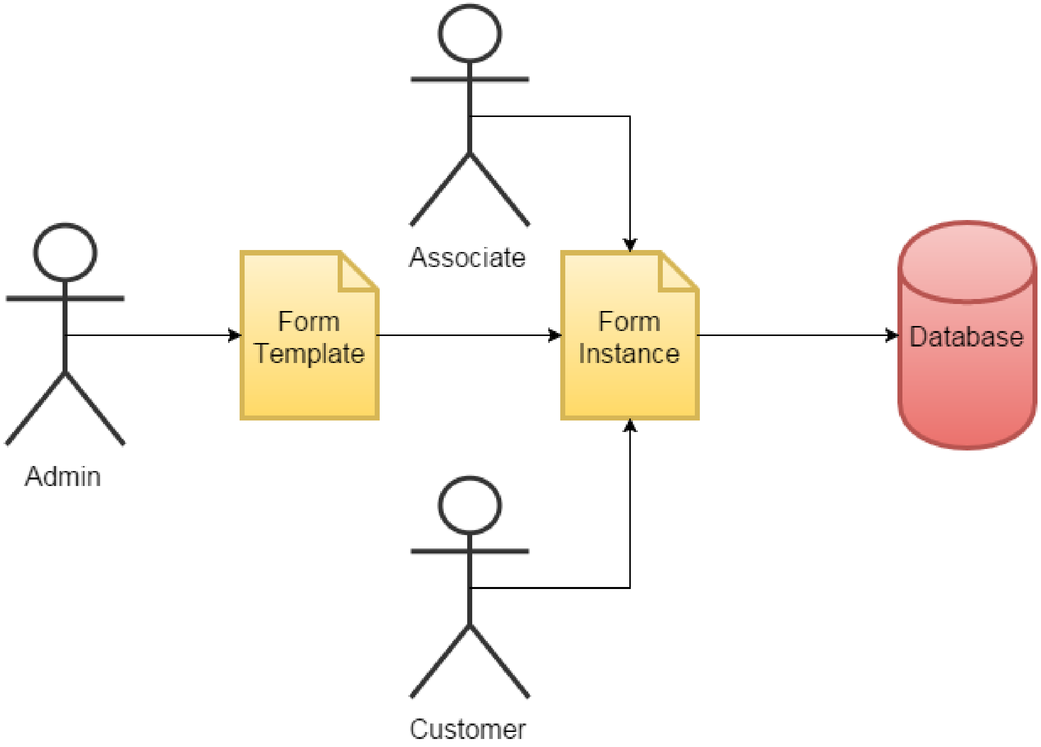


Figure 14: Form Data Flow Diagram

20.2 System Architecture

Figure 15 shows the overall architecture of the system at a technical level. The client sends a request to the server for a certain resource or action. The API parses the request, checks authentication, and determines how to respond. If the request requires some data processing, such as updating or submitting a form, then it will access the database and perform the necessary operations. If the client has requested a web resource such as HTML, JavaScript, or CSS, it will simply be returned to the client without modification. The HTML markup will provide the structure of the client's page, but the data will be passed in JSON format using AJAX calls to the API made via Angular.

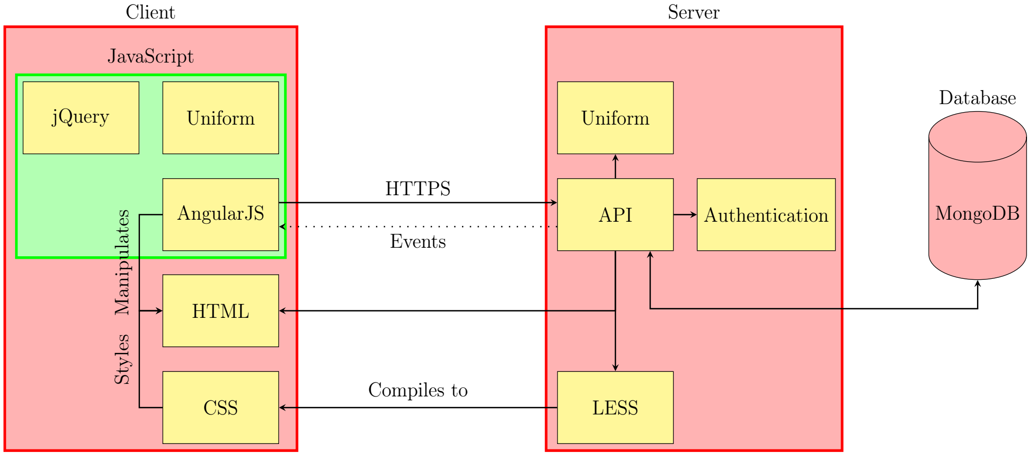


Figure 15: System Architecture

Server-sent events are implemented using Loopback.io change streams, which allow clients to subscribe to changes for a particular model or set of models. The server will then send events to the client when modifications are made to those models.

20.3 Validation

The Uniform language allows the client to validate inputs before sending the data to the server. This language lets business users can create complex validation logic for their forms quickly and easily. This logic will be implemented on the client side, however all inputs must be validated on the server side as well to ensure correct data is sent. When the server receives information, it has no guarantee that it went through the client-side checks of the web application, so it must validate the information as well.

A server-side wrapper was written for the Uniform language which will allow it to be executed on the server against the data received from the client. This enables the server to check the data received without requiring a programmer to manually code the validation logic for each individual form.

20.4 Database Schema

To provide scalable access to such a large amount of data, a NoSQL database system was used. The overall design is shown in fig. 16, displaying the relationships between various models within the application. Figure 17 shows the attributes in more detail for the form and user elements, while fig. 18 shows the attributes and hierarchy of the Control and its related models.

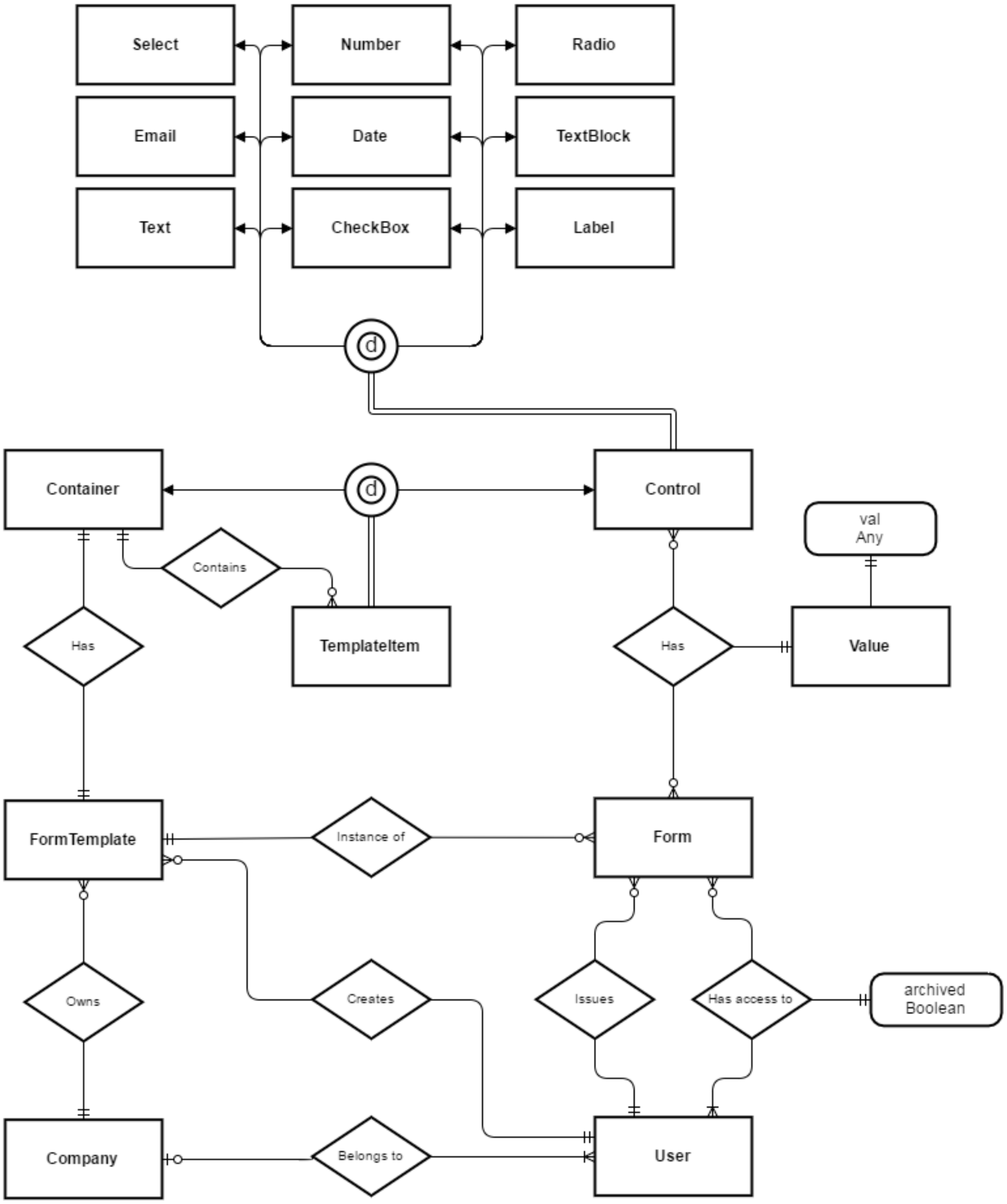


Figure 16: Database Schema (Relations)

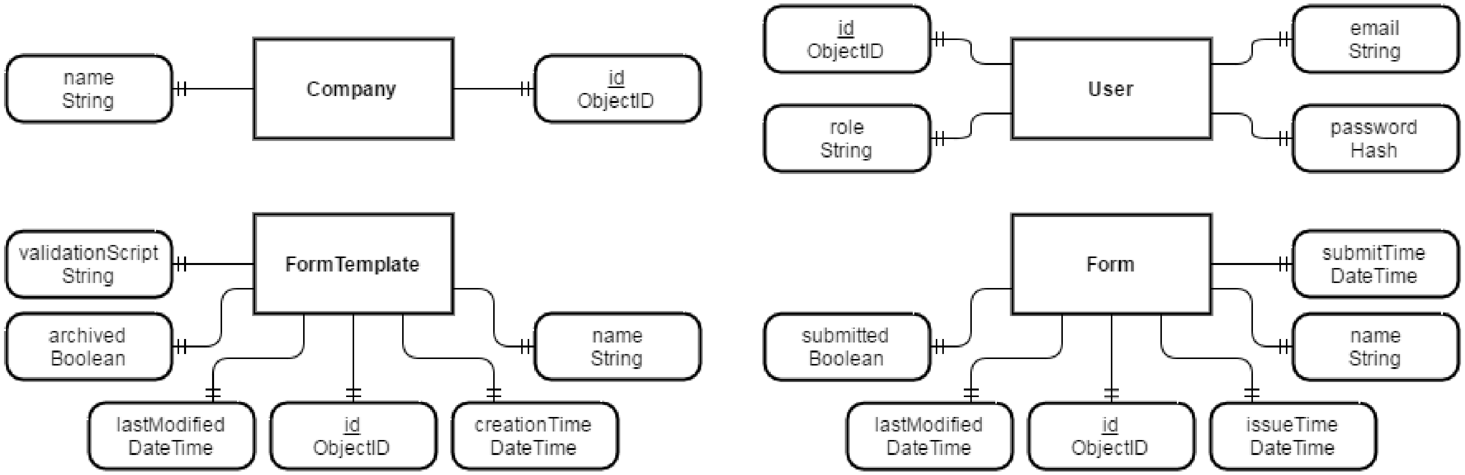


Figure 17: Database Schema (Forms)

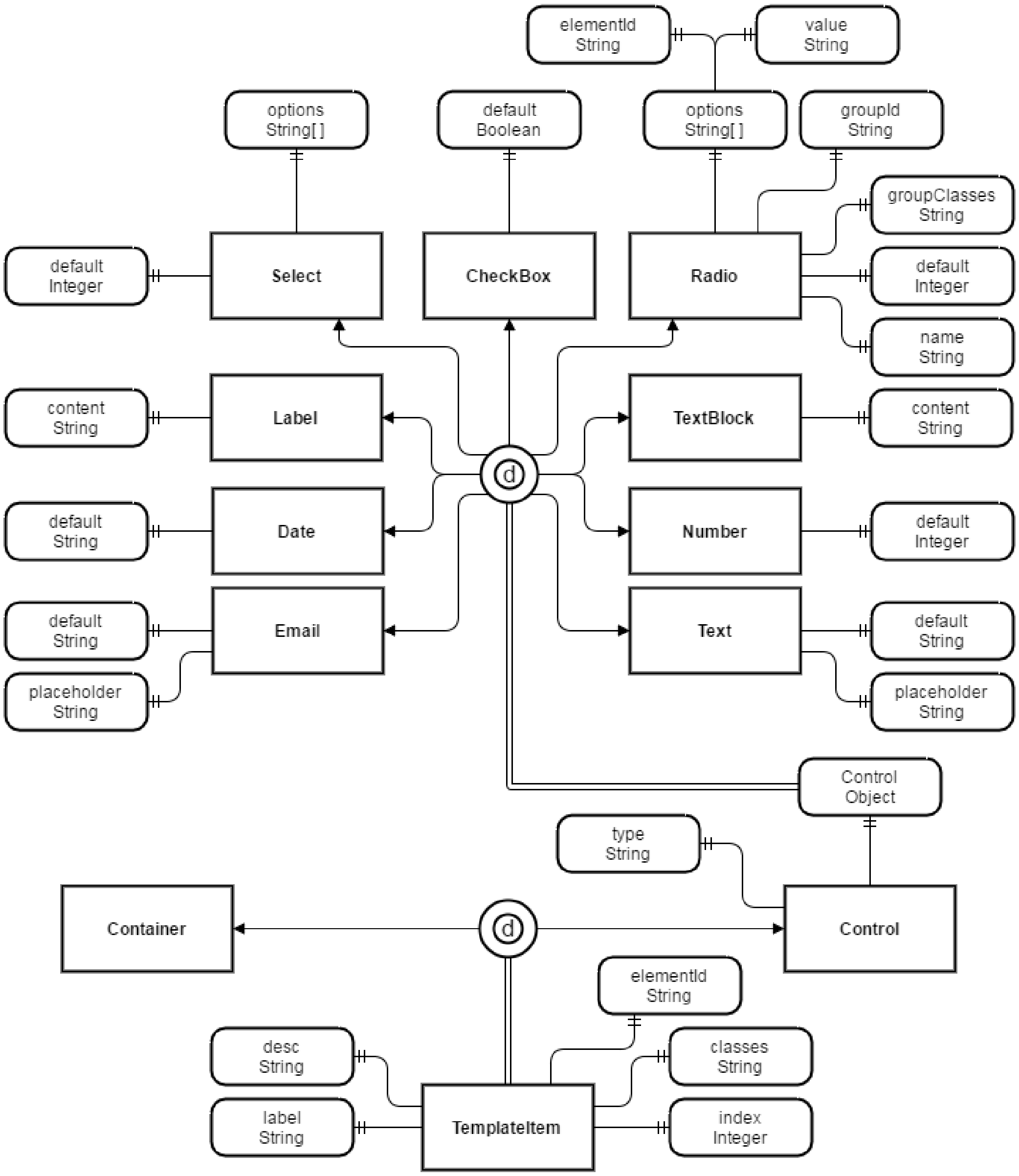


Figure 18: Database Schema (Controls)

21 Technologies Used

21.1 Server

- NodeJS: Implementation of the V8 JavaScript engine for use in server applications.
- LoopBack: Node framework designed for building REST APIs
- Jasmine: JavaScript test framework and assertion library.

