uVis.web

Egil Kristoffer Gorm Hansen
IT University of Copenhagen
August, 2013

Supervisor: Søren Lauesen

Table of Contents

1 Introduction

uVis is a Windows-based tool for creating custom interactive data visualizations following the drag-drop-set-property principle (Lauesen, et al., u.d.). Think of a drawing program where you drag and drop visual components onto a canvas, and then configure their appearance through properties. What sets uVis apart from tools in its category is that users can express the value of a property using a formula. uVis is targeted at *local developers*, a term use to describe a domain expert within an organization, who is also an advanced user of tools such as Microsoft Excel or Microsoft Access. Its mission is make it possible for local developers to create *interactive* visualizations of data, visualizations it is not possible to create in with e.g. Microsoft Excel, and would normally require a trained programmer to create. uVis is still in active development by Søren Lauesen and his research team at ITU.

The motivation for uVis came from the hospital sector in Denmark, where patient data is stored in many different database-driven systems, and there is a constant need to evolve the way the doctors and nurses view and interact with the patient data. However, employing one or more fulltime developer to advance the internal systems is too expensive, so uVis was created to enable internal hospital workers with the appropriate domain knowledge to develop advanced visualizations of patient data themselves. Figure 1 below has two examples of visualizations of patient data.

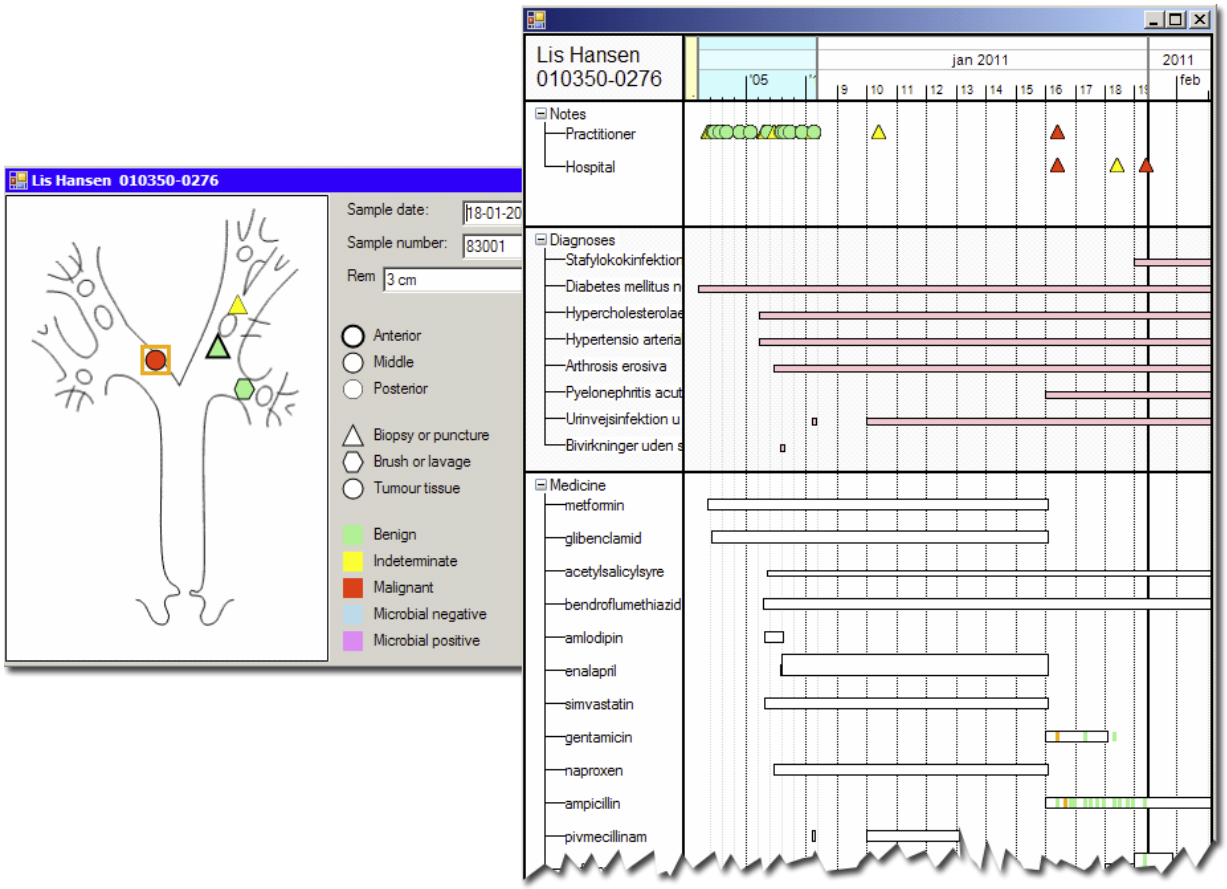


Figure 1: Two forms with custom visualization of medical data. On the left, we see a bronchoscopial biopsies and the later lab results. On the right, we see an overview of a patient's medical history, including notes, diagnoses, and medication. Source: (Lauesen, et al., n.d.).

However, uVis is tied to the Microsoft Windows platform since it is built on Windows Forms and the .NET Framework. That is a limiting the adaption of the technology, specifically for the hospital sector in Denmark that only wants web-based solutions.

Therefore, my first task was to explore the feasibility of bringing the uVis concept to the web, and if possible, my second task was to design and develop a prototype of the central parts of uVis, the uVis kernel, for the web.

This report contains the result of the first task. In it, I will show how to bring the concept of uVis to the web, including highlighting changes required to the uVis concept, and new opportunities that the web platform presents. This report will show how I designed and implemented a working prototype of the uVis kernel as well.

The prototype stays true to the concept of uVis. However, the implemented does diverge from the way uVis is implemented by using the *reactive programming* programming-paradigm. With reactive programming, it is possible to specify dependencies between variables that will automatically propagate changes in one variable to another variable that depend on it. Spreadsheet applications are an excellent example of reactive programming. In a spreadsheet, if a cell `A1 = 41`, and cell `B1 = A1 + 1`, the value produced by `B1`, e.g. 42, is automatically updated, if `A1` is changed. That is how variable assignment works in reactive programming, and it fits very well with how uVis formula properties work conceptually.

Reactive programming gives us some interesting possibilities for modelling how data should flow in an application. It specifically makes it easier to work asynchronous data from events and web requests and it could potentially result in a more responsive user interface. However, as this experiment shows, there are also downsides. Initial performance tests made with the prototype shows that memory usage could become a problem with visualizations containing more than 10,000 visual components. That said, with the core parts of the uVis kernel implemented and working, I do think using reactive programming has been a success, and a great learning experience as well.

The next step for uVis.web is to add the secondary functionality to the prototype that is currently missing, in particular a compiler that translates the uVis formula language to JavaScript. That would make it easier to start creating larger uVis applications for uVis.web, thus providing a more realistic way to measure performance. After that would come a web version of uVis Studio, the WYSIWYG editor use to create uVis applications.

This report does not expect the reader to have more than basic knowledge of web technologies, but experience with user interface programming, e.g. Windows Forms, and technologies such as Language-Integrated Query (LINQ) will make it easier to digest.

The source code for the prototype and the latest version of this report is available from my website at http://egilhansen.com/projects/uvis.web.

2 Background

The uVis kernel is a data-driven templating system, i.e. a system that takes template definitions and produces zero or more components for each template, depending on data passed to it. This is by itself not a novel idea, in fact, most templating systems works exactly this way, however, uVis has some interesting concepts that sets it apart from other templating systems. In particular, the ability to define each component's properties as formulas that can reference other component's properties sets uVis apart. However, before I go into details on how the property formulas work, I will first describe how the uVis kernel takes template definitions and transforms them into a visualization on the screen.

2.1 The uVis template algorithm

The uVis template algorithm is as follows:

1. uVis takes a ***.VIS file*** as input and creates a ***template tree*** from it.
2. Then uVis creates a ***component tree*** based on the template tree and on ***data*** from one or more databases.
3. Finally, each component renders itself to a canvas.

Let us try to create a simplified version of the medical history form shown to the right in Figure 1, focusing only on the medicine history part of the form. Using this as an example, we will go through each step in the algorithm and discuss the highlighted concepts (this example is inspired by a similar example in (Lauesen, 2011)).

Consider Figure 2 below. The **data map** to the right shows the relationship between a patient, his medicine orders, and the intakes for each order, i.e. a patient can have medicine orders, and each medicine order can have intakes. The data map represents the data source in the example.

The output form, shown to the left in Figure 2, is the final output from step 3, i.e. what uVis renders on the screen. The output is a simple medicine history chart for the currently selected patient[1]. Each of the patient's medicine orders is on its own row, starting with a label showing the text "Medicine:", followed by a box with the medicine name inside it. The horizontal position and the width of the box indicate the period in which the medicine was prescribed. The vertical bars in green and yellow indicates each intake of the medicine; green means full dosage and yellow means some of the dosages was skipped. As we see, there have been some intakes of medicine after the prescription period for this selected patient, which is not unusual.

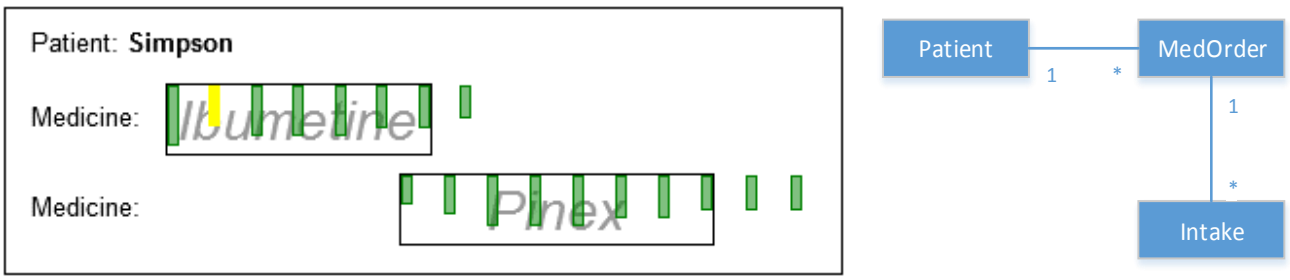


Figure 2: The rendering of the patient medicine chart (left), and the data map (right) for the patient medicine chart example.

[1] The output example is not optimized in terms of usability nor is it an efficient way to visualize medical data. However, it is a good example of the complexities the uVis kernel has to deal with.

Now that we have an idea of what we want to render on the screen, and how our database tables are structured, let us go through each step in the template algorithm.

2.1.1 Step 1: from .VIS file to template tree

A **.VIS file** is a textual representation of a single uVis form, i.e. a *Form* in Windows Forms terms, e.g. a single application window (in our example, the form is the output in Figure 2). The .VIS file contains descriptions of the form and all its templates, including their properties. The .VIS file is in plain text, and its notation follows the classic ini-file format, which makes it easy to edit and read directly in a text editor. However, most of the time, a local developer will use uVis Studio to create a uVis form, and save that form as a .VIS file.

A **template tree** is an in-memory representation of the content of the .VIS file, i.e. a tree of the templates that is part of a uVis form. The parent-child relationship between templates in the template tree, and thus the structure of the tree, is determined by the data source associated with each template via the special `Rows` property. If a template does not have a data source defined, it is automatically added as a child to the form template at the top of tree, and inherits its data source. This is a source of confusion for most people coming from other templating languages such as XAML or HTML, where templates are explicitly nested in each other, and their parent-child relationship is based on visual nesting and has nothing to do with data assigned to them, as is the case in uVis.

The template tree for our patient medicine chart example is visualized in Figure 3, with some properties included for each template. The properties listed inside each template are written as they would appear in the corresponding .VIS file (bold/italic added for legibility). Here we can see that the `Rows` property on the lower level templates reference the ID of their parents. We also see that the `lblPatient` template, that does not have an explicit data source defined, is a direct child of the `frmPatient` form.

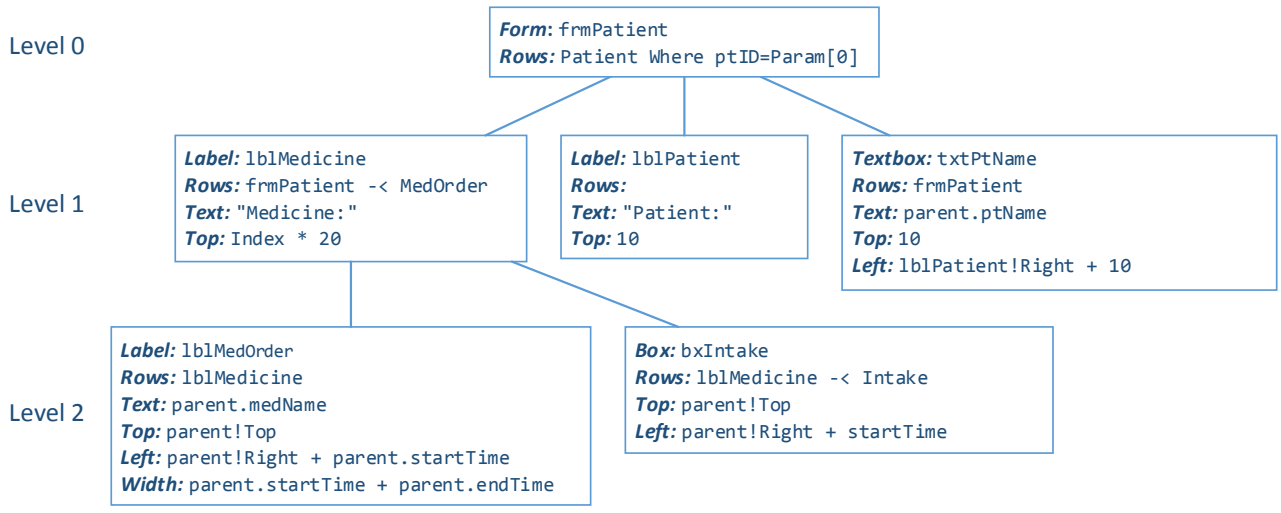


Figure 3: Template tree for the patient medicine chart example.

Creating a template tree from a .VIS file is just matter of parsing the .VIS file and creating the right object structure based on the content of each templates `Rows` property.

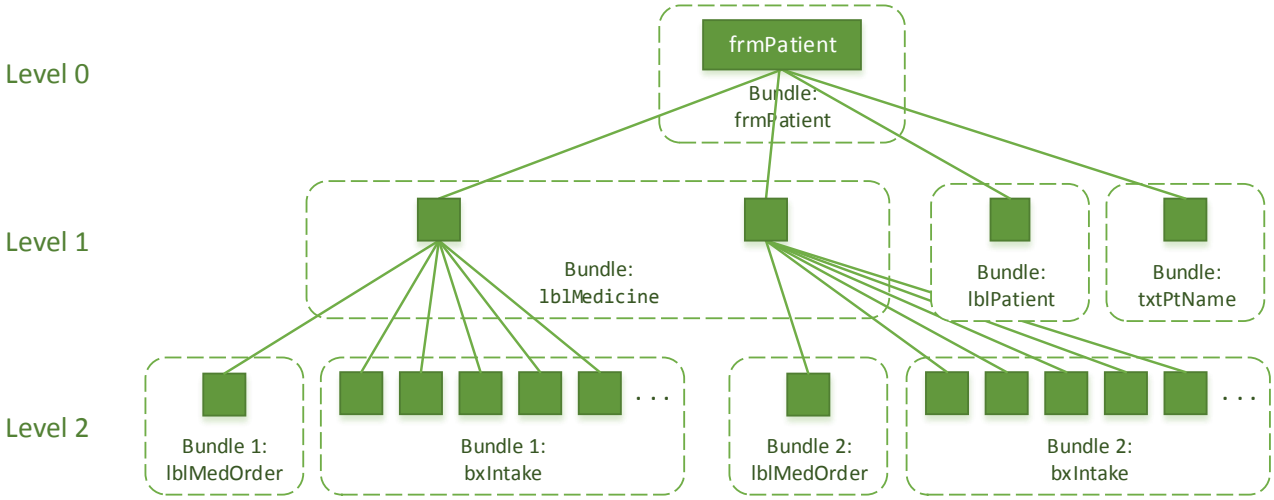


Figure 4: Component tree for the patient medicine chart example. Some intakes are not included explicitly.

2.1.2 Step 2: From template tree to component tree

Since the uVis kernel is a *data*-driven templating system, each template in the template tree can create zero or more components (see Figure 4), depending the data source attached to it. The exact rules are listed below in Table 1:

Rule #	Data source result	Instances created
1	No data source assigned	Create a single component
2	`Rows` property (data source) returns single record	Create a single component
3	`Rows` property (data source) returns N record	Create N components
4	`Rows` property (data source) returns zero records	Create zero components
5	`Rows` property (data source) returns an number X	Create X components

Table 1: Template tree to component tree mapping, based on data source for the template tree components.

With the rules in place, we can look at how each level in the template tree (Figure 3) creates components in the component tree (Figure 4).

Level 0 in the template tree: Here we see the form (`frmPatient`) that contains the patient medicine chart, i.e. the program window. It is bound to the data source `Patient Where ptID=Param[0]` as indicated by the `Rows` property. This is a database query written in uVis formula language that will select the patient whose ID equals `Param[0]`. The program that opened the form provided `Param[0]`.

Since the query returns one record, a single component is created.

Level 1 in the template tree: Here we see three components, `lblMedicine`, `lblPatient` and `txtPtName`. Let us start with the two latter.

The `lblPatient` label, which prints the “Patient:” at the top left corner of the output, does not have any data source defined via its `Rows` property, so it creates a single component at level 1 in the component tree.

After that comes a textbox, `txtPtName`, that prints the name of the current patient by looking up the value in its parent component’s data source, the selected patient record using the “dot” notation (`parent.ptName`). Since it shares its parent’s data source, which is a single record, it creates a single component at level 1 in the component tree.

The `lblMedicine` label prints the value “Medicine:” to the output. Its `Rows` property references its parent’s `Rows` property, the selected patient, to get to the patient’s medicine orders, using the join operator (`-<`). In this example, the patient *Simpson* has two medicine orders, *Ibumetine* and *Pinex*, so `lblMedicine` creates two components at level 1 in the component tree.

Level 2 in the template tree: Here we see the `lblMedOrder` textbox and the `bxIntake` box.

The `lblMedOrder` shares its parent’s data source. That means it creates a single component at level 2 in the component tree. However, since its parent, `lblMedicine`, created two components, `lblMedOrder` must create one component for *each* of the components `lblMedicine` created. The relationship is illustrated in the component tree, where the two `lblMedOrder` components each has one `lblMedOrder` attached to it.

Each `lblMedOrder` components shows the name of the medicine of their parent, i.e. *Ibumetine* and *Pinex*.

The `bxIntake` box’s data source references its parent’s `MedOrder` record (medicine order) and selects the intake records for that medicine order, and creates a vertical bar for each intake. Each intake is positioned based on the time the intake was taken. As we can see in the component tree, `bxIntake` creates components for both `lblMedicine` components. Looking at the expected output, we see there are 8 intakes for *Ibumetine*, and 10 for *Pinex*. However, some of the components have been omitted in the component tree drawing due to space restrictions.

Bundles

You might have noticed the word *Bundle* in Figure 4 and wondered what it means. Each template keeps track of the components it creates. However, if the parent of a template created more than one components, as in the case of `lblMedicine` and `biIntake`, the template must be able to identify which components belongs to each of the parent components. This is done by putting each set of components in a **bundle**, as we see on level 2 in the component tree.

2.1.3 Step 3: Rendering

uVis starts by creating a canvas component that fills the entire Windows Forms window. It is on this canvas that all components renders their visual component by default (each component has a visual component). It is possible for a developer to define additional canvas components that can be placed onto other canvases, dividing them up. The visual components can then explicitly be added to a canvas, making it easier to achieve certain layouts, via the `Canvas` property (e.g. `Canvas: c2`).

Visual components are placed on a canvas using absolute positioning, i.e. a number of pixels from the top, left, right, or bottom of the canvas they are rendered to. A canvas can scroll the components inside it and it will clip components that are outside of its viewport.

In the patient medicine chart example, there are no explicit canvases defined, so visual components are added to the default canvas. If two visual components overlap each other, as the `biIntake` boxes and the `lblMedOrder` boxes do, the visual components added last will be positioned on top (can be overridden by the `ZOrder` property).

Now that we know how the uVis template algorithm works, it is time to look at how properties work.

2.2 Properties expressed as formulas

As we saw in the template tree in Figure 3, the properties of a template can be formulas. The property formulas are evaluated once for *each* component. The result of each evaluation is a **component property**

that is assigned to the created component. For example, in the patient chart example, a component property named `Text` with the value "*Ibumetine*" is assigned to the `lblMedOrder` component in bundle 1.

The property formulas can combine data from several database tables with data from components, e.g. their component properties, and from data provided by the end-user, e.g. from keyboard or mouse. For example, in the patient chart example, we see that the `Top` property on the `bxIntake` template has a formula that is referencing the value of its parent's `Top` *component propert*y using the "bang" notation, i.e. `parent!Top`. This means that for each component `bxIntake` creates, it must reference the parent component in the component tree, and use its `Top` component property when evaluating the formula `parent!Top`.

This type of dependency between properties was inspired by how spreadsheets work. In a spreadsheet, you can set the value of a cell to be the sum of two other cells, e.g. `A3 = A1 + A2`, and the spreadsheet will make the calculation automatically whenever `A1` or `A2` is set or changed. Lauesen et al realized that this type of programming is within the scope of what a local developer can handle, e.g. expressing the values of properties through a formula that allows them to reference other things. Therefore, they created their own formula language, a DSL for uVis that has a syntax inspired by spreadsheets like Microsoft Excel, the Visual Basic programming language, and added additional operators for querying databases and traversing database table relations, as we saw in the example. To get a sense of the entire language, see the uVis reference card attached in appendix.

However, the dependencies between properties make creating components quite complex.

2.2.1 Creating instances in the right order

If there are no dependencies between properties, we can create the component tree starting from the root of the template tree. However, creating a component tree will most likely involve querying a database once or twice, and if those queries, defined by the `Rows` property, include a reference to a component property, the component property and the component it belongs to, must be created first. For example, if the `Rows` property of `bxIntake` referenced the `Text` component property on `txtPtName` in the component tree, the uVis kernel would have to jump over to `txtPtName`'s branch of the template tree, and create its component and `Text` component property first, before it could continue with creating `bxIntake` components.

The same problem exists for all the other properties. In general, a formula can contain references to any component's properties, even if that component is not in the same branch of the component tree. Therefore, the uVis kernel must make sure that components are created in the right order, and that component properties are created in the right other, otherwise a referenced component property might not exists.

This is also true when it comes to adding visual components to canvases. We must add the canvases in the right order, followed by the other visual components.

2.2.2 Cyclic dependencies

The cross dependencies described above can lead to cyclic dependencies, both between components, i.e. when creating the component tree, and between component properties, when they are being evaluated. This is a classic spreadsheet problem, where cell `A1` references cell `B1`, who references cell `C1`, who references cell `A1`, i.e. `A1 = B1`, `B1 = C1`, and `C1 = A1`. The uVis kernel needs to detect such conditions; otherwise, it is possible for a local developer to create an endless loop.

2.2.3 Spreadsheet algorithm

To detect cyclic dependencies, the uVis kernel uses a standard spreadsheet algorithm. When a spreadsheet starts evaluating a formula in a cell, it first marks the cell as *"visited"*, and when the evaluation of the formula finishes, it marks the cell as *"updated"*. If at any point during the evaluation of the formula, the spreadsheet tries to get the value of another cell that is marked as *"visited"*, i.e. a cell already visited during the evaluation, it knows that there is a cyclic dependency. The uVis kernel uses this technique when creating components and component properties.

2.2.4 Keeping the component tree up-to-date

Since the component tree is based on the template tree plus data from one or more data sources, the component tree must be kept up to date if data changes, or more precisely, the number of bundles and the items in the bundles must be updated when data changes. If for example, a new medicine order was added to the patient used in our example, a new `lblMedicine` component would be added to the `lblMedicine` bundle, and the bundles for `lblMedOrder` and `bxIntake` would be updated as well. Updating and deleting data has a similar effect on the bundles in the component tree.

2.2.5 SQL query generation

Another feature of the uVis DSL is the ability to express database queries using a simple notation. For example, in `frmPatient -< MedOrder`, the `-<` operator is a left/inner join between the `Patient` table and the `MedOrder` table.

This means that the uVis kernel must build a set of database queries when it is about to create a component tree, so the uVis kernel can query the database(s) and retrieve the data the templates need. The uVis kernel tires hard to limit the total number of queries it must send to the database to create the component tree. For example, a template will retrieve all matching rows in one request instead of performing one request for each component it must create, e.g. all `MedOrders` that belongs to the selected patient.

To help generate the SQL queries, the uVis kernel uses a data-map file, known as a **.VISM file**. It is, as the .VIS file, a plain-text file in ini-file format. It contains, among other things, connection strings to databases, descriptions of tables, their keys, and the relationships between tables. It can also define renaming of columns and tables that might have obscure names. While most of this information can be obtained automatically by inspecting the database schema, sometimes databases schemas will be missing e.g. obvious relationships between tables. With a .VISM file, it is possible to describe these relationships without modifying the database.

2.3 uVis Studio

Even though creating a web version of uVis Studio is outside the scope of my thesis, it deserves an introduction.

uVis Studio is a Windows application, seen in Figure 5 below, that is used by the local developer to create uVis applications. The primary features of uVis Studio includes support for the drag-drop-set-property workflow for creating user interfaces, support for "IntelliSense"-like text completion that makes it much easier to write the property formulas, and the ability to instantly see how the data from the connected database is visualized during development. As soon as a template is added and its `Rows` property is set, uVis Studio will query the database and update the rendering. This feature makes it very intuitive for local developers two create visualizations.