21.2 Client

- HTML: Markup language used for displaying content in a standardized way across browsers.
- JavaScript: Client-side scripting language.
  - Socket.io: Bidirectional socket communication technology.
  - AngularJS: JavaScript framework focused on providing the Model-View-Controller architecture to front-end web systems.
  - Jasmine: JavaScript test framework and assertion library.
- CSS: Styling language describing how HTML elements should be displayed to the user.
  - LESS: Superset of CSS which adds extra convenience and scalability features.

21.3 Database

- MongoDB: A database schema which uses the NoSQL document model.

22 Design Rationale

22.1 Comet

One of the major requirements for this project was to allow users to work on the same form simultaneously to encourage collaboration on confusing documents. This means forms must live update for each user with changes made by others editing the form. This is a major technical challenge for which typical web technologies were not built.

In the standard HTTP model, the client sends a request to a server which returns a response. This process is completely atomic, every request begins with the client, and has only one response from the server. This system works extremely well for most web systems, but functions very poorly for systems which require a persistent connection. Within the HTTP framework, there is no supported way for a server to send an unsolicited message to a client. This means that something as simple as a chat application can become a major technical challenge because the server cannot inform the client of new messages [2, 3].

There are many hacks to get around this limitation, such as infinite iframes or long polling. Techniques like these, which allow servers to communicate to clients without an explicit request are considered part of the Comet web application model. As the web has evolved over the years, Comet applications have increased significantly and are slowly becoming a supported standard. WebSockets in HTML 5 and the Socket.io library both allow client JavaScript to easily receive events triggered by the server. Even servers have grown to support this trend such as the Jetty framework for Java and Loopback's change streams.

Due to the requirement to live update forms on the client-side, the Comet architecture was used for this system by implementing change streams exposed by the Loopback framework. This enabled smoother functionality and a better user experience than a standard HTTP request system would allow. Unfortunately, change streams are consumed via the EventSource JavaScript object which is not supported by all modern browsers, notably any version of Internet Explorer or Microsoft Edge. As of 24th May, 2016, only 80% of users have access to EventSource and would be able to use HP Forms [4].

EventSource is the feature that limits the user base of the application the most and is the single largest browser compatibility issue present. Given more time, it would have been preferable to use WebSockets or Socket.io to have a greater user base, but doing so would require special server-side code to be written, which the development timeframe did not allow. Using Loopback change streams allowed leverage of existing server-sent event logic already present in the Loopback framework and saved a significant amount of time to implement a system that will still work for a majority of users.

22.2 Single Page Application

When the web was created, client side interaction was done by redirecting the user to a new page displaying the needed information. As web applications have grown in scale, this model has become very impractical for managing and displaying data on the client [5].

Modern web applications often direct users to a single page which simply downloads a web front-end to display data and handle user interaction. Data is sent to and from the server via AJAX or WebSockets which can be rendered to the user by the client JavaScript. This maintains the logic and state of the JavaScript and only requires the client to download the data to display, it does not have to re-download the structure or web application again. Data can be more easily manipulated on the client side using this model and allows better presentation to the user. Figure 19 illustrates the differences between the traditional page life-cycle versus those of a single page application [6].

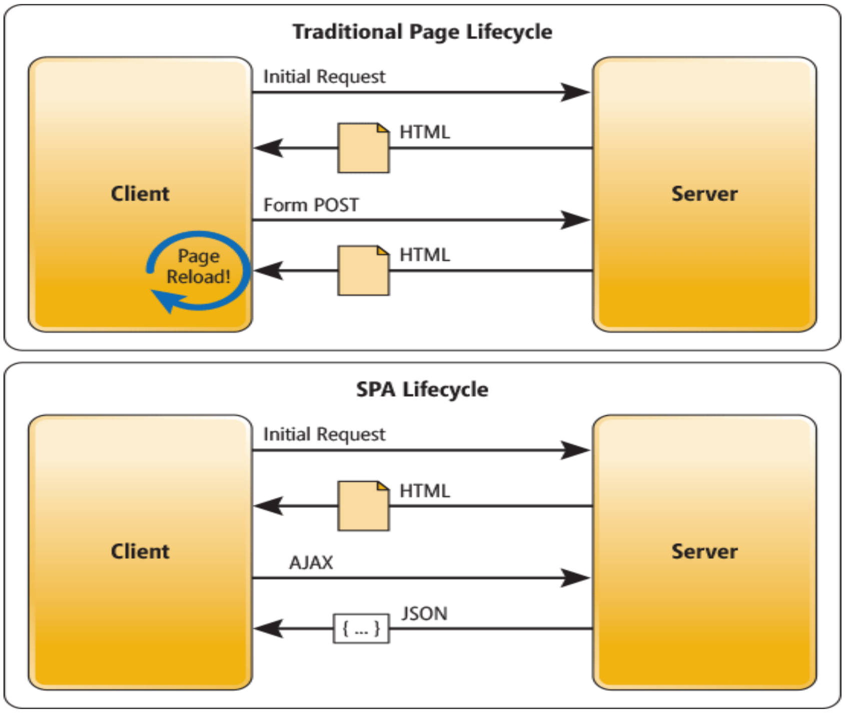


Figure 19: Single-Page Architecture Diagram

Single-page applications also benefit the Model-View-Controller architecture, as the data is stored and managed separately from its presentation. The HTML/CSS presentation layer is sent in the initial response to the client but does not contain any meaningful data. Follow-up requests from the client's JavaScript get the raw data, usually in JSON or XML format, and display it within the HTML/CSS structure. This makes it very easy to update the data without completely removing the existing page structure. A single page architecture will be used in this application to leverage these benefits and provide a modular and reusable approach to development.

22.3 Technologies

22.3.1 Server

- **NodeJS**: Node was built primarily to address the challenges which the Comet architecture presents on the server side. JavaScript's event handling system is well-suited to the back and forth communication required by Comet applications and this one in particular [7]. The Node package manger will also allow easy installation of third-party modules to assist development and reuse by the application.

- **LoopBack**: The LoopBack Node framework was used to create the server system; LoopBack offers simple API creation tools, connectors for various database schema, Comet-inspired change streams, and a framework for Model-View-Controllers which will enable the creation of a scalable application quickly and efficiently.

22.3.2 Client

- **HTML 5**: HTML was utilized to make this application widely available across all major platforms. Almost every computing device has some access to the Internet and is capable of rendering web technologies. This will allow the application to leverage such broad device support to provide access to as many users as possible with the devices they already have.

- **JavaScript**: The application uses JavaScript for its client-side scripting capabilities. Due to the Comet architecture of this system, the majority of communication is performed using events managed by the client's JavaScript rather than using standard HTTP requests sent by browsers. A number of JavaScript libraries will be used to assist in this application.

  - **AngularJS**: To ensure resuability and modularity on the client-side, the application will utilize the AngularJS framework. It will help enforce a Model-View-Controller architecture on the system's front-end logic. It also provides strong testing support, encouraging the test-driven development model. This will enable the team to improve code quality, maintain code coherence, and reduce coupling throughout the client system.
  - **Uniform**: To provide a solid user experience throughout the form completion process, this system uses the Uniform language to illustrate a practical use case. This enables a helpful interface for users and allows administrators to create forms with complex validation quickly and efficiently.
  - **jQuery**: Although AngularJS provides an implementation of jQuery lite, Uniform is dependent on jQuery specifically and is not design to run on a lite implementation. Uniform is not tested on that implementation, so the Web Application will use the full version of jQuery to avoid unexpected and difficult to debug errors.

- **CSS**: The client-side of the application takes advantage of CSS to style and present the data in a usable and user-friendly fashion to facilitate ease of use.

  - **LESS**: On the server, the Web Application takes advantage of LESS to enable more reusable and scalable styling code that is compiled down to CSS. This helps manage a large number of styles and colors, organize them logically, and enforce its own styling conventions.

22.3.3 Database

The system uses MongoDB for its database system. This is a NoSQL database implementation which was chosen over more traditional SQL options. SQL architecture is highly structured with a clear set of attributes and their relations among each other. This system will support a wide variety of form controls which have many different requirements and relations to each other. Beyond that, the initial system may only support a small subset of the controls desired, while the rest will be added in the future.

To more easily support the varying needs of each type of control and to allow easy modification and extension in the future, the system takes advantage of MongoDB's NoSQL architecture. MongoDB supports dynamic and nested schemas much better than SQL systems, allowing easy extension and modification of the database schema in the future [8, 9].

A major downside to MongoDB is the lack of transaction support, which could cause invalid database that may behave strangely with the application logic. In the future, it may become necessary to switch to more traditional SQL system to allow implementation of transactions on the database. Fortunately, the Loopback framework provides database connector with very few interface differences between them. As such, the MongoDB connector can simply be swapped out with the SQL connector with very little modification of application code, making a database switch trivial.

22.3.4 Testing

Jasmine was used as the system's test framework and assertion library because of its simplicity and ease of use. Its design encourages self-documenting tests, enabling any failing test to clearly describe exactly what went wrong, facilitating quicker fixes. Jasmine can also be used on the client and the server, simplifying and unifying the testing process for each piece.

23 Testing Plan

For any piece of software to be used in a production environment, it must be appropriately tested. This system is no different, and it will be using the following practices to ensure the level of quality that its users will expect of it.

23.1 Test-Driven Development

The team used a test-driven development process, where tests are written before the code satisfying them. This helped ensure that the team continued writing tests throughout the development process and that every piece of code written was properly tested. This did not replace traditional system testing at the end of the project life-cycle, but rather supplemented it during the implementation phase. Often, by the time the testing phase is reached, it is too late to perform necessary fixes. Early and frequent testing in the implementation phase allowed the team to fix errors as they were introduced and before they could become a significant problem. This also allowed easy regression testing to ensure that new bugs were not introduced into old code.

23.2 Unit Testing

Unit testing was performed throughout the development process as part of the test-driven development mentality. Unit tests were the majority of those written during the implementation phase and helped track down bugs that surfaced while implementing the system.

23.3 Acceptance Testing

To ensure that every part of the system is built to specification, the team met regularly with both the project adviser, Ron Danielson, and a representative from HP, Stas Neyman. In these meetings, the most recent build of the project was shown in order to receive feedback and direction as the project progressed. This ensured that everyone was on the same page throughout the entire development process and that everyone understood how each feature works and was in agreement on the result.

23.4 Security Testing

Due to the sensitive nature of many of the documents being manipulated by this system, security is major concern. To ensure a secure system, the team tested for many common software attacks against the system, including URL manipulation, cross-site scripting, database injection, and general API misuse. Since this project is not concerned with encryption of network traffic or database information, such insecurities will not be included as part of this testing. In a production system, such encryption would be expected and necessary, but this prototype will not include it to minimize scope.

23.4.1 URL Manipulation

URL manipulation is when the user directly edits the URL field in his/her browser to cause the application to move into an unexpected state. This could be as simple as putting a different ID or changing a query parameter. If the query is trusted by the server and not validated, then the server may believe that a user has different privileges than they actually have or may perform unwanted operations.

Fortunately, the application did not suffer any vulnerabilities from URL manipulation. This is because URL parameters are only used by the client to track the current navigation and are ignored by the server. A client could modify the URL to direct them to another form which they do not have access to, but any requests for that form's information would fail due to a lack of privilege. As such, the client will receive errors and will not pull any useful information.

Originally, there was one piece of data in the URL used by the server, the client's access token. This is not *necessarily* a security vulnerability because the client should not know the access token for any other users and would not have any data to replace it with. The client could replace it with an invalid token, but such a token would not grant the user any rights and would be useless. The only downside to using an access token in the query parameter is that it may be logged by the server when processing the request. Since an access token gives the same rights as the user, it should be kept as secret as their password, and a log on the server is likely not encrypted or particularly well protected. As such it is a best practice to store access tokens as cookies or headers, where they will not be logged. The application was modified to store access tokens in headers and no longer uses query parameters for this purpose.

23.4.2 Cross-Site Scripting

Cross-site scripting is when an attacker is able to put executable code inside the system's database, where a victim could later download and execute it on their computer. An attacker could easily put as an input to a form field the line:

```
<script>alert("I have you now...");</script>
```

This line would be stored into the database as that form field's value. When another user views that form, their browser would download that line and place it into their DOM tree, which would cause the browser to execute it. This can be avoided by sanitizing user inputs, for instance replacing the less than and greater than symbols with their encoded counterparts, resulting in the string:

```
&lt;script&gt;alert("I have you now...");&lt;/script&gt;
```

which would *not* execute when read by another user's browser. To achieve this, the application uses the `helmet` NodeJS library. This uses several response headers to set browsers to be much more restrictive about running code. This helps ensure that no malicious code is executed on the browser. Currently no attempt is made to sanitize data in the database and it is up to the client to only use valid inputs.

There are other ways of performing cross-site scripting such as injecting code into a JavaScript `eval()` call or other methods. All known cross-site scripting attacks have been patched and no other vulnerabilities are currently known. However, due to the complex nature of this kind of attack and the number of entry points, it is best that the application be reviewed by a security expert before being put into production.

23.4.3 Database Injection

Database injection is an attack where a user sends a script written the database language (SQL, Mongo, etc.) and is able to get it to execute on the database. This is usually a result of the server concatenating unsanitized user data with database code.

Fortunately, the application is not vulnerable to database injection due to the Loopback framework. Loopback uses object-relational mapping, meaning that application code does not directly run or otherwise touch database queries. Instead, Loopback exposes objects to the application code which have functions that perform various database interactions. This adds a layer of abstraction between application code and the database, removing any place where a malicious user could inject database code. As such, the web application does not suffer from database injection vulnerabilities and none needed to be patched.

23.4.4 API Misuse

API misuse is when a malicious client sends invalid data to the server. This invalid data may be as simple as requesting a form the user does not have access to. The server must see identify this and deny the request. Unfortunately, there are many ways an API can be misused and it is impossible to be certain that every case is handled. Consider the following code:

```
// Get all threads which belong to the given Form
Thread.get = function (formId) {
    return Thread.find({
        // Find every thread belonging to the given form
        where: {
            FormId: formId
        }
    });
};
```

When invoked as expected, this works correctly and returns all threads belonging to the given form. If that form has three threads, then it will return an array of length three.

```
var formThreads = Thread.get(1234);
console.log(formThreads.length); // 3
```

When invoked without a parameter however, things become a little more interesting.

```
var formThreads = Thread.get();
console.log(formThreads.length); // 100+ !!!
```

Now the function returned an array of *every* thread, regardless of what form owns it. This is because when the function was called with no arguments, the `formId` parameter was set to undefined. This means that `Thread.find({ ... })` was called with `FormId:undefined`. In JavaScript, setting a key to an undefined value is the same as omitting the key entirely, making this the same as running `Thread.find({ where: { } })`, with an empty object! In Loopback, calling `find()` on a model without a `where` query will return *all* its instances in the database.

If a user simply did not pass an ID, then they would suddenly receive *every* thread in the database, regardless of whether or not they had access to it. It can be argued that this is a form of database injection, but for the purposes of this paper, it will be classified as a form of API misuse. This is clearly a major security vulnerability, and it needs to be handled. Unfortunately, addressing it is not easy, as every API has different arguments and reacts differently depending on the format of each. The large variety of values and types that can be sent makes it difficult to fully test and show that no vulnerabilities exist.

The team has made a best effort to ensure that all edge cases are handled, and much of the server's unit testing is explicitly aimed at ensuring that invalid data is not processed. Unfortunately it is very difficult to be sure that no holes exist, and the application should be reviewed by a security expert before being put into production.

23.5 User Testing

Where the web application was concerned, user testing was primarily focused on ensuring a functional, visually appealing, and most of all usable user interface. User tests saw testers simulating each of the three system roles and undertaking appropriate tasks for each. Feedback on the learnability and memorability of the tasks was recorded using a combination of scale based answers and free-response. This feedback was used to drive appropriate design changes, specifically these changes focused on providing more accurate and in-depth system feedback and making the user interface more easily navigable. Additionally, general-purpose feedback statements were recorded and used to determine the focus of remaining front-end development time.

23.6 Browser Testing

To ensure that all audiences are capable of using the web application, browser testing was undertaken on the final system build on all major web browsers. By verifying that the web application provides full functionality on these browsers, the team was able to ensure that no user would be unable to access the system assuming reasonable access to appropriate technology.

24 Test Results

24.1 User Testing

User testing was performed by a group of eight Santa Clara University students. As they progressed through testing, the students were asked a variety of question focused on rating the difficulty of performing the actions of each role; client, associate, and admin.

24.1.1 Client

As clients, students were asked to fill out a form correctly and to submit this form. Of the eight students, three were unable to successfully submit their form. For one of these students, the failure was attributed simply to the system, thus making it a negative result. For the other two, however, the failure was attributed to the testing guidelines giving unclear directions as to what was necessary to fill out the form.

While most of the students found the client portion easy to achieve, feedback was focused on providing a clearer submission process to forms. This led to the implementation of a static submission button automatically located at the bottom of all forms and pop-up messages indicating successful submission.

24.1.2 Associate

As associates, students were asked to issue a form to a client. This section was largely considered the easiest section, with no student rating the experience as difficult and only one student having an issue with locating the correct buttons in the interface.

Feedback here primarily focused on clarifying the labels of specific buttons on the landing page. This led to the implementation of larger, more distinct buttons, a simplified navigation menu, and an indication of whether or not users were viewing forms or templates.

24.1.3 Admin

As admins, students were asked to create the car form following a set of instructions. All students were able to complete this section, with no student rating the experience as difficult and only two students having an issue with locating the correct buttons in the interface.

Feedback for this section focused around clarifying the fact that data was being updated and saved automatically by the web application. This led to the implementation of update indicators located atop any actively updating elements in the template editor.

24.1.4 Browser Testing

Desktop

Browser testing was performed without issue on the following desktop browsers

- Chrome
- Mozilla Firefox
- Safari
- Opera

Of the major desktop browsers, Internet Explorer was excluded due to the web application's reliance on EventSource updates for real-time updates and communication. Internet Explorer does not support EventSource and so the web application cannot run properly on that platform.

Mobile

Browser testing was performed without issue on the following mobile browsers

- Chrome
- UC Browser

Browser testing was also performed using the IOS browser with partial success. While all functionality was correct on the IOS browser, a number of CSS features rendered incorrectly, leading to limited functionality in some cases. In order to deal with these inconsistencies, the CSS of certain elements was adjusted, making use non-gradient color values where possible as these were largely the cause of display inconsistencies.

25 Install Guide

To run the web application on a new server, a few steps must be taken:

1. Install system dependencies
2. Install application dependencies
3. Test server (optional)
4. Run server

25.1 Installing system dependencies

System dependencies include NodeJS and MongoDB. These can be set up individually and even on different computers.

25.1.1 NodeJS

The process for installing Node will differ depending on the operating system of the server. Node must be installed, and it should install NPM (Node Package Manager) with it. Both Node and NPM should be on the system's path. This can be tested by running:

```
$ node -v
v4.4.2
$ npm -v
3.3.12
```

This system was developed and tested using Node 4.2.2. While future versions of Node should not break it, it is impossible to foresee such issues.

25.1.2 MongoDB

MongoDB can installed from the Mongo website and must be running before the server is started. If the server and database will be on the same machine (probably as a development or test server), then simply running

```
$ mongod
```

should be sufficient. If the server and database will be on different machines, then the server must be configured to connect to the remote database. This can be done by editing "server/datasources.json". This file should have in the connection information for the database in the "db" key value. The data that should be put here depends on the database connection information, but Loopback's documentation on datasource configuration can be used as reference [10].

25.2 Installing application dependencies

Installing application dependencies is very easy with NPM. Simply open a terminal/command prompt window in the directory of the source code. If building a development or test system, run

```
$ npm install
```

This will install all packages which the web application is dependent on for both production dependencies and development tools.

If building a production system to be used by the public, run

```
$ npm install --production
```

This will install all production dependencies but omit development tools and their dependencies which are not necessary on production systems. This may throw some warnings that certain `grunt-*` modules could not be found. That is expected as those modules are only needed for development systems.

Once completed, a `node_modules` folder should have appeared in the current directory which contains all the application's dependencies. The dependency tree should also be printed to the console to verify that it was successful.

25.3 Testing the server

When installed on a new system, the server should be tested to ensure that there are no obfuscated bugs that have been introduced. This can only be done on systems that were installed with

```
$ npm install
```

rather than

```
$ npm install --production
```

as the testing tools are only installed on development systems.

Tests can be executed by running

```
$ npm test
```

This will perform all server-side and client-side tests, so there should be two sections with two sets of results. All tests should pass for both test suites, if one or more does not, then there is a problem in the set up of the system.

25.4 Run server

If the server passes all its tests, then it is ready to be started. Simply run

```
$ npm start
```

to execute the server. It should print that the web server is listening on a certain IP and port. Those values can be changed by editing `server/config.json`.

Using a browser and navigating to the server's IP at the listening port should now display the application home page or login page. The server can be stopped by pressing `Ctrl+C` in the running terminal.

If running on a production system, it is wise to set the environment variable `NODE_ENV` to "prod" in the same terminal before starting the server. This is done with

```
Windows: $ set NODE_ENV=prod
Unix:    $ NODE_ENV=prod
```

which sets Node to use a production environment and some settings are different. For instance, the application will use `server/datasources.prod.json` instead of `server/datasources.json`. This is used to keep development and production database configurations separate, as the production environment probably uses an external database, while a development system likely has its own internal database. Note that this environment variable must be set *before* running the server within the terminal it is started. If the server is killed and restarted, it is wise to run the `NODE_ENV` command again, or configure the system to always have the correct environment set.

25.5 Common errors

25.5.1 Address in use

If the server is unable to start and displays an error message similar to the one below, then the port it is trying to bind to is taken by another process.

```
events.js:141
     throw er; // Unhandled ’error’ event
     ^

Error: listen EADDRINUSE 0.0.0.0:3000
...
```

If the server was running successfully earlier, then it is likely that the previous instance is still running and blocking the new instance from taking the port. Look for a running Node process owning the port and kill it. This should free the port for the new instance.

If the server has not been able to run successfully, then it is likely that another application has the port. Either find the application owning the port and kill it, or try changing the port that the Web Application binds to in `server/config.json` to make it use an open port.

25.5.2 Database connection error

If the server starts successfully but displays a “connected fails” message similar to the one below, then it is unable to connect to the database.

```
Web server listening at: http://localhost:3000
Connection fails:  { [MongoError: connect ECONNREFUSED
    127.0.0.1:27017]
  name: 'MongoError',
  message: 'connect ECONNREFUSED 127.0.0.1:27017' }
It will be retried for the next request.
```

If this was on a development machine running the database locally, then it is likely that the database was not started. Find the MongoDB installation directory and run the `mongod` command to start the server. Verify that the port the database listens to is the one that the application is using.

If this was on a server connecting to a remote database, then it is more likely that the connection information is wrong. The information in `server/datasources.json` should be verified. If the database IP and port do not match the configuration put in the datasources file, then it is likely using a different configuration file. If `NODE_ENV` is set to “prod”, then the system will actually use the configuration present in `server/datasources.prod.json`, and that is the file that must be edited.

25.5.3 Home page perpetually loads

If the server starts, but visiting the home page shows a loading icon that never disappears, then likely the current user profile is invalid. If the browser caches an access token, and then the database deletes that token or its associated user, then it can fail to load. Simply click the “logout” button and then log in again to reset the token on the client.

26 User Guide

The Uniform web application has been designed for ease of use, but for those unfamiliar with the system, the following instructions can guide a group of users through the implementation of a form template and the production and completion of a form instance.

26.1 Accessing the Application

Accessing the application requires user authentication performed at the landing page. Users attempting to access any portion of the application who have not been authenticated will automatically be redirected to the following URL:

yoursite/#/login.

At the landing page, a user need only input the email address and password linked to their account in the specified fields to authenticate their-self and access the application.

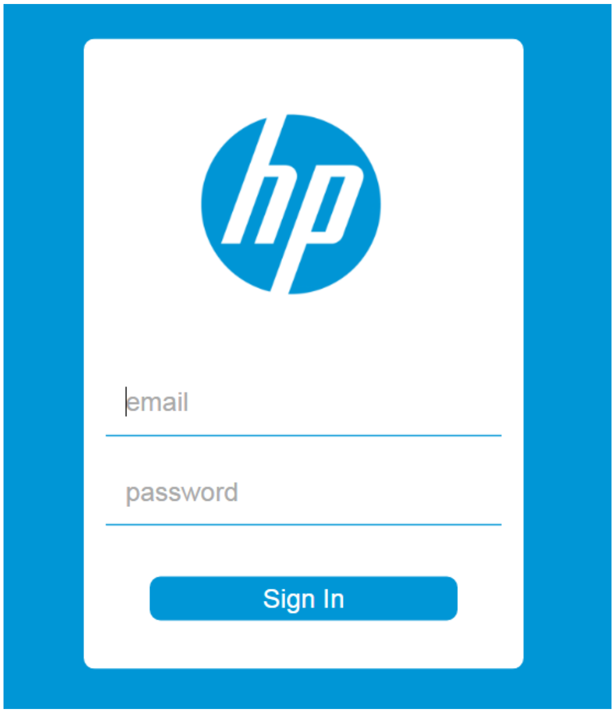


Figure 20: Accessing the Application

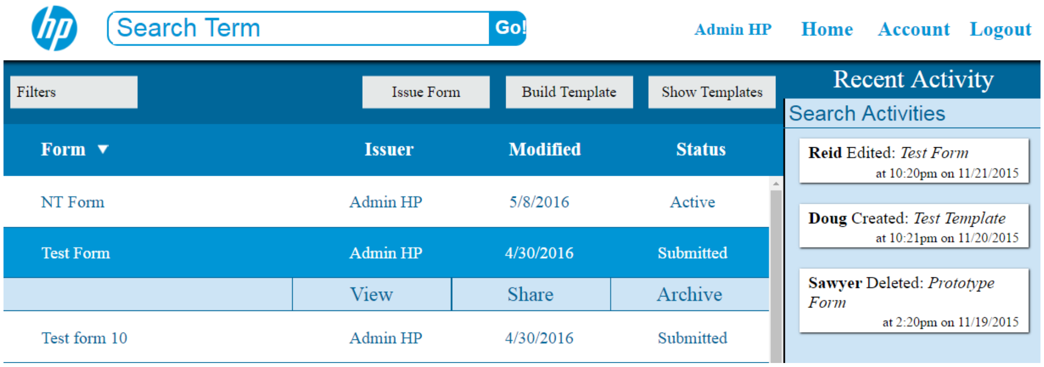
26.2 Navigating the Landing Page

Upon successful authentication, users will be redirected to the landing page. From this page, users can view the form instances and form templates available to them, access these objects, and, provided they are an associate or admin, create and issue new form templates and form instances.

26.2.1 Form Instances

By default, users accessing the landing will be presented with the comprehensive list of their available form instances. These will take the form of a large list that can be sorted by name, issuer, last modification date, or status by clicking the bold, white column headers that indicate these displays. A specific form instance, when clicked, will expand to offer user specific actions, such as the ability to view that instance or to remove it entirely all of which can be activated with a single click.

Expanding on this functionality, various toggleable filters can be set by selecting the button labeled 'Filters' and choosing from the available options all of which are enabled by default. Furthermore, users can search for a specific form instance by keyboard using the search menu located at the top of the screen.



Form ▼	Issuer	Modified	Status
NT Form	Admin HP	5/8/2016	Active
Test Form	Admin HP	4/30/2016	Submitted
Test form 10	Admin HP	4/30/2016	Submitted

Reid Edited: *Test Form* at 10:20pm on 11/21/2015

Doug Created: *Test Template* at 10:21pm on 11/20/2015

Sawyer Deleted: *Prototype Form* at 2:20pm on 11/19/2015

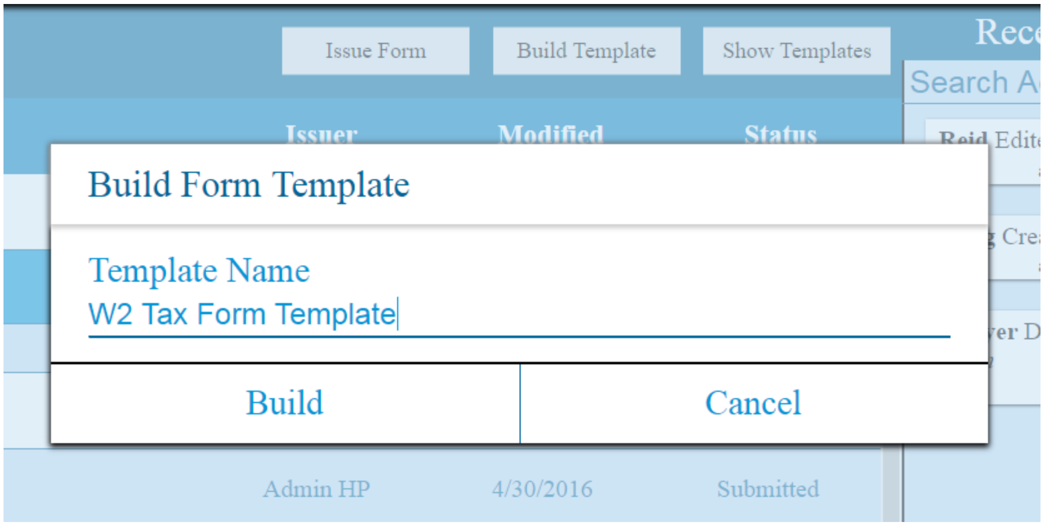
Figure 21: Form Instances

26.2.2 Form Templates

In order to view available form templates, a user requires associate or admin credentials. Assuming a user fills one of these roles, they may toggle their view between instances and templates by selecting the button labeled 'Show Templates' located just above the status column. All functionality available for form instances is replicated for form templates with the exception of the ability to directly share a template and certain filters which are unnecessary.

26.2.3 Creating a Form Template

To create a form template, users need admin credentials. Assuming a user is an admin, they can create a template by selecting the button labeled 'Build Template' just above the modified column. Upon selecting this option, users will be presented with a simple pop-up that asks them to title their template. After providing a title, users can select 'Build' and they will shortly be redirected to the template editor which will be covered later in the guide.