2.4 Development scenario with uVis Studio

To illustrate the power of uVis Studio, let us glance over a development scenario, where the bronchoscopy form in Figure 5 is created.

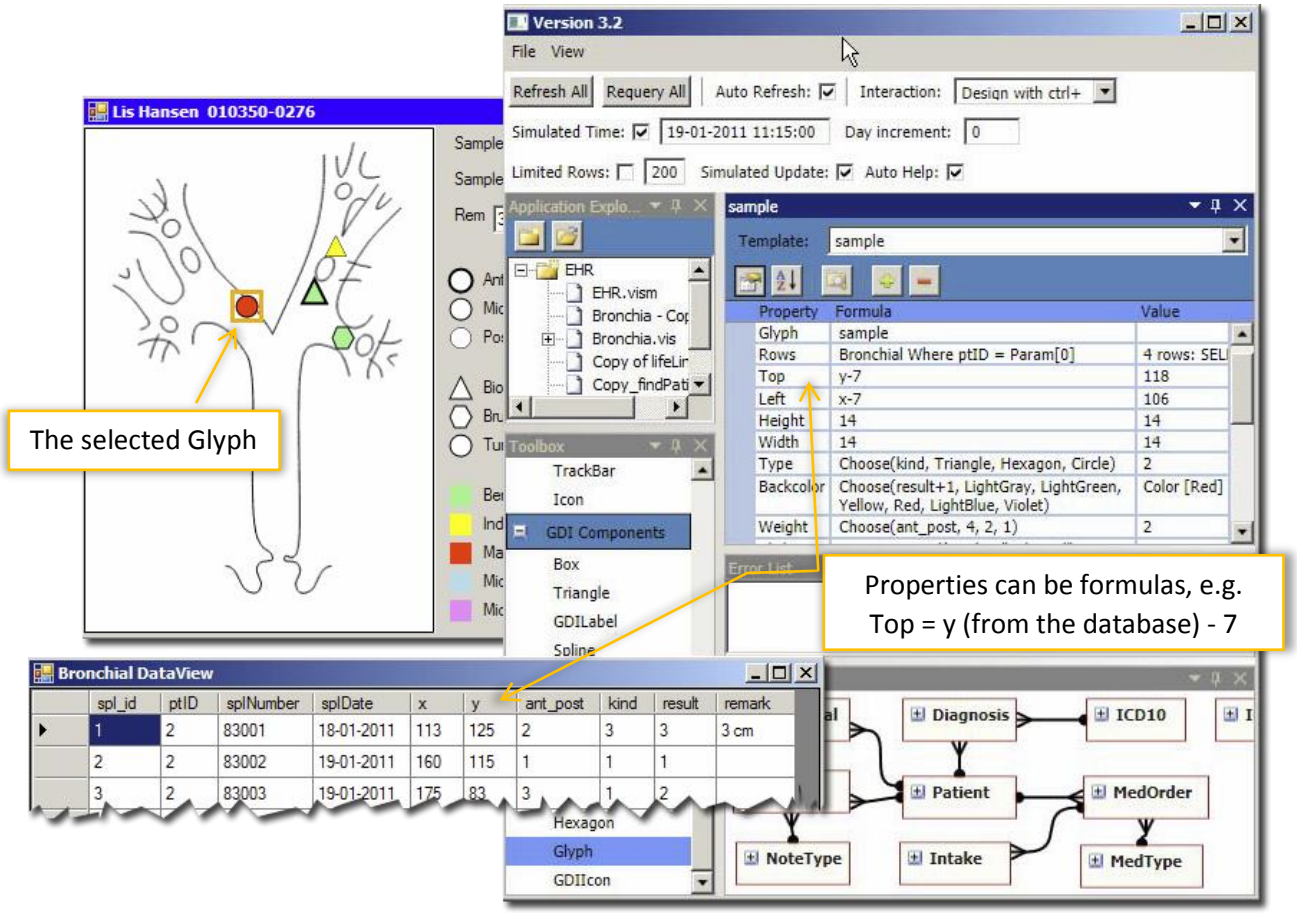


Figure 5: Screenshot of uVis Studio in action. The local developer is updating the properties of the selected glyph in the canvas (top left). The `Rows` *property is set to select each row from the Bronchial database table in the properties list (top right), where the* `ptID` *(patient id) column matches a certain value. The* `Top` *property is referencing the 'y' column in the Bronchial database table, and offsetting its value with* `-7`*. Source: (Lauesen, et al., n.d.).*

Connect to database: The local developer opens uVis Studio, and loads a .VISM file provided by the IT department. The .VISM file gives uVis Studio access to existing databases in the organizations that it uses to draw a data-map in uVis Studio (lower right corner, Figure 5). At this point, only the primary uVis Studio screen is displayed (right, Figure 5) and an empty screen/form (the screen to the left in Figure 5 shows the final result).

Adding templates: Next, the developer can add templates from the toolbox to the empty screen using drag-and-drop and set each component's properties. When a template is added, uVis gives it default property values so components can be created and rendered on screen. For example, if the developer wants to use glyphs to represent the test results, he would drag the Glyph template onto the canvas. Then he would set its `Rows` property, binding it to the Bronchial database table. That causes uVis Studio to query the database and retrieve data form the database table and generate as many glyphs on the screen as there are matching records in the table. He would then update the `Top` and `Left` property formula to position the glyphs correctly on screen on top of the bronchia drawing, as we see in Figure 5.

Testing: Once the new screen is finished, the developer can do additional testing, e.g. by using a test database with missing/mangled data, to test how the screen behaves when expected values are missing. Switching between databases, e.g. production and test, is easy, it is just a matter of changing a connection string in the .VISM file.

Deployment: When the testing is done and the developer is satisfied, he can save the screen as a .VIS file, that he can distribute to with the .VISM file the end users, e.g. via the company's network.

What we have covered so far should be enough to have a general understanding of what the uVis kernel is and what it does, and the complexities associated with building it. For more details on uVis and the uVis kernel, I refer you to (Lauesen, et al., n.d.).

3 Implementation

To design and implement a kernel for uVis.web, I had to consider the overall uVis.web system, since the requirements of the latter directly influence the functionality of the first. Consequently, this section will first cover the requirements, design, and architecture of the overall uVis.web system as I imagine it; next, it will discuss the implementation of the uVis.web kernel.

Before we enter the web-world, let us resolving a few naming conflicts. When discussing uVis in the previous section, we used the words *form* and *canvas*. To recap, a *form* in uVis terms means a single program window that can be moved around on screen independently of other uVis forms. In addition, it has its own template tree and originates from its own .VIS file. A canvas is a special component that is rendered on a form that other components can render themselves on. When I use the terms *form* and *canvas* going forward, these definitions still apply, and not the HTML version of a form or canvas. If I refer to either a form or a canvas in HTML, I will explicitly refer to them as a HTML form and a HTML canvas.

3.1 The uVis.web requirements

The original uVis requirements (Lauesen, 2011) do not discuss the requirements for a web version of uVis, so I built on top of the uVis requirements and created a set of additional requirements specific to the web version of uVis.

W.0: uVis.web must run in a web browser

This requirement is obvious. However, the implications of it are important. Existing data sources, e.g. databases, which uVis normally interact with directly, must be exposed as web services. JavaScript running in a web browser can only make HTTP (including XML-RPC) and Web Socket connections to another server. It makes it impossible to connect directly to existing databases using e.g. TCP/IP as uVis does. A web service, which we will call a *data web service* (DWS), is needed to bridges the gap between the existing data source and the JavaScript running in the browser.

W.1: A uVis.web application web server (AWS) should not have to run from the same domain (or server) as the data web services (DWS), e.g. be hosted on the same IP address or domain

It is unlikely that it is always possible to have an *application web server* (AWS) and a *data web services* (DWS) on the same server. DWS's will most likely be placed on servers close to the backend data store (database) they expose. It is also likely that the DWS administrators will want to keep DWS separate from AWS's, both due to security and scalability. In addition, W.1 and W.2 are tightly connected; fulfilling W.2 is not possible without W.1.

W.2: uVis.web should be able to connect and aggregate data from multiple DWS's

The ability to work with data from multiple data sources within a single uVis.web application provides intriguing possibilities to create visualizations not otherwise possible, especially in corporations where data is placed in many different database systems.

To leverage this ability, the uVis.web kernel should allow for in-memory querying, projection, grouping, aggregation, and joining of data from different data sources on the client.

W.3: uVis.web Studio should be built as an HTML5 solution and be able to run and work directly on a running uVis application

Products such as Cloud 9 IDE[2] demonstrate that it is indeed possible to build a complex application like uVis.web Studio directly in the browser, and I believe the big advantage of having it there is the possibility of truly seeing an application change live on the screen as the user changes properties and writes new formulas. This is how uVis Studio works in the Windows version and is a great benefit to the target audience.

Note: I did not create a web version of uVis Studio.

W.4: There must be no dependencies on browser plugins such as Java, Flash, Silverlight, etc.

Browser plugins have historically existed to provide features that were not otherwise available to web developers, such as playing audio, video, access to microphone and webcam, and other parts of the users system. With these features now supported directly in all modern browsers, and with a ban of any browser plugins on many popular mobile devices, it would be unwise to have uVis.web depend on e.g. Flash or Java. In addition, these plugins do not provide any necessary technical benefits related to uVis.web or uVis.web Studio, besides allowing developers familiar with those technologies to reuse their knowledge.

W.5: The architecture of uVis.web should support streaming or push-based DWS's

A push-based DWS is preferred whenever an uVis application requires continues updates from the data source or where new data must be displayed on the client as soon as they are available. Supporting push-based DWS's will make it possible to create applications like e.g. stock market trackers or live updated dashboards.

W.6: uVis.web should be compatible with touch-based devices (e.g. tablets and phones)

Many devices, both mobile and stationary, support touch input, e.g. smartphones, tablets and big wall-mounted monitors. This can be considered a pseudo-requirement since touch input is usually handled by the underlying operating system, but uVis.web should allow the developer to explicitly listen for touch events, including multi-touch events.

W.7: uVis.web should support mobile clients

More and more mobile devices are introduced in the workplaces that uVis.web is targeted at, so it is only natural that uVis.web applications is able to run directly on these types of devices.

W.8: uVis.web should allow third-party add-on's

The uVis.web architecture should make it possible for third-party developers to develop new components or add support for additional data web services.

[2] https://c9.io – Cloud 9 is a full featured IDE that runs entirely in the browser, based entirely on HTML5. It has a rich development experience with debugging, code highlighting, IntelliSense, project management, among other things.

3.2 The uVis.web architecture

The uVis.web architecture follows the client-server model commonly used on the Internet. In Figure 6 we see two data web services (DWS) that provide data to a uVis application running on the clients (tablet and laptop). The application web server (AWS) hosts the uVis kernel, as well as the definitions for the uVis applications running on the clients.

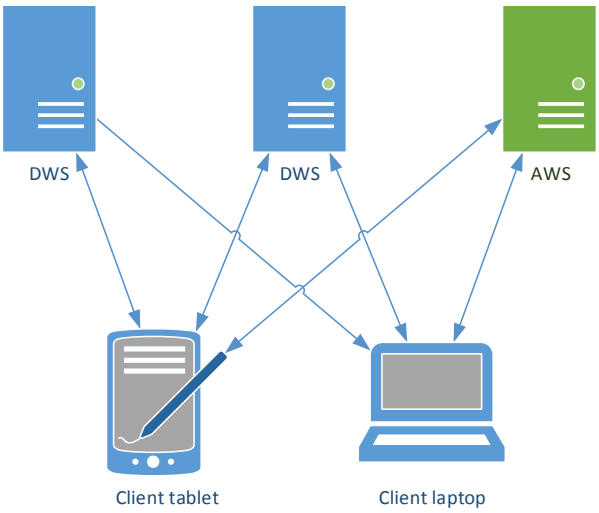


Figure 6: Client-server architecture of uVis.web.

The general responsibilities of each component are as follows:

Data web service: A DWS should just be an endpoint for whatever data it is exposing; it should not be tied to uVis.web or have nor need knowledge of uVis.web to work, i.e. be independent of uVis.web and work with any client that supports its interface. It is up to uVis.web to support the different technologies behind DWS's, e.g. by provide a way for the uVis.web to translate a database query written in the uVis DSL to the native query format of the web service. As specified in requirement W.0, a DWS must be available over HTTP(S).

Application web service: The AWS is responsible for hosting the uVis.web kernel, and any configuration/user database it needs to authenticate users and assign them different rights to a uVis.web application, e.g. *User* or *Developer*. An AWS can host one or more uVis.web applications, i.e. collections of .VIS files, and the user then selects which application to start, either by navigating directly to the applications URL, e.g. `http://uvis.local/EHRSystem`, or by selecting the desired application via a list of applications the AWS provide.

If a DWS does not provide adequate metadata about the entities it is exposing, e.g. to enable uVis.web Studio draw an entity diagram as seen in the lower right corner in Figure 5, the AWS will also host what corresponds a .VISM file in uVis. This file describes a DWS's entities and their relationship, and it could provide additional abstractions/renaming of the entities for the user, and contain additional data validation rules. A AWS/DWS administrator or data architect usually creates the .VISM files.