Build Form Template

Template Name

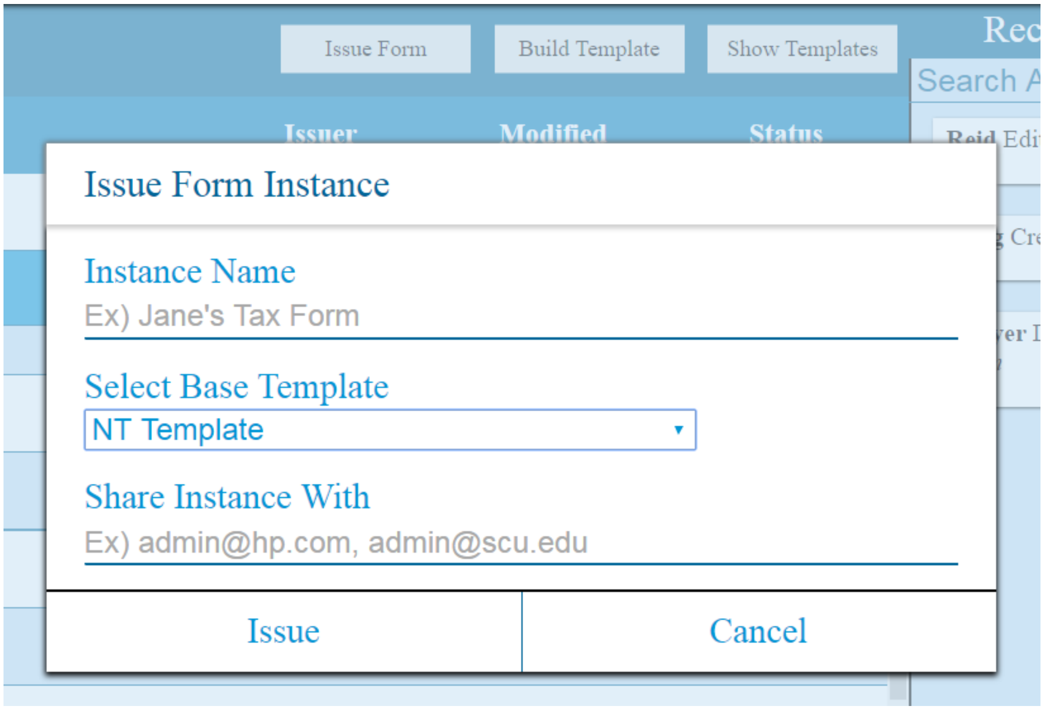
W2 Tax Form Template

Admin HP 4/30/2016 Submitted

Figure 22: Building a Template

26.2.4 Issuing a Form Instance

To issue a form instance, users must be either an associate or an admin. Assuming a user fills one of these roles, they may issue a new form instance by selecting the button labeled 'Issue Form' located just above the issuer column. Upon selecting this option, users will be presented with a simple pop-up that asks them to title their instance, select a base template from a dropdown list of the templates available to them, and input a comma separated list of user email addresses whom they wish to share this instance with. After providing this information, users can select 'Issue' and they will soon see their instances list populate with the new form instance.


Issue Form Instance

Instance Name

Ex) Jane's Tax Form

Select Base Template

NT Template ▾

Share Instance With

Ex) admin@hp.com, admin@scu.edu

Figure 23: Issuing a Form Instance

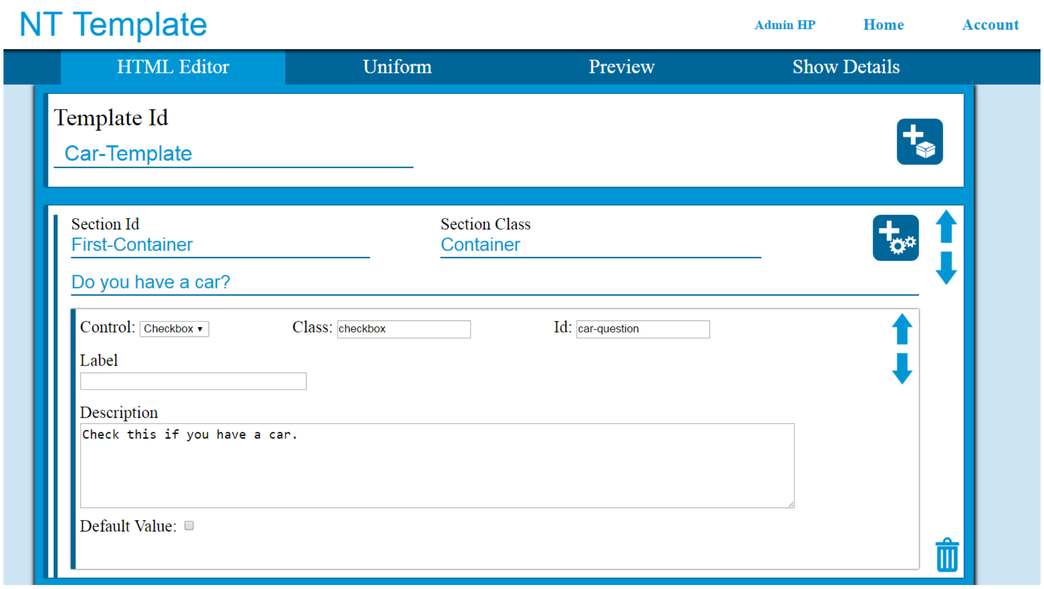
26.3 Using the Template Editor

The template editor can be accessed in one of two ways. First, a user is automatically redirected to the editor upon creation of a new template. Alternatively, a user can access the editor by selecting a form template in the landing view, and selecting the edit option if it is available.

The template editor is comprised of three main views, the HTML editor, the Uniform editor, and the instance preview.

26.3.1 The HTML Editor

The HTML editor is designed to replace the otherwise manual task of creating a web form's structure using HTML. This editor allows users to create groups of containers, and controls within these containers, that will later be turned into structured HTML when an instance is generated.



Template Id
Car-Template

Section Id
First-Container

Section Class
Container

Do you have a car?

Control: Checkbox ▾ Class: checkbox Id: car-question

Label

Description
Check this if you have a car.

Default Value: □

Figure 24: The HTML Editor

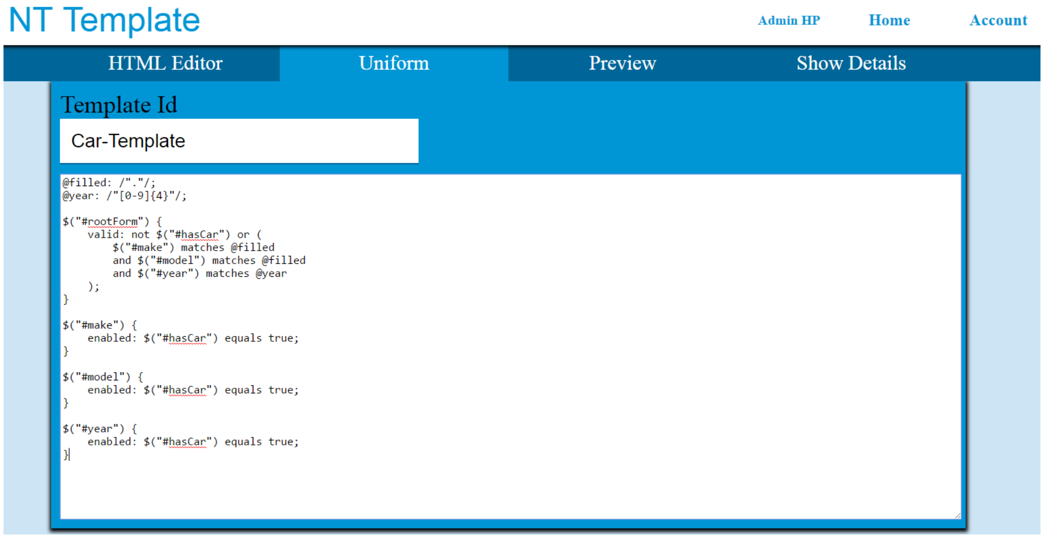
Initially, a new template will consist of the root template and a single container. Located at the top of the editor, the root template section allows users to create additional containers by selecting the button with a plus icon and a box icon, as well as set the HTML id of the entire form.

Beneath the root template, the default container will contain a single control, indicated by the indented blue line bordering it. Users can add additional controls to a container by selectin ghte button with a plus icon and a gears icon, as well as set the HTML id and class of the container, in addition to specifying a header for the form section that the container represents.

Controls are slightly more complex, with users being able to specify the type, such as text-box or check-box, default values, and attached descriptions in addition to the HTML id and class of the control. Both controls and containers can be deleted by selecting the trash-bin icon located in the bottom right corner of these objects, although the last control of a container cannot be deleted.

26.3.2 The Uniform Editor

The Uniform Editor offers a simple location to attach Uniform code to a form template. Users are able to use the large textbox to directly input Uniform or copy it in from an external editor.



Template Id

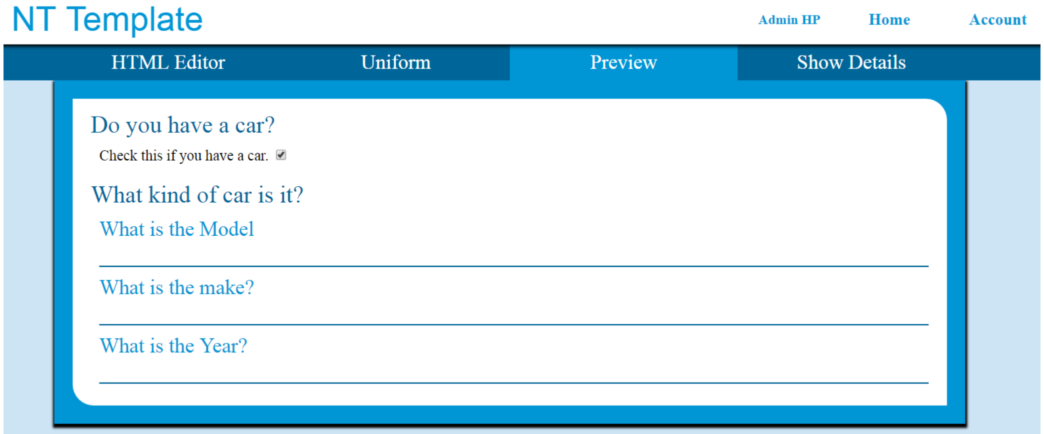
Car-Template

```
@filled: /"."/;
@year: /"[0-9]{4}"/;

$("#rootForm") {
    valid: not $("#hasCar") or (
        $("#make") matches @filled
        and $("#model") matches @filled
        and $("#year") matches @year
    );
}

$("#make") {
    enabled: $("#hasCar") equals true;
}

$("#model") {
    enabled: $("#hasCar") equals true;
}

$("#year") {
    enabled: $("#hasCar") equals true;
}
```

Figure 25: The Uniform Editor

26.3.3 The Instance Preview

Lastly, users can preview the base form instance that their template will generate through the preview tab which reflects the default values and structure they have set in place in the HTML editor.



Do you have a car?

Check this if you have a car. ☑

What kind of car is it?

What is the Model

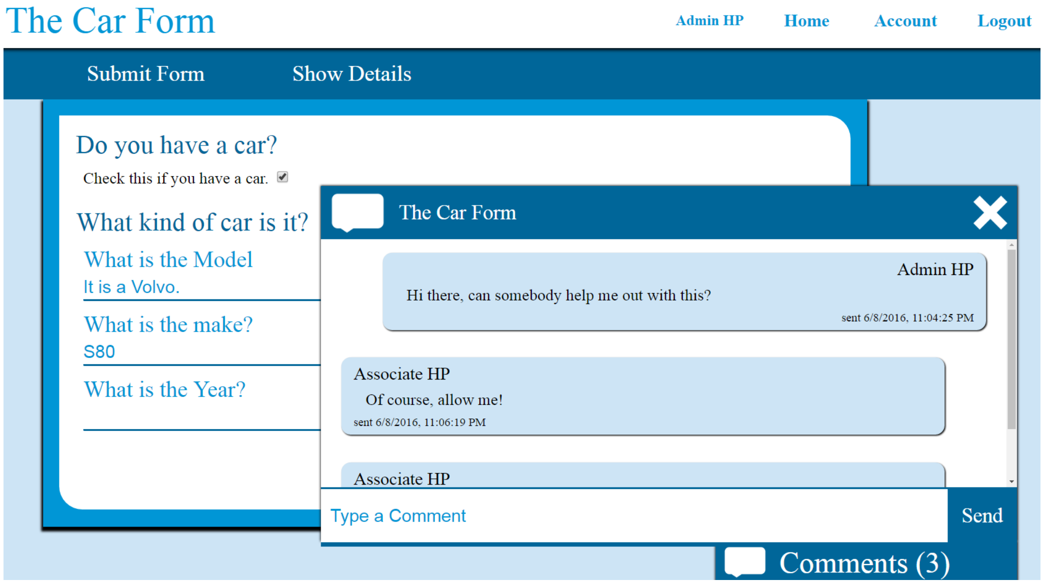
What is the make?

What is the Year?

Figure 26: The Instance Preview

26.4 Viewing a Form Instance

Any user who has received access to a form instance may view this instance by selecting the instance on their landing page and selecting the view action. Upon undertaking this action, users will be redirected to the form view which offers users the ability to collaboratively complete the form instance and communicate with each other through a chat style comment system.



Do you have a car?

Check this if you have a car. ☑

What kind of car is it?

What is the Model

It is a Volvo.

What is the make?

S80

What is the Year?

Admin HP

Hi there, can somebody help me out with this?

sent 6/8/2016, 11:04:25 PM

Associate HP

Of course, allow me!

sent 6/8/2016, 11:06:19 PM

Associate HP

Figure 27: Viewing a Form Instance

26.4.1 Editing a Form Instance

All users with access to a form instance have equal right to edit it by simply selecting the pertinent sections of the form and inputting appropriate data. Forms can be altered at any point up to their successful submission at which point they are permanently frozen in the state which they were submitted in. If a section of the form cannot be selected and is otherwise darkened or grayed out, it indicates that the Uniform code governing this form will not currently allow users to access this section based on the current form values.

Part III
Development

27 Tools

27.1 Git

Version controls software was used to enable the development team to work collaboratively on the same code base without interfering with each others' modifications. Team members were be able to work independently on individual issues while still maintaining easy merging with each others' work. Git's popularity over other version control systems gives it a strong community with a significant amount of support and documentation available for reference. It also supports offline work and enables quick and easy branching, giving it a significant advantage over other source control software.

27.2 JetBrains WebStorm

WebStorm was chosen as the integrated development environment for this project for a few reasons. It has some of the best syntax highlighting and auto-completion for JavaScript, has Git integration built into the tool, allows test running in editor, and live compilation of code as it is written. Most importantly, it is cross-platform, a critical feature, as the team used both Microsoft Windows and Apple Macintosh for development.

27.3 Atlassian JIRA

The JIRA management system was selected to keep track of the project's backlog, organize sprints, and perform code review. It supports many Agile practices out of the box which were utilized for this project. The team was able to receive access to a paid instance of JIRA courtesy of the project's sponsor, Hewlett-Packard.

27.4 Source Code Hosting

The Git source code repository was hosted on BitBucket throughout development. There are two major reasons for choosing BitBucket over competing products. Firstly, it supports viewing commits and leaving comments on them which will help facilitate the code review process. Secondly, it integrates with Atlassian JIRA and easily maps commits to the issues they relate to, helping the team track commits and issues together. However, BitBucket has much lower notoriety and fewer features regarding open-source development. As such, the Uniform Validation Language was migrated from BitBucket to GitHub at the end of this development cycle. GitHub provides the increased visibility such a project requires to gain support and interest from the open-source community. The future of the language is dependent on the response from the community, so it needs the highest visibility and notoriety that it can get, GitHub is the first step to entering the open-source community. The web application will continue to be hosted on internal HP repositories as necessary throughout its life cycle.

28 Development Problems and Solutions

28.1 Interface Redesign

As the development of the web application progressed numerous revisions and outright redesigns of the user interface became necessary. For instance, early designs of the front-end relied on a left side action bar. It was intended that this bar would display contextually appropriate actions to the user dynamically in reaction to actions taken by the user. However, as development progressed it became clear that new users were not connecting their actions to changes in the sidebar, and this, in turn, harmed overall system learnability. Moreover, many of the actions originally intended to reside in the left sidebar were more appropriately embedded in the page content itself, leading to a large portion of the sidebar remaining empty. After the sidebar was removed, a large amount of on-screen real-estate became available, triggering further alterations to the interface.

28.2 Development on Local-Host Masked Latency

Development of the web application was performed on a server running locally on development machines. The direct result of this was that issues of latency between the client and server were not made apparent because connections to localhost are effectively instantaneous. Once a development build was uploaded to a remote Heroku App environment a number of cases were found that some event-based functionality degenerated in more realistic conditions where our server was not able to keep up with requests triggered by these events. The system was updated to handle slower, more realistic connections, without causing an annoying user experience.