Client: The clients need only have a modern web browser.

Location of servers: When looking at Figure 6, there is no indication of where the web services/servers are located physically and/or on what networks. This is because it is possible that they can be on different networks, e.g. a DWS could be publicly available service on the internet, and the AWS could be hosted

locally inside a company. The opposite is also likely, e.g. some vendor hosts and run an AWS and provide uVis.web as a service to customers, who just manage and run their own internal DWS's.

Some of the requirements listed in the previous section as well as the proposed client-server model above offer some challenges that I will discuss and outline solutions for in the following sections.

3.2.1 Multiple DWS's and different servers for AWS and DWS

Requirements W.1 and W.2 are in conflict with the normal security restrictions imposed by browsers, namely the *Same-Origin Policy*. The same-origin policy says that JavaScript, e.g. the uVis.web kernel downloaded from the domain `http://uvis.local`, cannot retrieve any data from a web service hosted on another domain, e.g. `http://dws1.local`.

Luckily, this is a common enough problem in web development that solutions exist, and depending on the technology behind a DWS, different ones can be employed.

AWS can proxy communication with the DWS

A solution often used when no other options are available is to proxy communication between client and web service, e.g. let the AWS proxy requests between the client and the DWS. See Figure 7 to the right. This mitigates the issue since the same-origin policy does not apply when browser sends DWS requests through the AWS that served the uVis.web kernel script that is making the request. However, several issues make the solution far from ideal.

- All DWS bound traffic has to travel through the AWS. That puts extra requirements on the AWS in terms of bandwidth and other resources, and effectively makes the AWS a bottleneck. In particular, the added network latency of first going to the AWS instead of directly to the DWS might be a problem.
- The AWS must also be able to connect to the DWS. It might not always be possible in all scenarios, e.g. if the AWS is hosted in the cloud and must proxy for a DWS behind a firewall on an internal network.
- The security of the DWS and its data may also be compromised. Consider the scenario where a DWS provides several different levels of security to data it exposes, and each user of a uVis.web application provides his own credentials to gain the access granted to them. If their communication with the DWS has to travel through the AWS, an AWS administrator could gain access to data that would otherwise be out of reach. The proxy method is in essence a sanctioned man-in-the-middle attack.

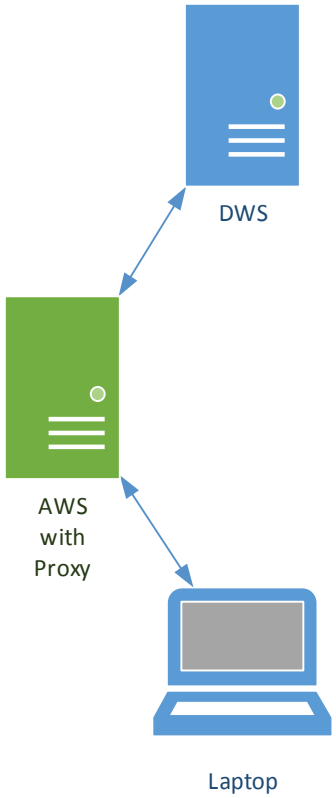


Figure 7: An AWS acting as a proxy for a DWS, making it possible for a client to connect and communicate with it.

JSON with padding

Another common solution seen in the industry is using JSON with padding, or JSON-P. JSON-P was invented as a patch to some of the issues mentioned with the proxy solution, and has gained quite widespread use since it does not require any changes to web browsers to work.

A script coming from an AWS is able to retrieve date from a DWS by creating an HTML `script` tag in the current browser document, setting its `src` attribute to the URL that to point to the resource it wants. The URL is padded with a `jsonp` query parameter that points to a function that should be executed once the request completes. The server then returns the requested data wrapped in a function call to the function specified in the `jsonp` query parameter.

For example, if we imagine DWS1 has patient records, then a request to `http://dws1.local/Patient(1)` should return the patient with ID = 1. To get the data using JSON-P, uVis.web creates a script tag, as that in Listing 1 below, and adds it to the document. This result in *the web browser making the request*, not the uVis.web kernel, and the uVis.web kernel can then wait for the `parseResponse` callback function, which it has created in the global scope in the browser, to be called. If this seems like a hack, it is because it is.

```
<script type="text/javascript"
    src="http://dws1.local/Patient(1)?jsonp=parseResponse">
</script>
```

Listing 1: Example of HTML script tag inserted into a DOM to create a JSON-P request.

As always, there are caveats with the solution:

- From a security aspect, we have to completely trust the DWS and that it has not been compromised, since we are allowing it to send us arbitrary JavaScript of its choosing and we gladly let our browser execute it, i.e. opening us up to cross-site request forgery (CSRF or XSRF) attacks.
- Another issue is that we are limited in types of HTTP requests we can make, namely `HTTP GET` and `HTTP POST`. This might not be problems with some web service technologies, while others require the use of `HTTP PUT`, `HTTP DELETE`, etc. to get the full functionality.

It does however mitigate the network performance problems and the same-network issues that the proxy solution has, i.e. by allowing us to connect directly to the DWS.

Cross-origin resource sharing

CORS (Cross-Origin Resource Sharing) mitigates many of the problems with both the proxy and JSON-P solution, and all the latest versions of browser support it today.

When the uVis.web kernel request patient 1 using a standard AJAX[3] request to `http://dws1.local/Patient(1)`, the browser appends a HTTP header to the request with the origin of the script making the request, e.g. `Origin: http://uvis.local`, and the DWS can then decide if the origin is allowed to make requests to it. If it is, the DWS responds with an `Access-Control-Allow-Origin` header, e.g. `Access-Control-Allow-Origin: http://uvis.local`. Then the browser knows that the request is allowed and proceeds.

This does solve the problems with possible CSRF attacks and allow all HTTP methods to be used, but it does require that both the server and the web browser support CORS, which is not always the case. The downside is the pre-flight checks, i.e. the browser first asks permission, and if it gets an OK, it sends the request, but that is a relatively small problem compared to the other proposed solutions.

[3] An AJAX (Asynchronous JavaScript and XML) request is the recommended way for JavaScript to send and receive data from a web service. Despite the name, JSON is often the data format used, not XML, and a request does not have to be asynchronous, although there are seldom any reason for it not to be.

That said, with all three solutions in our tool belt, it should allow connections to virtually all DWS's imaginable – it is just a matter of implementing support for each of these in uVis.web.

In the current prototype, I just assume that the browser and the DWS support CORS. It is however possible to support both JSON-P and the proxy solution without any changes the prototype.

3.2.2 Mobile clients, offline mode and loss of connection

When creating applications for mobile devices, one has to consider the possibility of intermittent connections to servers or complete connection loss for periods of time. There are also battery life considerations and the many different screen sizes and orientation of the device.

With requirement *W.7: uVis.web should support mobile clients*, we have to consider these issues, especially if a uVis.web application is used to view information that might be critical and needed at a moment's notice, e.g. patient charts viewed on a tablet device, where a lost connection to the DWS or an empty battery is bad.

Store data temporarily when connection is lost

Luckily, HTML5 provides us with a relatively well-supported standard for storing data on the local device's disk, the Web Storage standard. It offers us basic dictionary storage (`get(key)`, `set(key, value)`) on the client, but can only store string values. That means complex objects must be serialized to JSON before being stored.

If the uVis.web kernel detects a connection loss to one or more DWS, it can store changes to the data the user is working on the client, and when the connection is back, the changes can be pushed to the back-ends. This does complicate the issue with concurrent editing of the same data from multiple clients, so the developer will most likely need to have various options to choose from, e.g. optimistic/pessimistic concurrency.

Running an uVis.web application without any servers

Until now, we have only handled connection loss while the user is using a uVis.web application, but what if the connection to the AWS and one or more DWS is down before the user tries to start the application. For this, HTML5 also has a solution – an application cache manifest. It is possible to create a manifest file that details all the static files required by a web application to run, e.g. JavaScript files, image files, HTML files, and CSS files. This enables browsers that support the standard to create a local copy of the uVis.web kernel files, and use these if the browser is unable to connect to the AWS and download the runtime files from it. Similarly, the web storage standard described above can be used to store data from a DWS in the client, i.e. making a local copy of a data set available, e.g. a patient's medical data, even if the network connections is down before the uVis application is started.

Hence, an application cache manifest combined with local storage, should make it possible to create uVis.web applications that can both be started and can keep running in the face of connection loss.

Throttling data connections

To preserve battery on some devices, uVis.web could try to detect mobile devices and automatically throttle connection to DWS's to save battery. However, there is no API currently available in HTML5 that will provide a web application with notifications from the host device if that device is running low on

power[4]. Consequently, it might be better to simply give the developer the option of specifying a "battery saving mode" for an application, if he/she knows that battery life can be an issue with a specific uVis.web application, e.g. if it is only used with certain mobile devices.

Screen sizes and device orientation

When developing websites in 2013, a good developer will follow the responsive design pattern. That enables him or her to specify different layouts for different screen sizes and device orientation. The different layouts are specified via CSS, so the content (HTML) can stay the same, but its position and look on the screen can change. Thus, uVis.web just has to support the CSS syntax used to specify the different layouts depending on screen size and orientation. The challenge will be in the uVis.web Studio, where Studio will have to provide the developer with a way to specify different layouts for different screens.

The example in Figure 8 shows the same website on three different devices. We see that the elements of the website, which are represented by the colored boxes, are moved around and resized according the screen size. In uVis terms, the colored boxes could be e.g. forms that would change their position and size, depending on the space available to them.

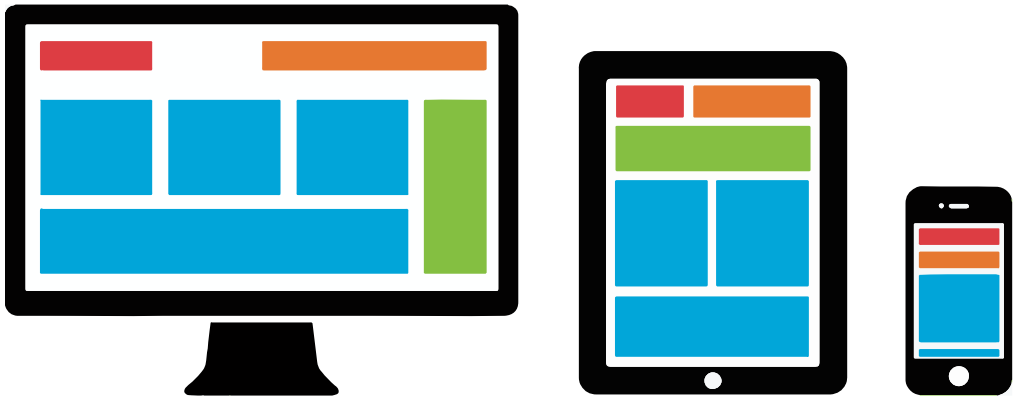


Figure 8: Example of a responsive web design that targets three different screen sizes. Source: connect.icrossing.co.uk.

The current uVis.web kernel does not explicitly handle the mobile client issues discussed above; however, all solutions can be implemented without changes to the kernel.

3.2.3 Authentication, authorization and data security

I have identified three challenges with data security, authentication, and authorization.

Users can have different authorization levels

It is normal that different users have access to different datasets within a database, and this translate into different datasets on the DWS's in the uVis.web world. It is normally the database administrator, who also provides users with credentials, that define these authorization levels. This is generally not a problem since it has nothing to do with uVis.web – all of that happens on the DWS.

However, since uVis.web is used to create visualizations, or more generally, UI's, that are driven by data that is available, the developer needs to be aware of the different permission levels a dataset can have, and handle the cases where the user does not have access to a particular data(sub)set.

[4] Mozilla, the maker of the web browser FireFox, and the upcoming mobile operating system FireFox OS, is actively working on a JavaScript API that will allow web developers to retrieve battery information. However, the API is not expected to be generally available in wide number of browser anytime soon.

While it is not impossible to handle such scenarios in the current implementation, an extension to uVis.web should be developed that would allow the developer to handle the cases where a user does not have all the expected access more easily. A simple addition would be to propagate `401 Unauthorized` HTTP status responses from DWS's that would indicate that the user is not authorized to perform the operation. That way, a developer could specify, not just the default value to use if the requested data is NULL, but also what to do, if the requested data is not available to the active user.

Single sign-on may not always be available

Inside companies, it is normal to have single sign-on solutions deployed such as Active Directory[5], so users do not need to remember passwords for all the systems they interact with.

However, if an uVis.web application connects to one or more DWS's that are not part of the company's single sign-on solution, e.g. because they are hosted externally, the uVis.web application can gather the users DWS credentials at runtime and provide it to the DWS when communicating with it. If the uVis.web application uses more than one DWS that requires explicit credentials from the user, the user will have to provide many credentials each time they use the application.

A solution to this annoyance could be a credentials store provided by the AWS for each user, and the user would only have one set of credentials, a username and password used to encrypt the content their credentials store. This would prevent anyone with access to the AWS's database from getting access to the users DWS credentials. The DWS credentials should only be decrypted in the browser at runtime when they are needed. I have sketched the login workflow to a DWS using a credentials store in Figure 9.

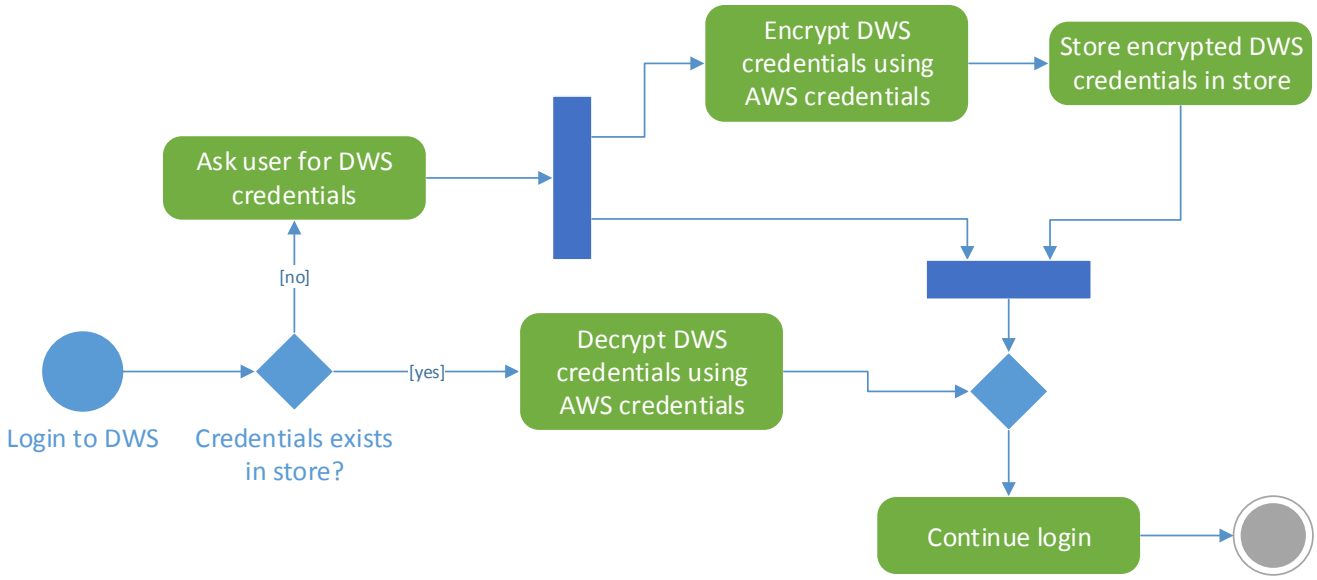


Figure 9: UML activity diagram illustrating the workflow for logging in to a DWS with credentials. This assumes the user has already provided his AWS credentials earlier.

Security of data stored in offline clients

As discussed in a previous section on mobile clients, sensitive data from a DWS can potentially end up being stored on the client, if the client device has local data storage support. Since data stored using the Web

[5] Active Directory is a directory service created by Microsoft included with Microsoft's Windows Server products. It is commonly used in corporate networks based on Windows Server technologies.

Storage API is not encrypted, anyone who gets access to the device can potentially get access to the sensitive data by looking for the data on the device's hard drive.

The solution to this is similar to the previous, i.e. to only write data to the client's local storage encrypted with the credentials of the DWS who owns the data. The data would only be decrypted in-memory when the uVis.web runtime was using it.

The current prototype assumes there is no explicit authentication required to talk to a DWS. However, if the browser is able to authenticate a user automatically against a protected DWS, e.g. using single sign-on, the uVis.web kernel is able to connect to it.

3.2.4 The trouble with separate browser windows/tabs

uVis and uVis Studio show each form as its own program window, i.e. they are *not* a part of a larger "uVis window" (see Figure 1 and Figure 5 on pages 5 and 13 respectively), and the uVis kernel will keep track of the open forms (windows). It allows the different forms to interact with other form's template and component trees. This allows the user to organize the open forms on his monitor as he wants, and have many open forms at the same time, if the scenario requires it (and he has the screen real-estate).

In uVis.web, it is possible to continue to use this pattern, i.e. open a browser popup window for each form, with limited chrome. However, due to the way browsers work, each browser window and thus form has to have its own instance of the uVis.web kernel loaded and it is not possible for components in one browser window to interact with components in another browser window directly, i.e. no shared memory across browser windows/browser tabs. In addition, popup windows or just multiple windows do not work well on most tablets and smartphones since two browser windows cannot be view on screen at the same time.

That said it is possible to establish communication between browser windows or browser tabs using the `localStorage` API available in modern browsers. It would require change to the uVis.web kernel, having it use `localStorage` to marshal communication between components in browser windows – enabling a component or property to depend on another component or property in another window.

The alternative is to emulate a window manager via JavaScript, having the multiple forms open as movable containers inside a single browser window. This way, each form is able to interact directly with other form's template and component trees, because they exist in the same browser window and have access to the same memory objects. In addition, there only needs to be one instance of the uVis.web kernel downloaded and running.

A third option is to completely break with classic window metaphor and allow users (or developers) to organize forms within a browser window using other methods than drag-and-drop and resize of overlapping forms/windows. It could for example be switching to a tile metaphor, where tiles partitions the screen into non-overlapping areas in which forms can be placed. However, dropping the window metaphor is a research project by itself – understanding usability implications is just one of the interesting problems that needs to be solved.

The solution currently used in the prototype is to open each form in its own window manually, and there is no support for cross-form communication. However, adding more options for cross-form communication or run all forms inside the same browser window can be done without changing the core concept of the uVis.web kernel.

3.2.5 uVis.web application distribution packing

A uVis application is distributed as a directory, either on a network or on a local computer, generally containing a .VISM file and numerous .VIS files, one for each form in the application. The developer specifies a start-up form in the .VISM file.

In uVis.web, we take a slightly different approach. There are three differences:

1. uVis applications are served through an AWS, as already established
2. The text format is JSON[6] instead of ini-file based
3. There is a global `app.json` file for each application

A AWS can host many uVis applications and each uVis application has its own unique URL, e.g. `http://aws.local/EHRSystem` and `http://aws.local/OperatingRoomDashboard`, making the individual applications accessible directly. This corresponds to e.g. `C:\uVisApps\EHRSystem` and `C:\uVisApps\OperatingRoomDashboard` in uVis, so no real difference here, only the addressing schema and communication protocol.

The .VIS and .VISM text format is different however. In uVis.web, we use JSON instead of the ini-file based format used with uVis. There are plusses and minuses attached to this decision. On the plus side is parsing speed. All modern browsers have a native parser for JSON, making it very fast to convert a JSON string into JavaScript objects. On the minus side there is the verbosity of JSON compared to the ini-file format, which results in more characters being send over the wire and makes it less readable. In the end, I choose the JSON format over creating my own ini-file format to JavaScript object mapper. In addition, all modern webservers can automatically compress text files before sending them to the client, which reduces the impact of using JSON. See Listing 2 below and Listing 3 on page 25 for a comparison of the two formats.

```
Rows: Patient where ptID = param[0]
Width: icon1!width + 50 + c!width
Height: icon1!height + 50
Text: ptName & "  " & (CPR Default "")
```

Listing 2: An uVis template definition in the original uVis ini-file format and uVis syntax.

The third thing that is different is the use of an `app.json` file that contains global information about an application. In particular, it currently has the following info:

- Application title and (optional) description
- List of data sources (type information and URL to each DWS)
- List of forms (URL to each forms JSON file, and an indicator of which forms are initially visible)
- Collection of global application event handlers
- Collection of global application properties

The `app.json` is to a uVis.web application what the .VISM file combined with the directory hosting the .VIS files is to a uVis application.

[6] JSON (JavaScript Object Notation) is based on a subset of JavaScript and is the by far most commonly used data exchange format between JavaScript based applications and their backend servers. Today, JSON is used in many other venues than the web, due to its lightweight nature, that it is easy to parse by machines and humans alike. See more at http://json.org/

```
{
    "rows": "Patient where ptID = param[0]",
    "properties": [
        {
            "id": "width",
            "formula": "icon1!width + 50 + c!width"
        },
        {
            "id": "height",
            "formula": "icon1!height + 50"
        },
        {
            "id": "text",
            "formula": "ptName & "  " & (CPR Default \"\")"
        }
    ]
}
```

Listing 3: The same template definition as in Listing 2, this time in JSON format. Notice that the uVis formulas are the same as in Listing 2, it is just the wrapping that is different.

3.3 From development to deployment to end user

When an AWS sends a .VIS file to a client, it will compile the formulas written in uVis DSL to JavaScript code snippets first, so the uVis.web kernel on the client does not have to do it on the client at runtime. That should improve the performance on the client and unless the client is a developer machine, the client has no need for the original uVis formulas. There is little overhead for the AWS, since it can cache the compiled formulas and only invalidate its cache when a developer makes changes to the original formulas. Take for example the last formula from Listing 3 above:

```
{
    "id": "text",
    "formula": "ptName & "  " & (CPR Default \"\")"
}
```

That would travel to the client precompiled looking like this:

```
{
    "id": "text",
    "formula": "row.ptName + " " + (row.CPR || \"\")"
}
```

Because we compile the formulas on the AWS, the AWS must also create the template tree structure, since the formulas is what determines the structure of the tree. This is not a problem however, since JSON is quite good at representing tree structures.

When the form definition arrives on the client, the uVis.web kernel moves through the exact same phases as the uVis kernel, e.g. convert the form definition to a template tree in memory, instantiate the component tree based on the template tree and data, and create a rendering of the component tree to the users screen.

Form definitions send to a developer machine running uVis.web Studio will still contain uncompiled formulas, and uVis.web Studio will have to do the compiling when needed. See Figure 10 for a visual explanation.

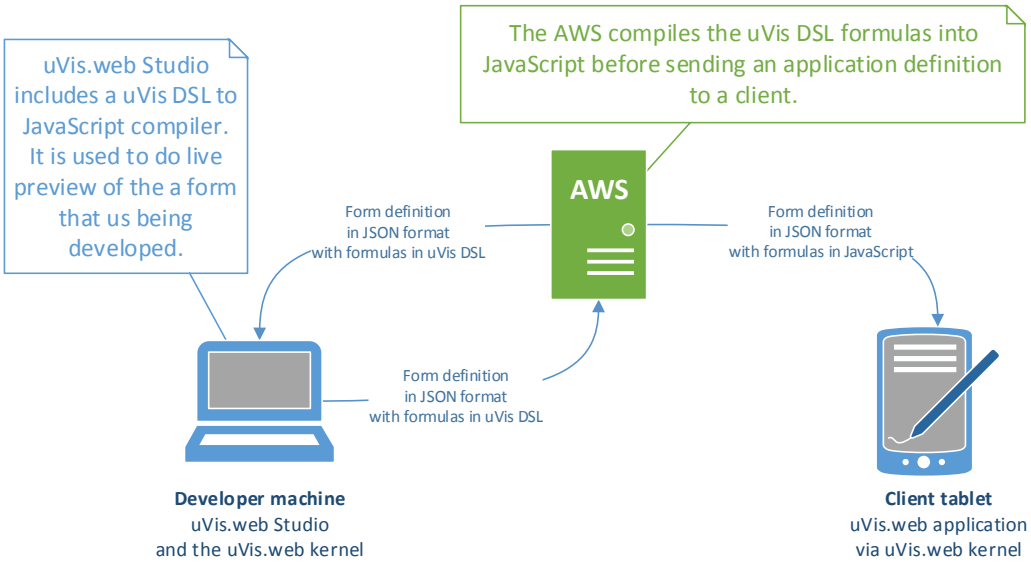


Figure 10: Diagram showing how form definitions flow from developers to end users.

3.4 The uVis.web kernel

In this section, I will talk about my implementation of the uVis.web kernel, about how it works and where it diverges from the kernel in uVis. What makes implementing a kernel for uVis non-trivial is the formulas used to express the value of properties and database queries and in particular, the dependencies they can have on each other. To recap, these are the challenges we have to deal with:

1. Properties must exist before they can depended on, i.e. the order we create components and component properties in matter.
2. Cyclic dependencies must be detected. E.g. `A` depends on `B` and `B` depends on `A`.
3. When one property is changed, we must somehow reevaluate all other properties that depend on it.
4. Dependencies can change at runtime when updates occur, e.g. due to `if-then-else` in formulas or a `Canvas` property changing value.

uVis have all of these challenges in common with spreadsheet applications, which is really no surprise, since the concept behind the uVis formulas are inspired by how spreadsheets works. The reason why we cannot just use the developer's manual for creating spreadsheets is that there are also differences. All cells in a spreadsheets exists from the beginning, as soon as the user creates a new spreadsheet, the equivalent in uVis, component properties, are created depending on database queries. So how do uVis and uVis.web handle these challenges?