28.2.1 Value Update Race Conditions

The most endemic issue through development builds was race conditions resulting from conflicting value updates and server responses. In initial builds, the web application performed general updates when it received data from the server. For simplicity's sake, these updates were not targeted at the specific data that had been changed, but rather the template or form as a whole. When running on localhost, this was fine and it did not cause any difficulties for the interface because it could update fast enough. Only after uploading the build to a remote server was it obvious that this "lazy updating" significantly impacted the user experience by overwriting user input with old values as they were typing it. This eventually lead to a rewrite of the update code to target individual components only.

29 Development Timeline

To accomplish these goals in accordance to Santa Clara University's expectations, a development timeline is presented in fig. 28, fig. 29, and fig. 30.

The team followed various Agile guidelines. Work was done in weekly "sprints," with the goal of creating a "potentially shippable product" each week. Each sprint began by estimating and planning the work to be done. As work was done, the development team met regularly to discuss progress made, current work, and impediments to progress. At the end of each sprint the team would meet to discuss the current state of the project, how it should change moving forward, and how the backlog should be re-prioritized. The team also discussed the process in place, any weaknesses or problems that have been identified, and improvements which could be incorporated moving forward. JIRA enabled the team to enforce this process by clearly defining issues, sprints, estimates, and even providing metrics such as burndown charts. With this process, the team was able to ensure that the product remained on schedule.

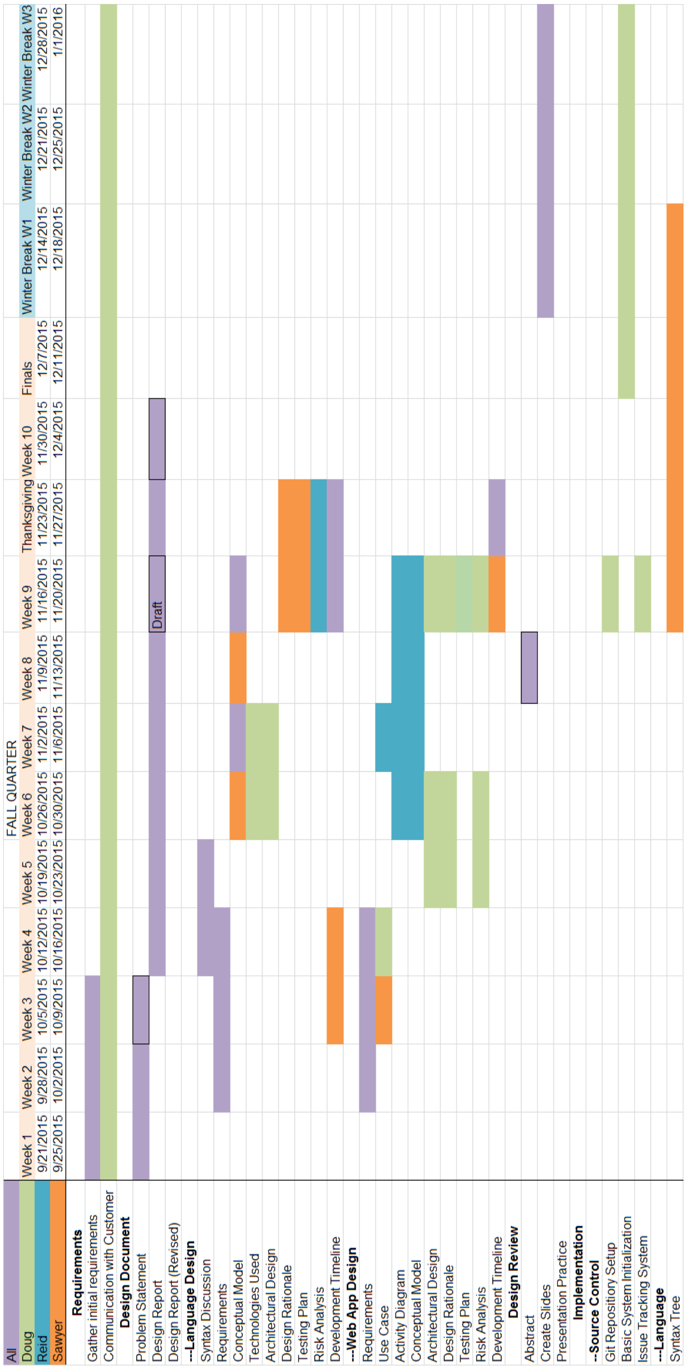
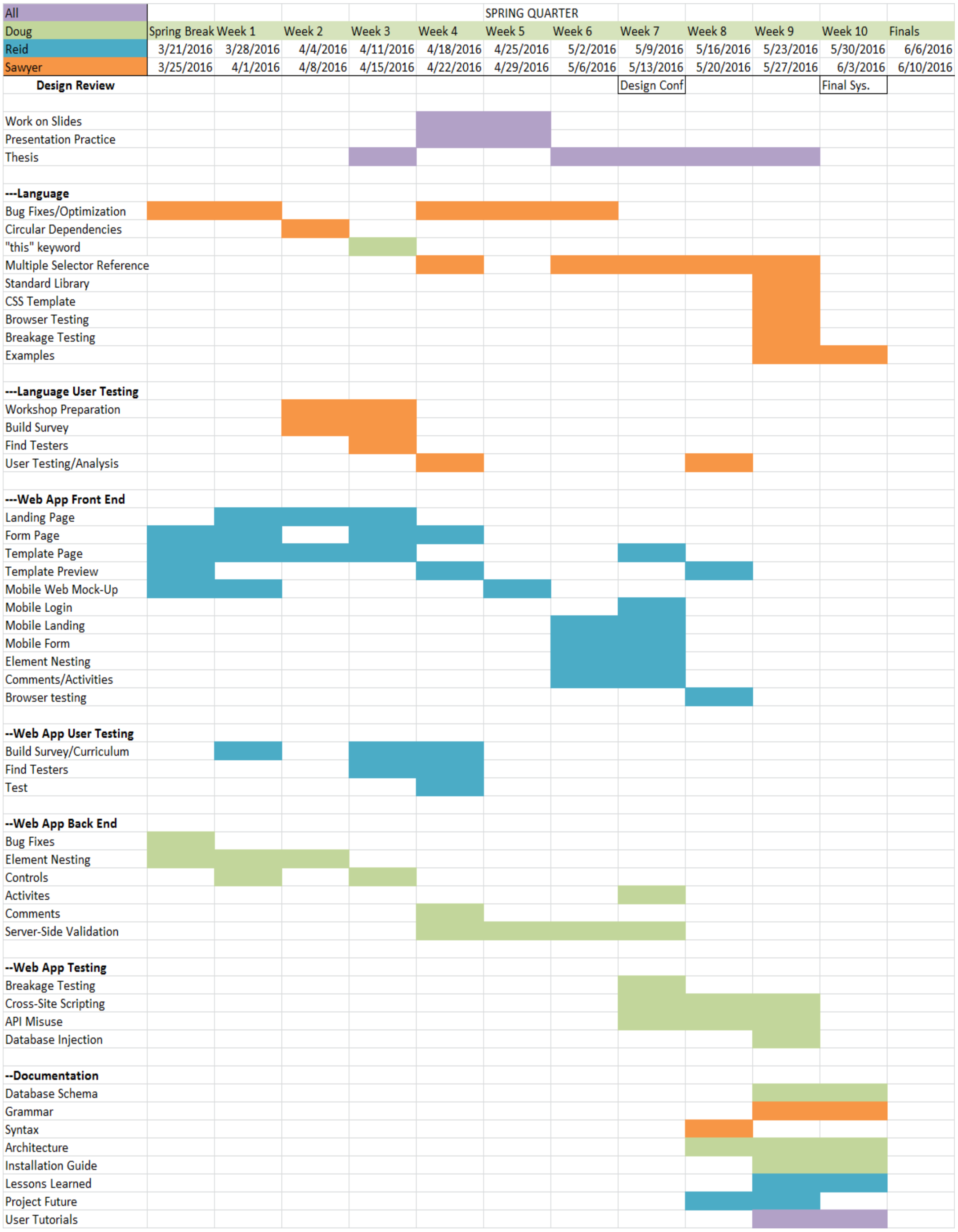


Figure 28: Development Timeline (Fall)

All 🟪						WINTER QUARTER					
Doug 🟩	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Finals
Reid 🟦	1/4/2016	1/11/2016	1/18/2016	1/25/2016	2/1/2016	2/8/2016	2/15/2016	2/22/2016	2/29/2016	3/7/2016	3/14/2016
Sawyer 🟧	1/8/2016	1/15/2016	1/22/2016	1/29/2016	2/5/2016	2/12/2016	2/19/2016	2/26/2016	3/4/2016	3/11/2016	3/18/2016
Design Review				Rev. D. Doc		Rev. Prac.				Op. Sys.	
Create Slides					🟪	🟪					
Presentation Practice					🟪	🟪					
Revised Design Doc				🟪							
---Language											
Syntax Tree											
Grammar			🟧	🟧							
Lexer		🟧	🟧	🟧							
Parser				🟧	🟧						
Semantic Analysis						🟧	🟧				
Code Generation (JS)							🟧	🟧	🟧		
Validation									🟧	🟧	
Examples							🟧			🟧	🟧
Server-side Validation							🟩	🟩			
---Web App Front End											
Detailed Web Mock-Up		🟦		🟦	🟦						
Landing Page			🟦	🟦							
View & Edit Form Page				🟦	🟦						
Issue Form Page						🟦					
View & Edit Template Page					🟦	🟦					
User Testing							🟦		🟦		
Revisions Based on Testing								🟦	🟦	🟦	
Uniform Integration									🟦	🟦	
--Web App Back End											
Database		🟩		🟩							
APIs Development		🟩	🟩	🟩	🟩	🟩					
Models			🟩	🟩	🟩	🟩	🟩				
Validation Integration									🟩	🟩	

Figure 29: Development Timeline (Winter)



🟪 All						SPRING QUARTER						
🟩 Doug	Spring Break	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Finals
🟦 Reid	3/21/2016	3/28/2016	4/4/2016	4/11/2016	4/18/2016	4/25/2016	5/2/2016	5/9/2016	5/16/2016	5/23/2016	5/30/2016	6/6/2016
🟧 Sawyer	3/25/2016	4/1/2016	4/8/2016	4/15/2016	4/22/2016	4/29/2016	5/6/2016	5/13/2016	5/20/2016	5/27/2016	6/3/2016	6/10/2016
Design Review								Design Conf			Final Sys.	
Work on Slides					🟪	🟪						
Presentation Practice					🟪	🟪						
Thesis				🟪			🟪	🟪	🟪	🟪		
---Language												
Bug Fixes/Optimization	🟧	🟧			🟧	🟧	🟧					
Circular Dependencies			🟧									
"this" keyword				🟩								
Multiple Selector Reference					🟧		🟧	🟧	🟧	🟧		
Standard Library										🟧		
CSS Template										🟧		
Browser Testing										🟧		
Breakage Testing										🟧		
Examples										🟧	🟧	
---Language User Testing												
Workshop Preparation			🟧	🟧								
Build Survey			🟧	🟧								
Find Testers				🟧								
User Testing/Analysis					🟧				🟧			
---Web App Front End												
Landing Page		🟦	🟦	🟦								
Form Page	🟦	🟦		🟦	🟦							
Template Page	🟦	🟦	🟦	🟦				🟦				
Template Preview	🟦				🟦				🟦			
Mobile Web Mock-Up	🟦	🟦				🟦						
Mobile Login								🟦				
Mobile Landing							🟦	🟦				
Mobile Form							🟦	🟦				
Element Nesting							🟦	🟦				
Comments/Activities							🟦	🟦				
Browser testing									🟦			
--Web App User Testing												
Build Survey/Curriculum		🟦		🟦	🟦							
Find Testers				🟦	🟦							
Test					🟦							
--Web App Back End												
Bug Fixes	🟩											
Element Nesting	🟩	🟩	🟩									
Controls		🟩		🟩								
Activites								🟩				
Comments					🟩							
Server-Side Validation					🟩	🟩	🟩	🟩				
--Web App Testing												
Breakage Testing								🟩				
Cross-Site Scripting								🟩	🟩	🟩		
API Misuse								🟩	🟩	🟩		
Database Injection										🟩		
--Documentation												
Database Schema										🟩	🟩	
Grammar										🟧	🟧	
Syntax									🟧			
Architecture									🟩	🟩	🟩	
Installation Guide										🟩	🟩	
Lessons Learned										🟦	🟦	
Project Future									🟦	🟦		
User Tutorials										🟪	🟪	

Figure 30: Development Timeline (Spring)

30 Ethics

With any project, no matter how good the intentions might be, ethics becomes a concern. This system is composed only of software components. As such, physical ethical concerns, such as personal safety, are out of the scope of the project. There are still however components that can compromise information security, and promised results.

30.1 Information Security

The Web Application includes a database which stores sensitive personal information. Privacy is always a major ethical concern, and several design decisions have been made to our system accordingly.

30.1.1 Design Decisions to Ensure Privacy

- Hierarchy of permissions: Admin > Associate > Client
  - Permissions can only be changed by an admin.
  - An associate for example, will only be able to view forms that involve his or herself.
- Data encryption
  - Although building mechanisms for encryption is out-of-scope for this project, the system will be supporting encrypted data. This ensures that data will not be open to the public.
- Developer privacy
  - The system might need maintenance by a developer or an admin. During maintenance, the parties will not be able to access personal information.

30.2 Team and Organizational Ethics

As in all team environments, it is important to treat team members with mutual respect and understanding. Fair treatment among all members of a team is always a concern. During development, work was divided as evenly as possible, and all feedback is heard and incorporated into the design. Extra consideration was given to each others' schedules. During code reviews, constructive criticism is the ideal, while differences of subjective opinion are avoided.

30.3 Social and Cultural Issues

30.3.1 Product expectations

There is always an ethical duty for any individual to do his/her best, be truthful, and guarantee results to a degree. The goal is to implement a form creation tool to build forms more easily, but more importantly, the goal is to make it easier on the user to create and fill out the forms required of them. Both of these conditions are required for a the project to be considered ethical at even the most basic level.

30.3.2 Sponsor expectations

This project was funded by Hewlett-Packard. They are funding it in exchange for special consideration of their input and concerns. HP will ultimately take ownership of the Web Application, and any future users of it will expect software on the level of quality of any other HP system.

30.3.3 Malicious use

By making a tool to create forms more easily, creating a phishing form is that much easier to build as well. This tool cannot prevent itself from being used for malicious purposes, such as tricking users and stealing information.

30.4 Documentation

30.4.1 Open source library

All code produced for the Uniform language will be public for the world to see, work on, and use. It is an ethical concern to make sure that the work done is well-implemented and well-documented to ensure that others picking up the project have the tools to properly understand and build off the current framework.

30.4.2 API documentation

The Web Application will be passed to Hewlett-Packard for further development and additional features. As such, it is extremely likely that other developers completely unfamiliar with the project will be required to write code building on top of the system defined in this document. In particular, the API exposed by the server is one of the most integral and complicated pieces of the project. Proper and in-depth documentation on the APIs is critical to bringing other developers up to speed and how the system works and how it can be extended.

31 Aesthetic Analysis

Aesthetics are a major part of any product, as it defines how users will interact with and perceive it. This is especially true for engineering, since anything developed ultimately must be used by people, and they are the ones who determine its value. The end user must always come first in any design, and the best way to modify how a user interacts with a particular product is to modify its aesthetics. The end goal of any engineering effort is to create the best user experience possible; this causes aesthetics to be among the most important aspects of the design. This project is no exception to this. The two major components each have their own aesthetic requirements and challenges to provide the user experience that will be expected of them.