In section 2.2.1, we discussed how the uVis kernel makes sure components and component properties are created in the right order. In section 2.2.2 and 2.2.3, we discussed how cyclic dependencies are detected using a standard spreadsheet algorithm, so we will not cover those again.

For dealing with changes, the uVis kernel uses a straightforward approach that does not force it to keep track of what formulas depend on, and what should happen when they change. It simply reevaluates all formulas. This works well because it takes relatively little time to reevaluate every formula, even in a complex form like the medical history form shown in Figure 1, which has many formulas. The uVis kernel is also smart enough not to requery databases unless the formula for a `Rows` property resulted in a different SQL expression than before the reevaluation. In addition, the Windows Forms framework will only redraw

visual components on screen if their property values actually change, making uVis able to redraw a form fast without a noticeable “blink”. With an ordinary PC and a local Microsoft Access database, uVis is able to refresh the timeline form in about 80ms (Lauesen, et al., n.d.).

The uVis.web kernel conceptually uses the same approach for the first challenge as the uVis kernel. However, for dealing with changes to properties and data, I choose to go a different route. To understand the reason why, we must look at how JavaScript executes in the browser.

JavaScript is inherently single-threaded, and more importantly, when running in a browser window/ tab[7], it is running on the same thread that is also used to render output to the screen and used to dispatch events, e.g. mouse and keyboard events. Because the browsers rendering engine and JavaScript (e.g. event handlers) run on the same thread, it generally means that the user is unable to interact with the content on screen when JavaScript code is executing. Textboxes stop accepting input, scrollbars are stuck, mouse clicks do not register, etc. In addition, the rendered output, e.g. the visualization of the DOM, will not update while JavaScript is executing, which can result in lagging or slow updates of content on screen. Olav Junker Kjær from Opera Software (makers of the Opera web browser) writes that *event handlers should never be time consuming*. He adds that we should be especially wary of *synchronous web service requests*, since they might cause a noticeable delay that blocks the browser or document window (Kjær, 2007). Web browsers are able to handle the scenario where multiple events are triggered at the same time. It does this by adding event handlers to an *event queue*, from which it uses an *event loop* to execute the event handlers asynchronously on the UI thread, one at the time. Each browser window/tab has its own event queue.

Since changes that cause a reevaluation of formulas in uVis are mostly user driven, e.g. via direct input (mouse, keyboard, etc.) we must thus try to limit the runtime of code that handles these changes, especially if these changes results in queries against web services, that might take seconds to complete if the network or DWS is congested. This is especially important for devices such as tablets and smartphones that suffer from both slower processor and network speeds.

Going asynchronous

The solution to a blocked UI because of synchronous blocking web requests is to do the opposite, perform web requests asynchronously. This approach is in no way novel, performing web service requests asynchronously is by far the most common way to query web services via a web browser and JavaScript, it does however require us to deal with asynchronicity in our code. The normal way to deal with asynchronicity is by providing the asynchronous request with a callback function that is added to the event queue when the request completes. When the callback function is executed by the event loop, it is passed the result of the request. This paradigm should not be too unfamiliar to developers used to UI programming. Event and event handlers work in the same way. Event handlers are just callback functions that are added to the event queue when the event it is attached to is triggered.

A great benefit with doing asynchronous web service requests is that we can now offer the user multitasking. The user can continue to interact with a form while a web service request is in progress in the background, e.g. view other data on screen, change values, possibly even cancel the request or replace it with another. The user can also scroll the browser window, move other components around, trigger events on other components, and interact with other forms visible in the same browser window. None of that would be possible with synchronous web requests.

[7] In all modern desktop browsers, browser tabs are just browser windows group together. All browser tabs have their own OS process and own resources, and will continue to function even of other browser tabs crash.

To utilize the advantages of doing asynchronous web requests, the code that depends on the web requests should also be designed to be asynchronous; otherwise, we just move the point where we block. Luckily, JavaScript with functions as first class objects makes it easy to practice the continuous-passing style programming paradigm (CPS). In CPS, you give a function that returns it results asynchronously a *continuation* function as input that the asynchronous function calls when it has its result at some later point in time. Writing code in CSP makes working with asynchronous code easier, since a list of asynchronous functions can be chained together, giving the same effect calling a number of normal functions one at the time. This makes it much easier to reason about the flow of data in an asynchronous program.

Using reactive programming for smarter reevaluation of formulas

While researching ways to avoid reevaluating all formulas that also works well in an asynchronous world, I came across the *"reactive programming"* programming paradigm. With reactive programming, we can bind variables with the same semantic effect as cells can be related to each other in a spreadsheet via formulas, i.e. where updates are automatically propagated to bound variables. If e.g. variable `C = A + B`, and either variable `A` or `B` is changed, that change is propagated to variable `C` automatically. This solves our problem since reactive programming implicitly solves our need for dependency tracking and for propagating updates, thus, we have a solution that avoids reevaluating all formulas, instead, only the formulas whose dependencies have changed are reevaluated.

In summary, the uVis.web kernel essentially uses the same the techniques and algorithms as the uVis kernel when creating components and properties in the right order. However, to keep the UI responsive, we switch to an asynchronous programming model and try to limit the time it takes to reevaluate formulas, using reactive programming concepts. This approach aligns well with the recommendations from Kjær.

In the rest of this section, we will discuss the implication of these design decisions in detail. First, we will look at how reactive programming enables us to deal with the challenges related to uVis formulas, and how uVis formulas can be compiled to JavaScript. Then I will describe the algorithm used to create the template tree and the component tree, and how we maintain the component tree when data changes.

3.4.1 From formulas to JavaScript using reactive programming

Since JavaScript is not a reactive programming language, I had to either create my own library that captures the reactive programing paradigm or base my solution on an existing library. Luckily, the Reactive Extensions for JavaScript library (RxJS) provides most of the functionality we need. RxJS is part of the family of open source Reactive Extensions libraries (Rx)[8] available from Microsoft.

Formulas in uVis, i.e. assignments in reactive programming, are represented by *observables* in Rx. Just as a formula result in a value when *evaluated*, an observable *produces* a value when *subscribed* to. Moreover, as formulas can be composed together to create a new formula, so can observables be composed to create a new observable. In short, formulas = observables. Before we see how uVis formulas convert into observables, let us first look at how Rx works.

Aside: The code snippets used in the examples in this and the following sections are written in TypeScript. TypeScript is a superset of JavaScript that compiles to JavaScript, and comes with optional static typing and other constructs that makes it much easier to build large-scale applications. TypeScript is closely aligned with the next version of JavaScript, EcmaScript 6, and should be easy to read for anybody familiar with JavaScript, Java, or C#. In particular, arrow/lambda functions, e.g. `(x, y) => x + y`, should be easily

[8] Rx is available from Microsoft in .NET, JavaScript, Python, Ruby and C++ variants, and from third-parties developers in numerous other languages.

recognizable for C# and Java developers (this arrow function has two input parameters, `x` and `y`, and it returns the sum of the two, i.e. `x + y`).

The Reactive Extensions library

The key type in Rx is the `Observable` type. For our purpose, I think the easiest way to understand what Rx and the `Observable` type is, is to think of it in terms of the *Observer* pattern, the *Iterator* pattern, and *Language-Integrated Query* (LINQ).

I will try to illustrate this using the example code below in Listing 4. It is an RxJS version of the relationship between cells `A`, `B` and `C`, and the formula `C = A + B`, in a spreadsheet.

```
1  // Cell A
2  var A = new Rx.Subject();
3
4  // Cell B
5  var B = new Rx.Subject();
6
7  // Cell C, with formula = A + B
8  var C = A.combineLatest(B, (a, b) => a + b);
9
10 // Somebody subscribers to C, e.g. to use its value in a graph
11 var subscription = C.subscribe(
12     nextValue => {
13         // This is an anonymous "onNext" callback function.
14         // Do something with nextValue from C
15     }, error => {
16         // This is an anonymous "onError" callback function.
17         // Do something about the error.
18     }, () => {
19         // This is an anonymous "onCompleted" callback function.
20         // React to the completed event from the source subject.
21     });
22
23 // Set the value of A to 1
24 A.onNext(1);
25
26 // Set the value of B to 41
27 B.onNext(41);
28
29 // Set the value of A to 215
30 A.onNext(215);
31
32 // End the subscription to values from C
33 subscription.dispose();
```

Listing 4: C = A + B example in Reactive Extensions. The code is written in TypeScript.

Relation to the Observer pattern

In the Observer pattern, we have *subjects* (observables) that *observers* can subscribe to. When subscribed, an observer will be notified by the subject if the subject's state or value is changed (Gamma, et al., 1994). An observable works in a similar fashion. We can use it to represent a component property, whose value might change at some point during the execution of a uVis application. If somebody subscribes to the component property, they will be notified when it changes, by having the new value pushed to them.

In Listing 4, there are three observables declared in lines 2, 5 and 8. Cells `A` and `B` are represented by the `Subject` type. It is a special observable that we can push new values to that it will push to its observers. Cell `C` is a formula, `=A + B`, that we represent by applying the RxJS `combineLatest` query operator to the subjects `A` and `B`. `combineLatest` takes the latest value produced by `A` and `B` and passes them to a *selector* function that computes their sum, `(a, b) => a + b`, that it will push to its observers. Now let us look at how the act of subscribing to an observable relates to the Iterator pattern.

Relation to the Iterator pattern

The Iterator pattern is used to iterate over a collection of elements, via a `Iterator` or `Enumerator` object. You are able to ask the `Iterator` object for the next element using a `next()` method and ask it if there are more elements available using the `isComplete()` method (Gamma, et al., 1994). If an error occurs during iteration, the `Iterator` object will throw an exception you can catch. The relationship to Rx becomes obvious when we look at how we subscribe to an observable. When an observer subscribes to an observable using the `subscribe` method, it can provide three callback functions to the observable: `onNext`, `onError`, and `onCompleted` (line 11 in Listing 4). The observable will invokes the `onNext` callback function each time it has a new value, passing in the new value to the `onNext` callback function. It invokes the `onError` callback function if an error occurred in the observable. The make it possible for an observable to provide error messages to observers, similar to how classic `try-catch` exception handling works. The last callback function, `onCompleted`, is used by the observable to signal to observers that no more new values will arrive, similar to `isComplete()` from the Iterator pattern. The Iterator pattern and Rx diverges on a key point. In Rx, we *push* values to the subscriber. In the Iterator pattern, we *pull* elements. Because of this, an observer is able to dispose of a subscription when it no longer needs it. It does this through the `dispose()` method available to it on a token it received from the observable when it subscribed to it (line 33 in Listing 4). The observable can end a subscription from its end by invoking either the `onError` or the `onCompleted` callback function.

In the example in Listing 4, no value are pushed to the `onNext` function we provided in line 12-14, before the JavaScript runtime moves past line 24 and 27. When `1` is pushed to `A` in line 24, it pushes that onwards to `combineLatest`. Since `combineLatest` only has one of the two values it needs, it does nothing more at this point. When `41` is pushed to `B` in line 27, `combineLatest` receives the `41` value from `B` that it then passes to its selector function along with the value it received from `A`. The selector function returns the value `42` (1+41) that `combineLatest` passes to our `onNext` function that makes the value available to us via the `nextValue` function parameter.

The reason why it works is that `combineLatest` automatically subscribes to both `A` and `B` when we explicitly subscribed to it in line 11, but not before. When `A` is updated again in line 30, `combineLatest` receives the new value 215 which it uses, combined with the value it received previously from `B` (41), to compute the value 256 that it pushes its observer. When we dispose of our subscription to C in line 33, `combineLatest` will also dispose of its subscriptions to `A` and `B`. If either `A` *or* `B` pushes an error message to `combineLatest`, it will forward that error to its observers, and dispose of itself. If both `A` *and* `B` are marked as completed, `combineLatest` will also signal to its observers that it is completed, and no new values will arrives. This is key to prevent memory leaks and allow us to propagate (error-) messages between asynchronous functions nicely.

Relation to the LINQ

The ability to compose formulas together to form a new formula is where the relationship to LINQ comes into play. We have already seen one example of this using the `combineLatest` query operator. With LINQ, you are able to use query operators such as `Select`, `Where`, `Join`, `Aggregate`, `OrderBy`, etc., to create a query over an enumerable collection. Most of these query operators are available in Rx, including new ones that are specific to working with observables. This includes query operators that make it possible to compose observables together to create a new observable, e.g. `combineLatest`. This ability is critical to imitate reactive programming. Using the LINQ-style query operators, we are able to model how data should flow from one formula to another – from one observable to another – as we have seen in the example

above and in later examples to come. There is no limit to how many of query operators we can apply to an observable to model data flow; we can even create an observable that produces other observables.

If for example, we were only interested in odd values coming from `A`, we could use the `where` query operator when defining `C` in line 8, i.e.:

```
8   var C = A.where(a => a % 2 !== 0).combineLatest(B, (a, b) => a + b);
```

Here we filter the values coming from `A` *before* they arrive in `combineLatest`. In fact, we create a new anonymous observable using the `where` query operator that `combineLatest` subscribes to instead of `A`. The where function takes a *predicate* function as input, i.e. a function that takes some input and returns a boolean. The alternative syntax, which does not use an anonymous observable, looks like this:

```
8   var A_Odd = A.where(a => a % 2 !== 0);
9   var C = A_Odd.combineLatest(B, (a, b) => a + b);
```

This is useful if other observers are interested in only the odd numbers from `A`. Then the `A_Odd` observable can be reused.

A note about asynchronicity: Because observables push notifications to observers, it is actually aligning well with the asynchronous programming model we established earlier in section 3.4. In fact, the subscription pattern with callback functions matches the event and event handler pattern exactly, since events also push notifications to event listeners via event handlers. Asynchronous web service requests are similarly push-based – they push the results of a request to the callback function they were given.

I hope that this short introduction to Rx has given you an idea of how it works. This understanding should be further enhanced with the examples coming in the next section, where I will demonstrate more Rx query operators used in uVis.web. For even more details on Rx, I refer to the excellent book “Introduction to Rx” (Campbell, 2012), which is available in an online edition free of charge.

Formulas as observables

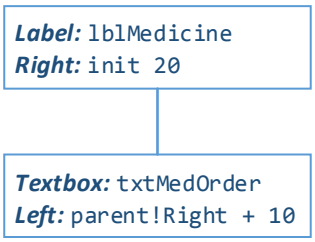
A quick note about who observes (or subscribes to) the observables before we get the examples. In uVis.web, templates explicitly subscribes to the `Rows` property when it creates components, and components explicitly subscribes to a canvas and to its properties when it is about to renders itself. Implicitly, properties (i.e. observables) will lazily subscribe to other observables they depend on as we saw previously.

To keep the code snippets in the examples succinct and easy to follow, observables reference each other directly as if they exist in the same variable scope. In uVis.web, the observables can reference each other in a similar way. However, the reference syntax is more verbose because observables are attached to components in different scopes.

All the examples will have a small template tree with template properties to the left, and Rx code to the right with a set of component properties based on the template properties. I hope that this will illustrate how a compiler can translate uVis DSL into Rx/JavaScript.

Example of basic dependency on another property

In the first example, we have a single dependency between a component property that belongs to a parent component and a component property that belong to its child.




```
// 'rightProp' is a component property created by the 'Right' template
// property of the lblMedicine template.
var rightProp = new Rx.BehaviorSubject(20);
```

```
// 'leftProp' is a component property created by the 'Left' template
// property of the txtMedOrder template.
var leftProp = rightProp.select(right => right + 10);
```

First create the `rightProp` observable using the `BehaviorSubject` class. The `BehaviorSubject` class is a special version of the `Subject` class introduced earlier that takes an initial value, in our case 20, and allow us to update that value at a later point, e.g. in response to user input. When the value is changed, `BehaviourSubject` will push the new value to observers. We use the `BehaviorSubject` whenever the property formula results in a static value, i.e. a value that is not dependent on other properties or data, or when the formula contains the `init` keyword.

A note about the keyword `init` used in `Right` template property. In uVis, a developer can specify that a template property should create a component property whose value it will not override later by using the `init` keyword. If the developer does not use the `init` keyword, the uVis kernel will override the component property's value whenever it reevaluates all formulas. Using the `init` keyword, the developer is able to tell the uVis kernel that it should ignore a component property when reevaluating. In uVis.web, there are no reevaluations as such; however, the component property will not receive updates from any dependencies it might have had used during the initial evaluation.

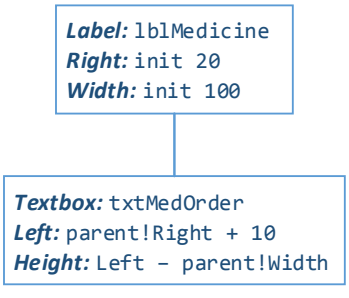
Then we create the `leftProp` observable by using the `select` query operator on the `rightProp` observable. The `select` query operator takes a selector function as input that it uses to modify values coming from the source observable (`right => right + 10`).

Key points:

- Using `select`, or any other query operator on an existing observable, creates a new observable that will handle the subscription to the original observable automatically.
- If nobody subscribes to `leftProp`, it will never subscribe to `rightProp`.
- When somebody subscribes to `leftProp`, it will subscribe to `rightProp`. Any values coming from `rightProp` will be transformed (`right + 10`) and forwarded. Any error messages or completed messages will also be forwarded.

Example of dependency on multiple properties

We must also handle the case where a formula references two (or more) formulas. This example is similar to the C = A + B example we saw earlier.




```
// 'rightProp' and 'widthProp' is a component property created by the
// 'Right' and 'Width' template property of the lblMedicine template.
var rightProp = new Rx.BehaviorSubject(20);
var widthProp = new Rx.BehaviorSubject(100);
```

```
// 'leftProp' is a component property created by the 'Left' template
// property of the txtMedOrder template.
var leftProp = rightProp.select(right => right + 10);

// 'heightProp' is a component property created by the 'Height' template
// property of the txtMedOrder template.
var heightProp = leftProp
    .combineLatest(widthProp, (left, width) => left - width);
```

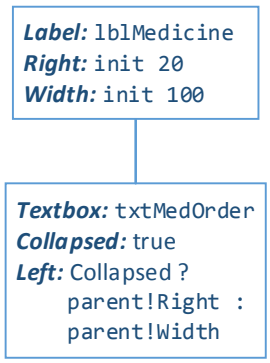
As in the `C = A + B` example, we use `combineLatest` to combine the values from `leftProp` and `widthProp`. `combineLatest` is able to combine two or more observables, i.e. by passing the observables as arguments to it, e.g.: `cp1.combineLatest(cp2, cp3, cpN, (val1, val2, val3, valN) => …);`

Key points:

- `combineLatest` will produce a new value whenever one of the observables it depends on produces a new value.
- We use `combineLatest` when a formula depends on two or more properties.

Example of changing dependencies

Because the uVis DSL support if-then-else (`b ? c : d`), there is a scenario where the dependencies can change at runtime.




```
// 'rightProp' and 'widthProp' is a component property created by the 'Right' and
// 'Width' template property of the lblMedicine template.
var rightProp = new Rx.BehaviorSubject(20);
var widthProp = new Rx.BehaviorSubject(100);

// The collapsed property is a custom property created by the developer to
// indicate the collapsed/expanded state of a visual component.
var collapsedProp = new Rx.BehaviorSubject(true);

// 'leftProp' is a component property created by the 'left' template property of
// the txtMedOrder template.
var leftProp = collapsedProp.select(collapsed => {
    return collapsed ? rightProp : widthProp;
}).switchLatest();
```

In this example, the `Left` property references either its parent's `Right` or `Width` property, depending on the value of its own `Collapsed` property. The `Collpased` property is a custom property created by the developer to control a component visual collapsed/expanded state on screen. Admittedly, defining the value of `Left` like this might not make much sense in the real world, but it serves as a good example in this case.

To make this work, we create our `leftProp` using the `select` query operator on the `collapsedProp`. The selector function we pass to the `select` query operator does not return a value as we have seen previously examples, but instead it returns an observable, either the `rightProp` or the `widthProp` observable to be exact. However, we do not want observers of `leftProp` to receive an observable; we want them to receive the values that either the `rightProp` or the `widthProp` observables produce. Therefore, we apply the `switchLatest` query operator (last code line). The `switchLatest` query operator will take the observable passed to it and subscribe to it, passing on values and notifications to its observers. If `switchLatest` receive a new observable from its source, it will unsubscribe for the previous observable it received, and subscribe to the new one, and use that instead.

Key points:

- `switchLatest` subscribes to observables it receives from its source. When a new observable arrives, it automatically unsubscribes from the previous and subscribes to the new one.
- `switchLatest` forwards data from the observable it has subscribed to, to its observers.
- If all of `switchLatest` observers unsubscribes from it, it will also unsubscribe from the observable it is subscribed to.

Example of data source query

In this example, we see how a `Rows` property is converted into JavaScript. This is the first example that uses a data source, in this case, an OData-based data source. Data sources are shared between all components in an uVis application, and can be used to create an observable, that will, when subscribed to, query the data source and return its result to the observers.

Label: `lblPatient`
Rows: `Patient OrderBy ptName`
Text: `ptName`

```
// A ODataDataSource object, that can be used to
// create an observable query to an OData web service.
var patientDS = new ODataDataSource("http://dws.local");

// 'rowsProp' is a special property that the template
// uses to determine how many components it must create.
var rowsProp = patientDS.query("/Patient?$select=ptName&$orderby=ptName");

// 'index' of the component in its bundle
var index = 2;

// 'rowProp' is a component property created automatically by the template
// the template that created the component, e.g. lblPatient template.
var rowProp = rowsProp.select(data => {
    return Array.isArray(data) ? data[index] : data
});

// 'textProp' is a component property created by the 'Text' template
// property of the lblPatient template.
var textProp = rowProp.select(function (patient) {
    return patient.ptName;
});
```

First, we create the `rowsProp` by calling the `query` function on the `patientDataSource` object. The query function for an `ODataDataSource` takes an OData formatted query string as input. This query string in this example looks at the `Patient` table, selects the `ptName` column, and sorts the result set alphabetically according to the `ptName` column[9]. The uVis DSL compiler sees that the `Text` property uses the `ptName` column. That is how it generates an efficient query, which only returns the necessary columns.

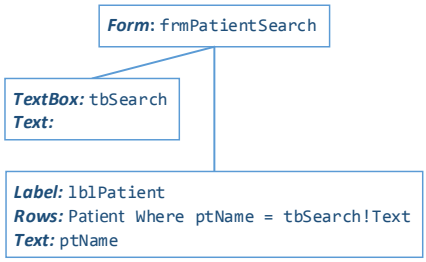
The number of rows in a result set determines how many components a template creates. A template will automatically create the special `row` component property that makes each component's row from the result set available to it. The row is selected using the `index` variable, which is a special read-only variable that exist on each component. It refers to each component's position in the bundle it is in. In this case, our component is located at the third position (zero-indexed) in its bundle.

Last, we have the `Text` property. The `textProp` observable is created by using the `select` query operator on the `rowProp` observable. It returns the value of the `ptName` column.

[9] To learn more about the OData query syntax, see http://odata.org

Example of data source query with dependency

This example builds on the previous, thus some code has been omitted. The important thing to note is that in this example, the `Rows` formula are depending on data from another property, the `Text` property on a `tbSearch` component that it uses to filter the patient names.




```
// 'textProp' is a component property created by the 'Text' template
// property of the tbSearch template.
var textProp = new Rx.Subject("");

// 'rowsProp' is a special property that the template
// uses to determine how many components it must create.
var rowsProp = patientDS.query("/Patient?$select=ptName&$where=ptName
eq @1", textProp);
```

The `textProp` from the `tbSearch` component is a `Subject` observable that allows us to push new values to observers, when the user changes the value of the text box on the screen.

When creating the `rowsProp` for this example, we use another version of the `query` function on the `patientDS`. This version takes as input an OData formatted query string, with a replacement token where the patient name should be, e.g. `@1` (similar to prepared statements in SQL). It also takes one or more observables, in this case the `textProp` observable, that provides the patient name that query replaces with the token in the query string.

When somebody subscribes to `rowsProp`, `rowsProp` will subscribe to `textProp`, and once `rowsProp` receives a value from `textProp`, it will create its query string and send a request to the OData web service. When the result of the request arrives, it passes it on to its observers. If `textProp` pushes a new value to it, it will recreate the query string, and use that to send a new request to the OData web service, again passing on the result to observers.

Example of merging results from two data sources on the client

In section 3.1 we discuss requirement W.2, the option of having uVis.web merge data from two or more web service requests (from the same or different DWS's). This can enable some scenarios that might not otherwise be possible if a web service does not support server-side joins, group by, etc., which uVis normally allows the developer to do.

Simple things such as merging the results of two web service requests can be done using the `combineLatest` query operator we have seen in used a few times. The only hard part is to create the selector function that receives the input from the two requests and combines it somehow into a new object or array. Other observable query operators enable scenarios that are more advanced; among those are the query operators `groupBy`, `join` and `selectMany`.

Example of properties without dependencies

Depending on the uVis application, each component can have numerous properties defined that have a formula with no dependencies, e.g. `Color: red` or `Width: 200`. These properties are just created using the `BehaviorSubject` class, as we have a few times in the examples above.

Putting it all together – a Patient Search Form example

As a final example, let us try to build a simple search form for finding a patient in the patient database exposed via a DWS. We have a textbox for entering a search term and below that, we display the search results. The search form should work similarly to the live-search feature of search engines webpages, where the user starts to enter a search term, and as soon as the user pause for a moment, the results are retrieved. In other words, we do not query the DWS after each keystroke, but allow the user to enter as much as the name as he remembers, and then queries the DWS. Figure 11 shows the expected output in three different cases, depending on the content of the search textbox (and database).