31.1 Uniform Validation Language

The first component, the validation language, does not contain any visual component which can be manipulated aesthetically to influence our users. Rather, its aesthetic comes from the design of the language itself. Its syntax, grammar, diction, consistency, clarity, and readability are no different than traditional human languages. The main difference between this design and that of natural language is that Uniform was rigidly defined for a very clear and specific purpose, while human tongues have slowly evolved over time to encompass new thoughts and discuss new ideas. Even computer languages have evolved over time, from C to C++ or JavaScript to NodeJS, but their original creators exercised a level of control that no one person could claim over any human language. The development team is currently in this position for Uniform, and the effectiveness of the design will decide if it will ever be used by society.

The aesthetics of natural languages are often most clearly seen in poems, where words are aligned to create a specific meter and rhyme. This involves choosing the single perfect word from a chaotic dictionary of options accumulated over centuries of use. Uniform will find its aesthetic appeal in the expansive meaning that can be conveyed with a minimal set of terms. A simple and concise yet powerful statement can have the same impact as a perfectly timed rhyme in natural language. This will provide users with the most productive and enjoyable experience writing our language, making such a design the end goal of the language. Table 3 illustrates a few decisions made to achieve this purpose.

Design Decision	Aesthetic Impact
Minimal keywords	Reducing the keywords of the language minimizes the dictionary users must learn and understand. This makes the language simpler to learn and easier to understand.
Internal consistency	There are intended to be **`zero`** irregularities within the grammar and style of Uniform. The inconsistency of natural languages often makes them difficult to learn and adds unnecessary complexity to otherwise simple statements.
Clear meaning	Many computer languages use symbols as shortcuts for many concepts, such as `&&`, ||, and `!` for `and`, `or`, and `not`. These symbols obfuscate the meaning of very simple concepts, so Uniform avoids this and tries to be as verbal as possible, only using symbols where they would be clearer than their spoken counterparts (such as `x < y` instead of `x lessThan y`).
CSS selectors	Uniform uses CSS selectors to identify form elements due to its familiarity with users and strong documentation. It also gives a huge amount of power very simplistically by allowing a single “term" to refer to any number of arbitrary elements within a given form.
Recursive validation	Humans tend to think hierarchically, so it is very easy to conceptualize a form as being completed when a set of subconditions are met, which may depend on further subconditions and so on. Uniform directly supports this conceptual model to help users understand their particular problem and invent a clean and scalable solution.

Table 3: Design decisions and their aesthetic impact on the Uniform Validation Language

31.2 Web Application

The Web Application uses the Uniform Validation Language to improve the user experience of filling out forms. Today, completing a form is something everyone must do and no one enjoys. Forms are often complicated, confusing, and contradictory leading to errors, time and money lost, and dissatisfied users. This system improves this user experience by utilizing the three main methods listed below.

31.2.1 Mobile Accessibility

People are always on-the-go in today's society and cannot afford to sit down at a computer to fill out many forms of a simpler nature. The Web Applications places emphasis on being mobile-accessible to cater to this user persona and improve their experience.

31.2.2 Error Handling

Forms are not always straightforward and it is easy to enter invalid or contradictory data without realizing it. Many modern form system protect against this, but implementations are often sloppy, lazy, or simply incorrect. This system will utilize Uniform to enable form creators to easily define logic for their forms which is clean and user-friendly without requiring any extra effort on the part of the author.

31.2.3 Collaboration

Even the best laid forms encounter unique situations that were not planned for, and the best way for users to be satisfied that everything is correct is to speak with an expert on the form or with the others involved in its completion. Collaboration among users adds a human element to the cold complexity of unfriendly forms and helps streamline the process to reduce unnecessary, unpleasant, and untimely back and forth emails between users.

Part IV

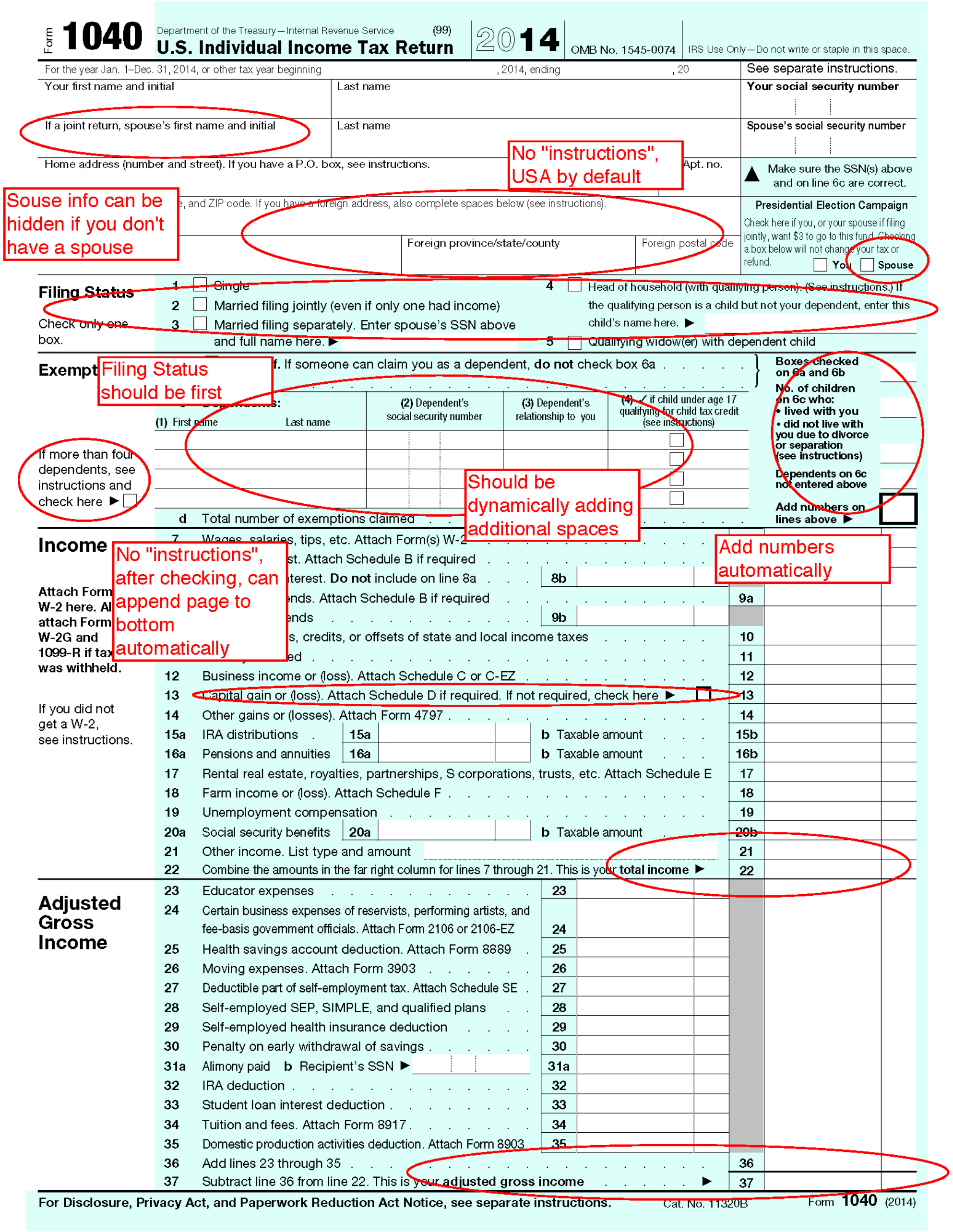
Appendix

A Design Philosophy of Paper Forms

A major problem with digital forms today is that they are designed as if they are actually on paper. This is no longer true, as most users will interact with digital forms instead of their paper equivalents. Entire software solutions have been built to expedite and simplify the completion of these forms, but it's important to recognize that none of these solutions address the underlying issue: forms bloated by edge cases, exceptions, and an archaic design tailored toward physical completion in a digital era. These problems exist in forms that everyone uses. For example, the U.S. Individual Income Tax Return Form, shown below with comments in fig. 31 and fig. 32.

The U.S. Government is notorious for having extremely complicated forms. They are often long, confusing, misleading, ambiguous, and otherwise difficult. People pay accountants to make this process as easy and painless as they can. Despite this reputation, the form shown in fig. 31 and fig. 32 is surprisingly well-made. Form 1040 was first published for tax year 1913, over 100 years ago, and has been slowly updated over time as tax codes have become more and more complicated. With this context, below are listed a few interesting design choices made on this form, and how they actually make a good amount of sense.

- Only uses one page front and back
    - Paper is expensive, this form is filled out by every household in America every year, which adds up drastically.
    - Big forms of many pages are intimidating to taxpayers. It is difficult to know where to start and where attention should be directed. A single page is much easier to manage and conceptualize.
    - Common cases are listed on this form, while edge cases defer to more paperwork. This keeps taxpayers focused on the small amount of forms that matter to their particular case while reducing the complexity of this form.



Form **1040** Department of the Treasury—Internal Revenue Service (99)

U.S. Individual Income Tax Return 2014 OMB No. 1545-0074 IRS Use Only—Do not write or staple in this space.

For the year Jan. 1–Dec. 31, 2014, or other tax year beginning , 2014, ending , 20 See separate instructions.

Your first name and initial | Last name | **Your social security number**

If a joint return, spouse's first name and initial | Last name | **Spouse's social security number**

Home address (number and street). If you have a P.O. box, see instructions. | Apt. no. | ▲ Make sure the SSN(s) above and on line 6c are correct.

e, and ZIP code. If you have a foreign address, also complete spaces below (see instructions).

Foreign province/state/county | Foreign postal code

Presidential Election Campaign
Check here if you, or your spouse if filing jointly, want $3 to go to this fund. Checking a box below will not change your tax or refund. □ You □ Spouse

Filing Status

Check only one box.

1 □ Single
2 □ Married filing jointly (even if only one had income)
3 □ Married filing separately. Enter spouse's SSN above and full name here. ▶
4 □ Head of household (with qualifying person). (See instructions.) If the qualifying person is a child but not your dependent, enter this child's name here. ▶
5 □ Qualifying widow(er) with dependent child

Exempt

f. If someone can claim you as a dependent, **do not** check box 6a

s:

(1) First name Last name	(2) Dependent's social security number	(3) Dependent's relationship to you	(4) ✓ if child under age 17 qualifying for child tax credit (see instructions)
			□
			□
			□
			□

If more than four dependents, see instructions and check here ▶ □

d Total number of exemptions claimed

Boxes checked on 6a and 6b
No. of children on 6c who:
• lived with you
• did not live with you due to divorce or separation (see instructions)
Dependents on 6c not entered above
Add numbers on lines above ▶

Income

Attach Form W-2 here. Al attach Form W-2G and 1099-R if tax was withheld.

If you did not get a W-2, see instructions.

7 Wages, salaries, tips, etc. Attach Form(s) W-2
st. Attach Schedule B if required
terest. **Do not** include on line 8a 8b
nds. Attach Schedule B if required 9a
ends 9b
s, credits, or offsets of state and local income taxes 10
ed 11
12 Business income or (loss). Attach Schedule C or C-EZ 12
13 Capital gain or (loss). Attach Schedule D if required. If not required, check here ▶ □ 13
14 Other gains or (losses). Attach Form 4797 14
15a IRA distributions 15a b Taxable amount 15b
16a Pensions and annuities 16a b Taxable amount 16b
17 Rental real estate, royalties, partnerships, S corporations, trusts, etc. Attach Schedule E 17
18 Farm income or (loss). Attach Schedule F 18
19 Unemployment compensation 19
20a Social security benefits 20a b Taxable amount 20b
21 Other income. List type and amount 21
22 Combine the amounts in the far right column for lines 7 through 21. This is your **total income** ▶ 22

Adjusted Gross Income

23 Educator expenses 23
24 Certain business expenses of reservists, performing artists, and fee-basis government officials. Attach Form 2106 or 2106-EZ 24
25 Health savings account deduction. Attach Form 8889 25
26 Moving expenses. Attach Form 3903 26
27 Deductible part of self-employment tax. Attach Schedule SE 27
28 Self-employed SEP, SIMPLE, and qualified plans 28
29 Self-employed health insurance deduction 29
30 Penalty on early withdrawal of savings 30
31a Alimony paid b Recipient's SSN ▶ 31a
32 IRA deduction 32
33 Student loan interest deduction 33
34 Tuition and fees. Attach Form 8917 34
35 Domestic production activities deduction. Attach Form 8903 35
36 Add lines 23 through 35 36
37 Subtract line 36 from line 22. This is your **adjusted gross income** ▶ 37

For Disclosure, Privacy Act, and Paperwork Reduction Act Notice, see separate instructions. Cat. No. 11320B Form **1040** (2014)

Souse info can be hidden if you don't have a spouse

No "instructions", USA by default

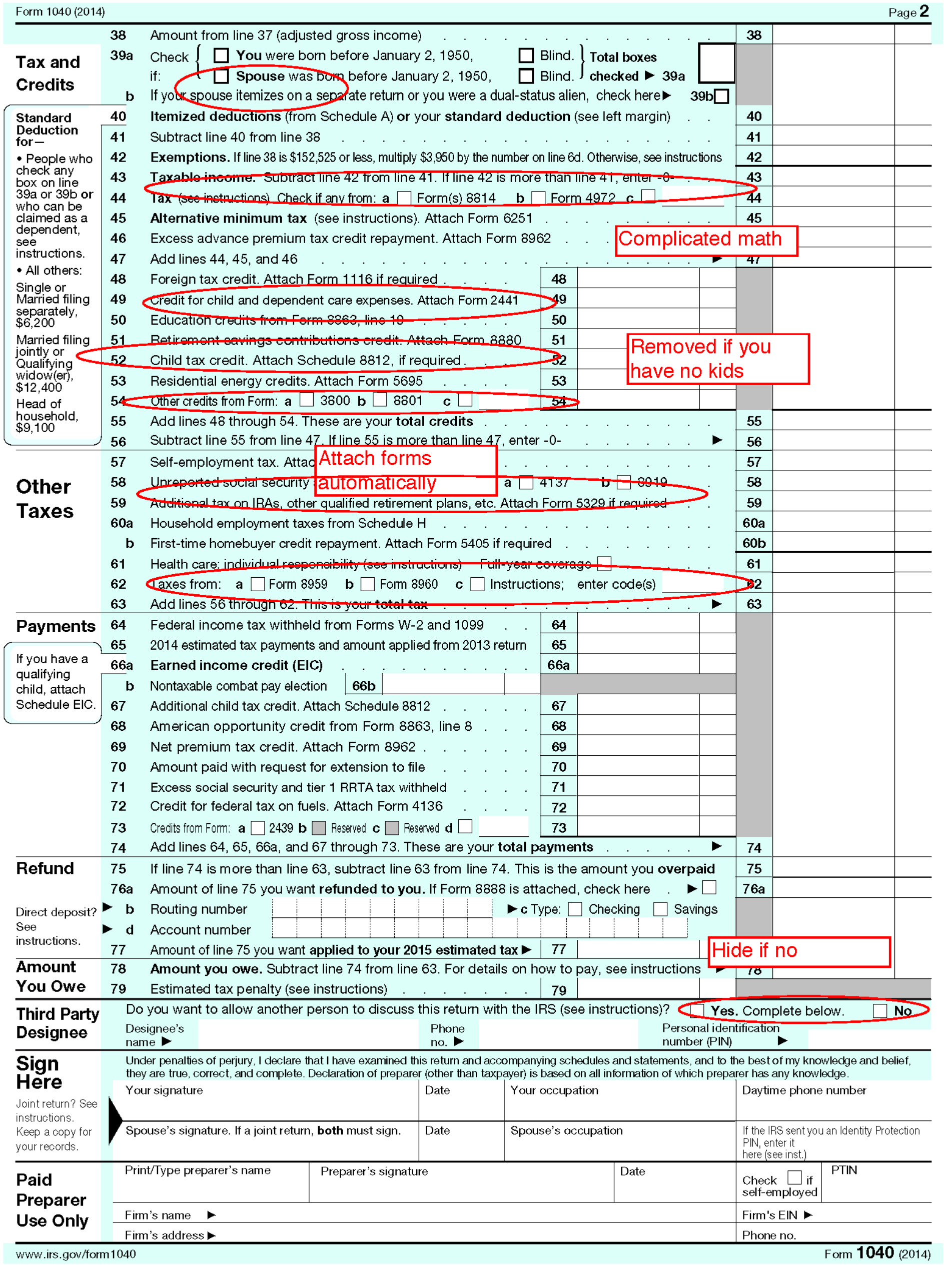
Filing Status should be first

Should be dynamically adding additional spaces

Add numbers automatically

No "instructions", after checking, can append page to bottom automatically

Figure 31: Form 1040 "U.S. Individual Income Tax Return"



Form 1040 (2014) Page **2**

Tax and Credits

38 Amount from line 37 (adjusted gross income) . . . 38

39a Check if: { □ **You** were born before January 2, 1950, □ Blind. } **Total boxes**
□ **Spouse** was born before January 2, 1950, □ Blind. } **checked** ▶ **39a** □

b If your spouse itemizes on a separate return or you were a dual-status alien, check here ▶ **39b** □

Standard Deduction for—
- People who check any box on line 39a or 39b **or** who can be claimed as a dependent, see instructions.
- All others:

Single or Married filing separately, $6,200

Married filing jointly or Qualifying widow(er), $12,400

Head of household, $9,100

40 **Itemized deductions** (from Schedule A) **or** your **standard deduction** (see left margin) . . 40

41 Subtract line 40 from line 38 . . . 41

42 **Exemptions.** If line 38 is $152,525 or less, multiply $3,950 by the number on line 6d. Otherwise, see instructions 42

43 **Taxable income.** Subtract line 42 from line 41. If line 42 is more than line 41, enter -0- . . 43

44 **Tax** (see instructions). Check if any from: **a** □ Form(s) 8814 **b** □ Form 4972 **c** □ 44

45 **Alternative minimum tax** (see instructions). Attach Form 6251 . . . 45

46 Excess advance premium tax credit repayment. Attach Form 8962 . . . 46

Complicated math

47 Add lines 44, 45, and 46 . . . ▶ 47

48 Foreign tax credit. Attach Form 1116 if required . . . 48

49 Credit for child and dependent care expenses. Attach Form 2441 49

50 Education credits from Form 8863, line 19 . . . 50

51 Retirement savings contributions credit. Attach Form 8880 51

52 Child tax credit. Attach Schedule 8812, if required . . . 52

53 Residential energy credits. Attach Form 5695 . . . 53

54 Other credits from Form: **a** □ 3800 **b** □ 8801 **c** □ 54

Removed if you have no kids

55 Add lines 48 through 54. These are your **total credits** . . . 55

56 Subtract line 55 from line 47. If line 55 is more than line 47, enter -0- . . . ▶ 56

Other Taxes

57 Self-employment tax. Attac . . . 57

Attach forms automatically

58 Unreported social security **a** □ 4137 **b** □ 8919 . 58

59 Additional tax on IRAs, other qualified retirement plans, etc. Attach Form 5329 if required . . 59

60a Household employment taxes from Schedule H . . . 60a

b First-time homebuyer credit repayment. Attach Form 5405 if required . . . 60b

61 Health care: individual responsibility (see instructions) Full-year coverage □ . . . 61

62 Taxes from: **a** □ Form 8959 **b** □ Form 8960 **c** □ Instructions; enter code(s) 62

63 Add lines 56 through 62. This is your **total tax** . . . ▶ 63

Payments

If you have a qualifying child, attach Schedule EIC.

64 Federal income tax withheld from Forms W-2 and 1099 . . 64

65 2014 estimated tax payments and amount applied from 2013 return 65

66a **Earned income credit (EIC)** . . . 66a

b Nontaxable combat pay election 66b

67 Additional child tax credit. Attach Schedule 8812 . . . 67

68 American opportunity credit from Form 8863, line 8 . . . 68

69 Net premium tax credit. Attach Form 8962 . . . 69

70 Amount paid with request for extension to file . . . 70

71 Excess social security and tier 1 RRTA tax withheld . . . 71

72 Credit for federal tax on fuels. Attach Form 4136 . . . 72

73 Credits from Form: **a** □ 2439 **b** ▨ Reserved **c** ▨ Reserved **d** □ 73

74 Add lines 64, 65, 66a, and 67 through 73. These are your **total payments** . . . ▶ 74

Refund

75 If line 74 is more than line 63, subtract line 63 from line 74. This is the amount you **overpaid** 75

76a Amount of line 75 you want **refunded to you.** If Form 8888 is attached, check here . ▶ □ 76a

Direct deposit? See instructions.

▶ b Routing number ▶ c Type: □ Checking □ Savings

▶ d Account number

77 Amount of line 75 you want **applied to your 2015 estimated tax** ▶ 77

Hide if no

Amount You Owe

78 **Amount you owe.** Subtract line 74 from line 63. For details on how to pay, see instructions ▶ 78

79 Estimated tax penalty (see instructions) . . . 79

Third Party Designee

Do you want to allow another person to discuss this return with the IRS (see instructions)? □ **Yes.** Complete below. □ **No**

Designee's name ▶ Phone no. ▶ Personal identification number (PIN) ▶

Sign Here

Joint return? See instructions. Keep a copy for your records.

Under penalties of perjury, I declare that I have examined this return and accompanying schedules and statements, and to the best of my knowledge and belief, they are true, correct, and complete. Declaration of preparer (other than taxpayer) is based on all information of which preparer has any knowledge.

Your signature	Date	Your occupation	Daytime phone number
Spouse's signature. If a joint return, **both** must sign.	Date	Spouse's occupation	If the IRS sent you an Identity Protection PIN, enter it here (see inst.)

Paid Preparer Use Only

Print/Type preparer's name	Preparer's signature	Date	Check □ if self-employed	PTIN
Firm's name ▶			Firm's EIN ▶	
Firm's address ▶			Phone no.	

www.irs.gov/form1040 Form **1040** (2014)

Figure 32: Form 1040 “U.S. Individual Income Tax Return" (Back)

  - Instructions are separate so they do not need to be copied for every form filed.

- Metadata is required in the margins
  - Taxpayers must write extra data in the margins listing the number and types of their dependents, whether or not additional forms should be expected.
  - Taxes are complicated, as such there many different departments covering many different classifications and cases that are possible. When processing a form, it is easier look at a single number or check box to determine where to send it to the group most applicable, rather than training the entire IRS staff to evaluate every possible case on the entire form for everyone all at once.

- Numbered lines
  - Each question and answer are appropriately numbered to clearly associate the two and avoid errors of writing in the wrong box.
  - Questions can reference each other, telling taxpayers to use values from other questions in a clear and concise manner.

This illustrates the design philosophy that went into the 1040, and it has stood the test of time. Looking at all the various iterations of this form, from 1913 to 2011, the form has only undergone minor changes and additions. By 1930, the overall structure of the form is nearly identical to its modern counterpart. While more questions have been added fairly regularly, the form itself hasn't had a major update in over 85 years.[1]

However the world of 2016 is not the same 1913, the digital age brings different requirements and constraints than previous, and the 1040 has not kept up. Below is listed the same design choices praised in 1930, and why they are no longer applicable today.

- Only uses one page front and back
  - Computers are not paper. A screen can scroll as far as it needs to and no trees are harmed.
  - Digital forms can separate their content more practically and provide clear progress indicators to their users. The physical size of a form is irrelevant today.

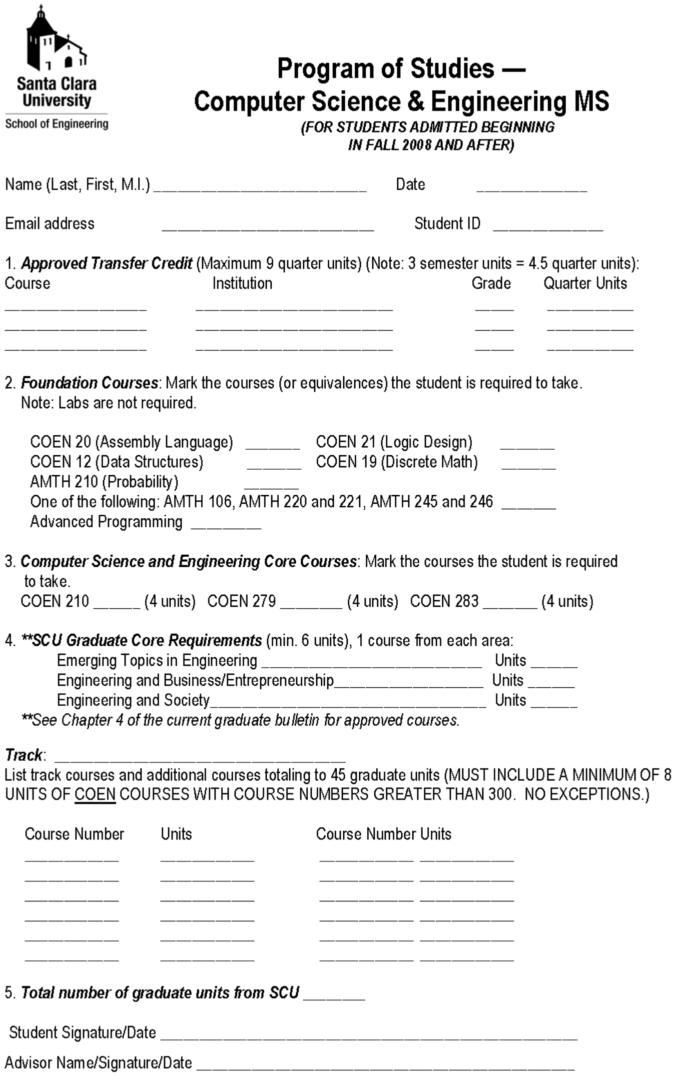
  - Users are forced to redirect to other forms and fill them out as necessary. In a digital environment, it is possible to customize the form for each particular user. A form of information for a spouse should be hidden until the user indicates that they married, at which time they can be presented with the extra fields for their spouse to be completed immediately without confusion.
  - Printed instructions are often expensive, easily lost, bulky, and out of date, making them unreliable and confusing for users. Digital instructions can be retrieved instantly on-demand with zero cost to the user.
- Metadata is required in the margins
  - Computerized environments should be smart enough to compute this kind of metadata for their users. Computers are far faster and more reliable than any human in this regard, reducing errors and frustration.
  - Computers automate the processing of all forms of information, and can compute whatever values they require from the raw data. No one will ever look at the form and manually direct it one way or another.
- Numbered lines
  - In a reasonably spaced form, there is no risk of entering data in the wrong field, making a numbering system obsolete.
  - Any question that is dependent on another should simply have its value computed using the raw data, avoiding inconsistencies.

Form 1040 was designed with a piece of paper in mind, and succeeded in that context quite well. Now the 1040 is filled out primarily online, in a completely different context. For this form to continue to be usable in a modern digital environment, it needs to adapt and redesign for the digital age. This example focused on a common “bad" form, but it is only indicative of a larger problem of an out-of-date methodology regarding forms, which is prevalent almost everywhere.

The Uniform language aims to address these problems by creating a simple tool which will allow forms to be built quickly and easily while taking advantage of digital features without falling back to old methods.

B Santa Clara Graduate Program of Studies Form

The SCU Graduate COEN Program of Studies form is a good example of a lackluster form that can be cleaned up and expedited through the use of Uniform. There are several sections in the program of studies that either expect a specifically formatted input or enforce constraints on the combinations of inputs which can be provided. On a physical form the individual filling it out is required to ensure that these criteria are met. On a hard coded form it would be fairly tedious to implement these criteria using JavaScript on the client-side and again in PHP or another language on the server-side to provide a back-end. With Uniform, implementing the Program of Studies is simple.



Santa Clara University
School of Engineering

**Program of Studies —
Computer Science & Engineering MS**

*(FOR STUDENTS ADMITTED BEGINNING
IN FALL 2008 AND AFTER)*

Name (Last, First, M.I.) _________________________ Date _____________

Email address _________________________ Student ID _____________

1. ***Approved Transfer Credit*** (Maximum 9 quarter units) (Note: 3 semester units = 4.5 quarter units):

Course Institution Grade Quarter Units
________________ ______________________ ____ _________
________________ ______________________ ____ _________
________________ ______________________ ____ _________

2. ***Foundation Courses***: Mark the courses (or equivalences) the student is required to take.
Note: Labs are not required.

COEN 20 (Assembly Language) ______ COEN 21 (Logic Design) ______
COEN 12 (Data Structures) ______ COEN 19 (Discrete Math) ______
AMTH 210 (Probability) ______
One of the following: AMTH 106, AMTH 220 and 221, AMTH 245 and 246 ______
Advanced Programming ________

3. ***Computer Science and Engineering Core Courses***: Mark the courses the student is required to take.
COEN 210 _____ (4 units) COEN 279 _______ (4 units) COEN 283 ______ (4 units)

4. *****SCU Graduate Core Requirements*** (min. 6 units), 1 course from each area:
Emerging Topics in Engineering __________________________ Units _____
Engineering and Business/Entrepreneurship________________ Units _____
Engineering and Society_________________________________ Units _____
***See Chapter 4 of the current graduate bulletin for approved courses.*

Track: ___________________________________
List track courses and additional courses totaling to 45 graduate units (MUST INCLUDE A MINIMUM OF 8 UNITS OF <u>COEN</u> COURSES WITH COURSE NUMBERS GREATER THAN 300. NO EXCEPTIONS.)

Course Number Units Course Number Units
__________ __________ __________ __________
__________ __________ __________ __________
__________ __________ __________ __________
__________ __________ __________ __________
__________ __________ __________ __________
__________ __________ __________ __________

5. ***Total number of graduate units from SCU*** _______

Student Signature/Date ____

Advisor Name/Signature/Date ____

Figure 33: SCU Graduate Program of Studies Form

```
<form id="programOfStudies">
    <div class="subForm" id="studentInformation">
        <input type="text" id="name"/>
        <input type="text" id="scuID"/>
        <input type="text" id="email"/>
        <input type="text" id="gradDate"/>
    </div>
    <div class="subForm list" id="transferCredits">
        <!--  Row div can be duplicated as many times as necessary -->
        <div class="row">
            <input placeholder="Course & Title" type="text" class="CaT"/>
            <input placeholder="Institution" type="text" class="inst"/>
            <input placeholder="Grade" type="text" class="grade"/>
            <input placeholder="Units" type="text" class="transferUnits"/>
            <input placeholder="Year Completed" type="text" class="yrComp"/>
        </div>
    </div>
    <div class="subForm" id="foundationCourses">
        <input type="checkbox">COEN 20</input>
        <input type="checkbox">COEN 21</input>
        <input type="checkbox">COEN 12</input>
        <input type="checkbox">COEN 19</input>
        <input type="checkbox">AMTH 210</input>
        <input type="checkbox">Advanced Programming</input>
    <div class="subForm" id="amthSplit">
        <input type="radio">AMTH 106</input>
        <input type="radio">AMTH 220</input>
        <input type="radio">AMTH 245</input>
    </div>
    <div class="subForm" id="coenCore">
        <input type="checkbox">COEN 210</input>
        <input type="checkbox">COEN 279</input>
        <input type="checkbox">COEN 283</input>
    </div>
    <div class="subForm" id="gradCore">
        <input type="text" id="ete"/>
        <input type="text" id="ebe"/>
        <input type="text" id="eas"/>
    </div>
    <div class="subForm list" id="track">
        <!-- Row div can be duplicated as many times as necessary -->
        <div class="row">
```

```
            <input placeholder="Title" type="text" class="trackTitle"/>
            <input placeholder="Course Number" type="text" class="trackCN"/>
            <input placeholder="Coen Units" type="text" class="coenUnits"/>
            <input placeholder="Non-Coen Units" type="text" class="nonCoenUnits"/>
        </div>
    </div>
    <div class="subForm" id="totalUnits">
        <input type="text" id="tuInput"/>
    </div>
    <div class="subForm" id="signatures">
        <input type="radio" id="newPS">New PS</input>
        <input type="radio" id="oldPS">Updated PS</input>
        <input type="signature" id="studentSignature"/>
        <input type="signature" id="advisorSignature"/>
    </div>
</form>
```

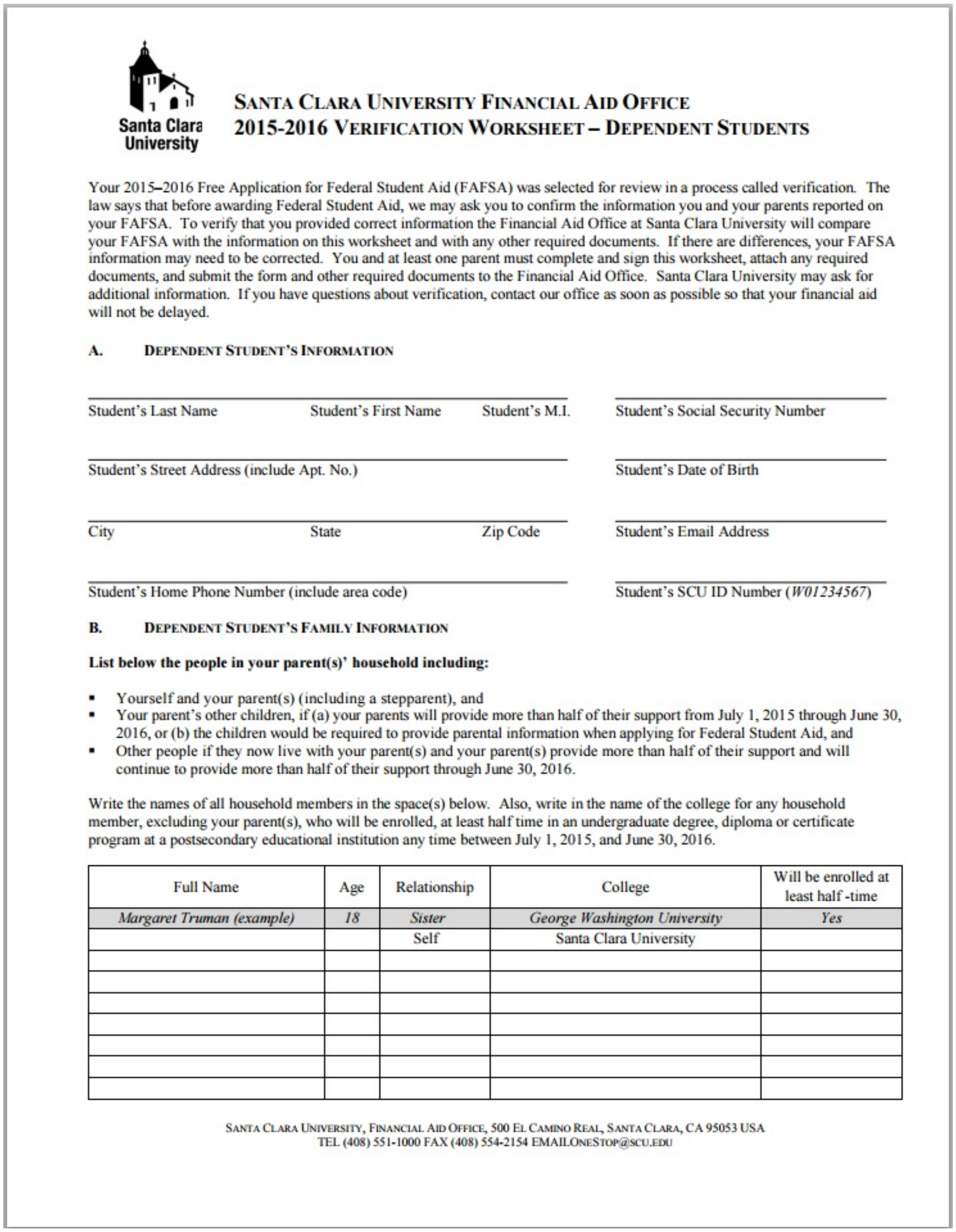
As stated previously, the Uniform language offers RegEx support, both pre-defined and user-defined, which can be used to enforce basic input formatting.

```
$("#programOfStudies") {
    //Form valid if all sub-forms are valid
    valid: all $(".subForm") is valid;
}

// Id is a ’W’ followed by 7 digits
@idRegex: /"W[0-9]{7}"/;
@emailRegex: /".*@.*"/;
@yearRegex: /"[0-9]{4}"/;
@filled: /".*"/;

$("#studentInformation") {
    valid:
        $("#name") matches @filled and
        $("#scuID") matches @idRegex and
        $("#email") matches @emailRegex and
        $("#gradDate") matches @yearRegex;
}

// Valid if less than 9 transfer units
```

```
$("#transferCredits") {
    valid: all this.find("input") is valid and
        sum this.find(".transferUnits") < 9;
}

$(".grade") {
    valid: this matches @integer;
}

$(".transferUnits") {
    valid: this matches @integer;
}

// sum not implemented yet, but is an intended feature
// Valid if more than 6 credits of grad core courses listed
@gradCoreSum: sum $("#gradCore input");
$("#gradCore") {
    valid:
        // Short circuit evaluation
        all this.find("input") is valid and
        @gradCoreSum >= 6;
}

$("#gradCore input") {
    valid:
        this matches @integer and
        this > 0;
}

// Sum values for each section of units
@coenUnitsSum = sum $("#track .coenUnits");
@nonCoenSum = sum $("#track .nonCoenUnits");

// Set requirements on totals
// Must have more than 8 COEN units
// Must use less than 10 non-COEN units
$("#track") {
    valid:
        all this.find("input") is valid and
        @coenUnitsSum > 8 and
        @nonCoenSum < 10;
}

$(".trackCN") {
    valid:
        this matches @integer and
        this > 300;
}

$(".coenUnits") {
    valid:
        this matches @integer and
        this >= 1;
}

$(".nonCoenUnits") {
    valid: this matches @integer;
}

$("#totalUnits") {
    valid: $("#tuInput") is valid;
}

$("#tuInput") {
    valid: this matches @integer;
}

$("#signatures") {
    valid: all this.find("input") is valid;
}
```

C Santa Clara Financial Aid Office Verification Worksheet

The Santa Clara Financial Aid Verification Worksheet is clearly designed to be printed out and completed physically before being scanned and returned electronically. Below is the HTML markup for a version of this form implemented using the Uniform language which would drastically improve the turnaround time on such a document.



Santa Clara University

SANTA CLARA UNIVERSITY FINANCIAL AID OFFICE
2015-2016 VERIFICATION WORKSHEET – DEPENDENT STUDENTS

Your 2015–2016 Free Application for Federal Student Aid (FAFSA) was selected for review in a process called verification. The law says that before awarding Federal Student Aid, we may ask you to confirm the information you and your parents reported on your FAFSA. To verify that you provided correct information the Financial Aid Office at Santa Clara University will compare your FAFSA with the information on this worksheet and with any other required documents. If there are differences, your FAFSA information may need to be corrected. You and at least one parent must complete and sign this worksheet, attach any required documents, and submit the form and other required documents to the Financial Aid Office. Santa Clara University may ask for additional information. If you have questions about verification, contact our office as soon as possible so that your financial aid will not be delayed.

A. DEPENDENT STUDENT'S INFORMATION

Student's Last Name Student's First Name Student's M.I. Student's Social Security Number

Student's Street Address (include Apt. No.) Student's Date of Birth

City State Zip Code Student's Email Address

Student's Home Phone Number (include area code) Student's SCU ID Number (*W01234567*)

B. DEPENDENT STUDENT'S FAMILY INFORMATION

List below the people in your parent(s)' household including:

- Yourself and your parent(s) (including a stepparent), and
- Your parent's other children, if (a) your parents will provide more than half of their support from July 1, 2015 through June 30, 2016, or (b) the children would be required to provide parental information when applying for Federal Student Aid, and
- Other people if they now live with your parent(s) and your parent(s) provide more than half of their support and will continue to provide more than half of their support through June 30, 2016.

Write the names of all household members in the space(s) below. Also, write in the name of the college for any household member, excluding your parent(s), who will be enrolled, at least half time in an undergraduate degree, diploma or certificate program at a postsecondary educational institution any time between July 1, 2015, and June 30, 2016.

Full Name	Age	Relationship	College	Will be enrolled at least half -time
Margaret Truman (example)	*18*	*Sister*	*George Washington University*	*Yes*
		Self	Santa Clara University	

SANTA CLARA UNIVERSITY, FINANCIAL AID OFFICE, 500 EL CAMINO REAL, SANTA CLARA, CA 95053 USA
TEL (408) 551-1000 FAX (408) 554-2154 EMAILONESTOP@SCU.EDU

Figure 34: SCU Financial Aid Verification Form

```html
<form id="financialAidVerification">
    <p>
        Your 2015, 2016 Free Application. . . {remaining text omitted for brevity}
    </p>
    <h1>Dependent Student Information</h1>
    <div class="subForm" id="studentInformation">
        <input type="text" id="lastName"/>
        <input type="text" id="firstName"/>
        <input type="text" id="middleInitial"/>
        <input type="text" id="socialSecurity"/>
        <input type="text" id="address"/>
        <input type="text" id="dateOfBirth"/>
        <input type="text" id="city"/>
        <input type="text" id="state"/>
        <input type="text" id="zipCode"/>
        <input type="text" id="email"/>
        <input type="text" id="phone"/>
        <input type="text" id="studentID"/>
    </div>
    <h1>Dependent Student Family Information</h1>
    <ul>
      <h2>List below the people in your parent household including:</h2>
      <li>Yourself and your parent(s) (including a step-parent), and</li>
        . . .
        {Remaining list omitted for brevity.}
    </ul>
    <p>
      Write the names of all household members. . . {text omitted for brevity.}
    </p>
    <div class="subForm list" id="familyInformation">
        <!-- Row div can be duplicated as many times as necessary -->
        <div class="row">
            <input placeholder="Full Name" type="text" class="memberName"/>
            <input placeholder="Age" type="text" class="memberAge"/>
            <input placeholder="Relationship" type="text" class="memberRelationship"/>
            <input placeholder="College" type="text" class="memberCollege"/>
            <input placeholder="Will be enrolled at least half-time" type="text" class="memberEnrollment"/>
        </div>
    </div>
</form>
```

The application of the Uniform language follows.

```
$("#financialAidVerification") {
    // Root form valid if all sub-forms are valid
    valid:
        $("#studentInformation") is valid and
        $("#familyInformation") is valid;
}

$("#studentInformation") {
    // Valid if all inputs falling under this id are valid
    // By default, text inputs are valid if they contain data
    valid: all this.children("input") is valid;
}

$("#socialSecurity") {
    valid: this matches /^\d{3}-\d{2}-\d{4}$/;
}

$("#zipCode") {
    valid: this matches /^\d{5}$/;
}

$("#studentID") {
    valid: this matches /W[0-9]{8}/;
}

$("#familyInformation") {
    // Valid if all items falling under this id are valid
    valid: all this.find("input") is valid;
}

$("#age") {
    // @number is a pre-set variable defined by the language
    valid: this matches @number;
}
```

D Uniform Server Validator Example

The following code comprises a NodeJS server which asks the user to check a box for the request to be considered valid. If the user does *not* check the box, then the request is invalid. If the user *does* check the box, then the request is valid.

Note that normally invalid requests will be caught on the client and not be sent to the server at all. This is how most users would want to interact with the application. This example is meant to show how the server can react to bad data, so client validation has been disabled by running the line

```
uniform.options.disableClientValidation();
```

on the client. This is useful for debugging a server when it is given bad requests, but is not intended to be used on a production system.

server.js

```
var express = require("express");

// Require Uniform server-side validator
var validate = require("uniform-validation");

// Create express application
var app = express();

// Set directory to serve static files
app.use(express.static("www"));

// On POST request to /submit
app.post("/submit", validate({
    path: "www/script.ufm", // Uniform script to validate
    main: "#rootForm" // Check if $("#rootForm") is valid
}), function (req, res) { // Callback function
    ...
});

// Start server and listen for requests on port 8000
app.listen(8000);
```

www/index.html

```html
<!DOCTYPE html>
<html>
    <head>
        <script src="/jquery.js"></script>
        <script src="/uniform.js"></script>
        <script>uniform.options.href("/script.ufm");</script>
        <script>uniform.options.disableClientValidation();</script>
    </head>
    <body>
        <form id="rootForm" method="POST" action="/submit">
            <input id="chkBox" name="chk" type="checkbox"/>
            Check me to validate!

            <button type="submit">Submit</button>
        </form>
    </body>
</html>
```

www/script.ufm

```
$("#rootForm") {
    // Form is valid if the check box is checked
    valid: $("#chkBox");
}
```

E API Documentation

The API for the Web Application was developed using Loopback.io, which supports the Swagger API standard. It is able to export all the APIs it exposes as a JSON file that can be interpreted by any Swagger renderer and displayed in any format. The framework even contains a built in API explorer to help developers learn, build, and debug the API. It can be accessed by running the server in a development environment and visiting “/explorer”. This shows all the APIs the server can accept and how to use them. It also allows users to directly send API requests to easily examine their effects.

Not all API documentation is accurate, this is because many APIs were custom-built for this application and were not auto-generated by Loopback, while other APIs were modified from their original intended usage. As such, the APIs may not work *precisely* as indicated in the documentation.

References

[1] Yanofsky, David. "Line for line, US income taxes are more complex than ever." *Quartz.* 13 Dec. 2012. Web. 11 Apr. 2015.

[2] Synodinos, Dio. "HTML 5 Web Sockets vs. Comet and Ajax." *InfoQ.* N.p., 11 Dec. 2008. Web. 18 Nov. 2015.

[3] Lubbers, Peter. "HTML5 WebSocket - A Quantum Leap in Scalability for the Web." *WebSocket.org.* Kaazing Corporation, n.d. Web. 10 Oct. 2015.

[4] *Can I use....* N.p. N.d. Web. 24 May 2016.

[5] Sotelo, Caleb. "Evolution of the Single Page Application." *Paislee.io.* N.p., 08 July 2014. Web. 10 Oct. 2015.

[6] Wasson, Mike. *The Traditional Page Lifecycle vs. the SPA Lifecycle.* Digital image. *MSDN Magazine.* Microsoft, Nov. 2013. Web. 11 Nov. 2015.

[7] Salihefendic, Amir. "Is Node.js Best for Comet?" *Hacking and Gonzo.* N.p., 22 Oct. 2010. Web. 10 Oct. 2015.

[8] McNulty, Eileen. "SQL vs. NoSQL- What You Need to Know." *Dataconomy.* N.p., 1 July 2014. Web. 15 Oct. 2015.

[9] Tezer, O.S. "A Comparison Of NoSQL Database Management Systems And Models." *DigitalOcean.* N.p., 21 Feb. 2014. Web. 15 Oct. 2015.

[10] Strongloop. "datasources.json." *Loopback - Documentation.* 15 Apr. 2016. Web. 25 May 2016.