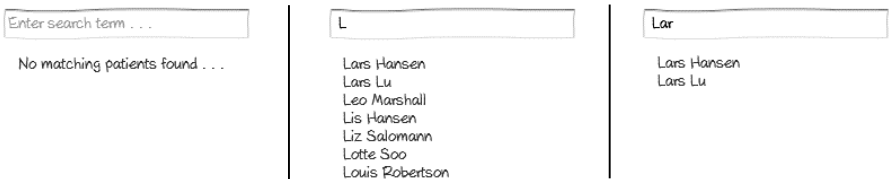


Figure 11: The output of the "Patient Search Form" example. Left, we have the search form without any text typed into the search field. Center, we have the search form with the letter 'L' typed into the search field, and matching names starting with 'L' listed below. Right, we have the search form with the letters 'Lar' typed into the search field, and the patients with a name starting with 'Lar'.

The data map is trivial in this example. It is a single table named `Patient` that has, among others, a column named `ptName`, which we use to filter the patients.

Before we look at the template tree and property formulas, let us first look at the HTML and JavaScript code that a developer would manually write to create the search form without uVis.web.

RxJS + JavaScript + HTML version

Listing 5 has the HTML code needed for this example. For this example, we only have a minimum of HTML, an `INPUT HTML` element that allows the user to enter his search term, and a single `P HTML` element showing the default result message.

```
1  <div id="frmPatientSearch">
2      <input id="tbSearch" type="search" placeholder="Enter search term . . ." />
3      <p class="lblPatient">No matching patients found.</p>
4  </div>
```

Listing 5: HTML code for the "Patient Search Form" example.

Listing 6 has the relevant parts of the JavaScript needed to make the example run. Let us look through the code to see how it works.

In line 2, we get a reference to the `tbSearch` DOM element and in line 3, we create the observable named `textChanged` using the `fromEvent` function, passing it the reference to the `tbSearch` DOM element and the event we want it to subscribe to on the `tbSearch` DOM element, i.e. the input event. The `textChanged` observable will send an input `Event` object to observers when new input is added to the textbox. This input `Event` is used in line 7 to extract the much more interesting text value of the `tbSearch` DOM element.

In line 11, we change the behavior of the `textChanged` observable by applying the `throttle` query operator to it. The `throttle` query operator ensures that new values are only pushed to observers if there have been no changes to the textbox for a certain period, in our case 250 milliseconds. That way, if the user is keying in new values fast, we stop pushing changes to observers, until the user pauses for at least 250 milliseconds.

```
1  // First we create the basic observable from the 'input' DOM event on tbSearch
2  var tbSearch = document.getElementById("tbSearch");
3  var textChanged = Rx.Observable.fromEvent(tbSearch, "input");
4
5  // Then we extract the value from tbSearch and remove any whitespace around it.
6  textChanged = textChanged.select(event => event.target.value.trim());
7
8  // Then we extend the textChanged event with throttle to limit number of events passed to observers.
9  textChanged = textChanged.throttle(250);
10
11 // Then we make sure to only return distinct new values to observers.
12 textChanged = textChanged.distinctUntilChanged();
13
14 // Our patient data source
15 var patientDS = new ODataDataSource("http://dws.local");
16
17 // Now we can create a searchResult observable, that uses the textChanged event
18 // and queries a data source when every there is new content from textChanged
19 var searchResult = patientDS
20      .query("/Patient?$select=ptName&$filter=startswith(ptName,'@1')&$orderby=ptName", textChanged);
21
22 // Then we create a function that will update search results on screen
23 function updateSearchResults(patientNames /* array of strings */) {
24      var i, elm;
25     // find existing results
26     var currentResults = document.getElementsByClassName("lblPatient");
27     // and remove them from the DOM
28     for (i = 0; i < currentResults.length; i++) {
29         currentResults[i].remove();
30     }
31     // add new search results
32     for (i = 0; i < patientNames.length; i++) {
33         elm = document.createElement("p");
34         elm.innerText = patientNames[i];
35         elm.setAttribute("class", "lblPatient");
36         document.appendChild(elm);
37     }
38 }
39
40 // Finally we subscribe to the searchResult observable using the function created above.
41 searchResult.subscribe(updateSearchResults);
```

Listing 6: The JavaScript (TypeScript) that a developer would manually write when building the search form example without uVis.web.

In line 15, we change the `textChanged` observable's behavior again, this time with the `distinctUntilChanged` query operator. The `distinctUntilChanged` query operator ensures that only new values are sent to the observers. If e.g. the user enters "Lar" into the textbox and pauses, causing "Lar" to be send to observers, then hits backspace deleting "r" and immediately adds the "r" back followed by another pause, "Lar" would be resend to observers. `distinctUntilChanged` sees that the new value is the same as the previous, and ignores it. This way we avoid unnecessary web service requests.

At this point, our `textChanged` observable is finished. We have taken the initial output from the `tbSearch` `input` event and transformed it three times using `select`, `throttle`, and `distinctUntilChanged`. This is a good example of how we can model the data flow easily in an application with Rx.

The rest of the code is a variant of what we have seen before, expect for the `updateSearchResults` that acts as our `onNext` callback function. It will remove an existing search result and add a new one, when it receives an array of patient names from the `searchResult` observable.

Now we can look at how the corresponding uVis.web version looks.

Template tree and formulas for the uVis.web version

The template tree for the uVis.web version is in Figure 12. Let us go through the templates and their properties to see how they correspond to the HTML and JavaScript code above.

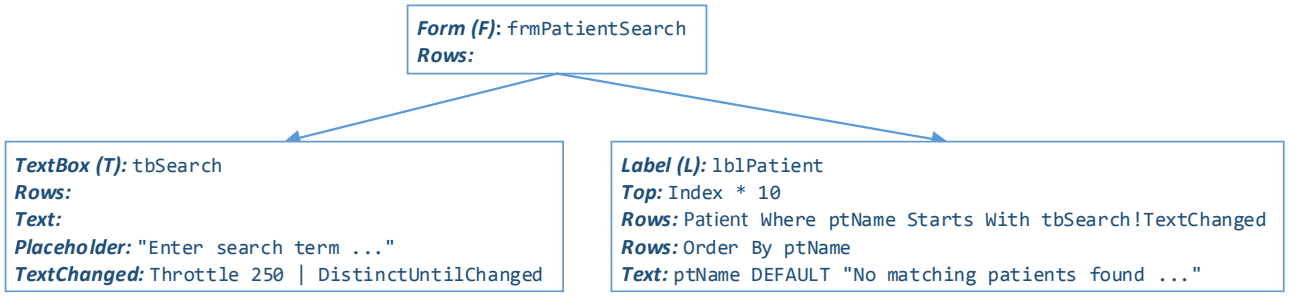


Figure 12: Template tree for the "Patient Search Form" example.

At the root of the tree, we have the `frmPatientSearch`. This corresponds to the outer `DIV` HTML element in Listing 5. The `tbSearch` and `lblPatient` templates correspond to the `INPUT HTML` and `P HTML` element in Listing 5 respectively.

Looking at the `tbSearch` template, we see the `Placeholder` property. It enables the developer to specify a helpful placeholder text that provides the user with a hint for the usage of the textbox. As soon as the user selects the textbox, the placeholder text is removed, and it is only displayed when there is no regular text in the textbox. See Figure 11 (left). It corresponds to the placeholder attribute on the `INPUT HTML` element in Listing 5.

`tbSearch` also has the event property `TextChanged`. Without any modifications, it behaves just as the `textChanged` observable we have on line 6 in Listing 6. In other words, an observable that returns the latest value from `tbSearch` immediately when an "input" event is triggered. However, the developer has added both the `Throttle` and `DistinctUntilChanged` functions, using chain operator, i.e. `|` character. The chain operator can be used to chain functions together, as long as they are compatible, as observables are[10]. With the two functions added, the `TextChanged` property now corresponds to the final `textChanged` observable on line 12 in in Listing 6.

The `Rows` and `Text` properties from the `lblPatient` template both make up the `searchResult` observable on line 22 in Listing 6, and are converted as the `Rows` and `Text` properties in the previous example.

A note about the `TextChanged` property: In reality, creating a live search functionality like this might be too hard for the target audience, especially concerning the use of `Throttle` and `DistinctUntilChanged` to limiting web services queries. A better solution is to have specialized version of the textbox component, e.g. a live search textbox that comes preconfigured with this functionality, and allows the developer to reference to the `Text` property instead of the `TextChanged`.

I hope what we have covered so far in this section has left you with an understanding of how Rx works and how it can be used to represent uVis formulas at runtime. I also hope it is clear how it is possible to solve the challenges with dependencies between formulas, outlined at the start of section, using the techniques discussed above.

[10] The chain operator is also not part of the standard uVis DSL. It is a suggested addition to better support scenarios such as this. Another identifier might be more suitable for the target audience; no usability studies have been done on this.

Now I will continue the discussion of the uVis.web kernel implementation with a look at major types in the kernel.

3.4.2 Types in the uVis.web kernel

In this section, I describe the major types in the uVis.web kernel, and some of their important characteristics. That should make it easier to understand how a template tree and component tree is created and how a specific component can be found in the component tree, including its properties.

The UML class diagram in Figure 13 below includes all the major types of the uVis.web kernel and their relationships. I have tried my best to keep it as simple as possible, including only what is relevant and necessary to understand of how kernel works.

App

The `App` type is equivalent to the content of the `app.json` file sent to clients from an AWS. It holds the basic information about the uVis application the user is starting. It has a list of all forms in the application and a list of shared data sources available to the templates. It also has a collection of application level properties and event handlers. The properties are used to configure the browser window the uVis application is running in, e.g. to change its appearance, and the event handlers are used to subscribe to global browser events such `window.onload`, `windows.unload`, etc.

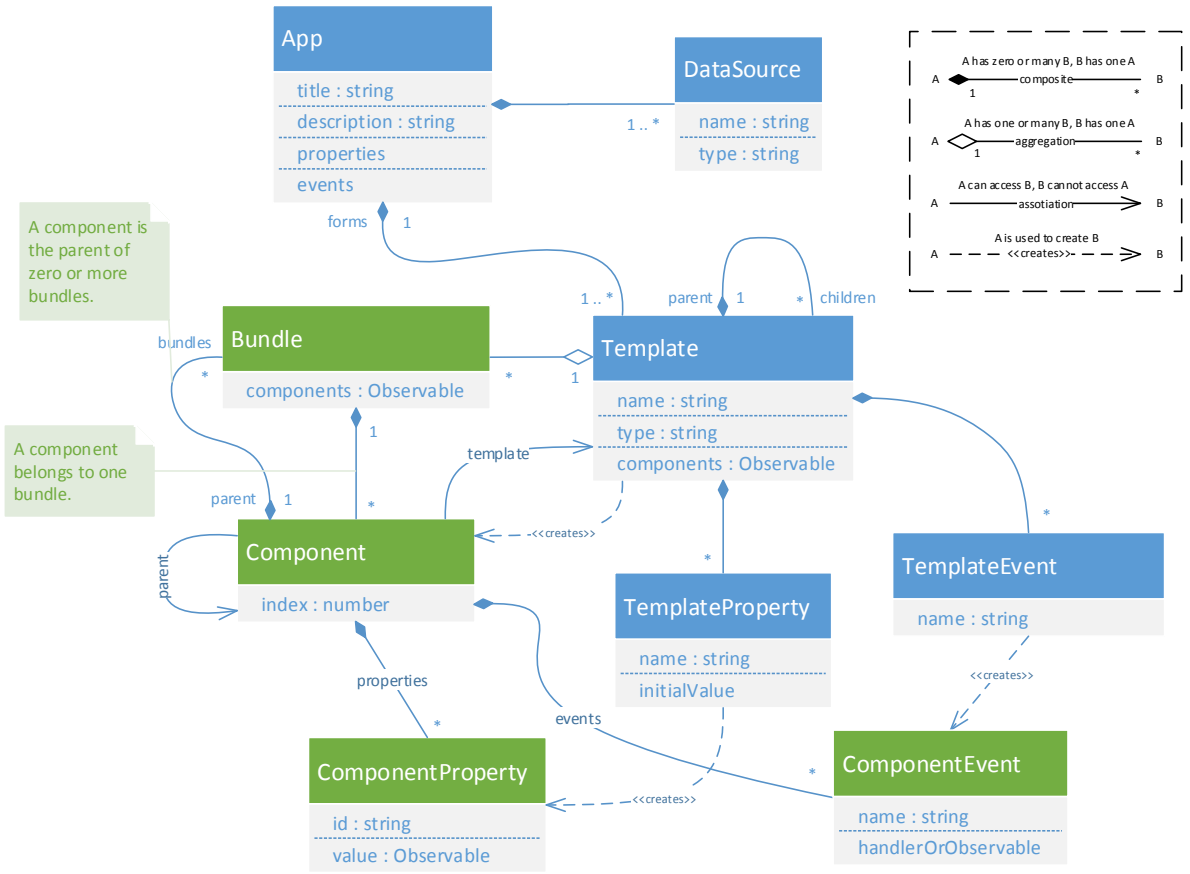


Figure 13: Major types in the uVis.web kernel (simplified UML class diagram). Types in blue are used in the template tree; types in green are used in the component tree. Specializations (sub-classes) are excluded for brevity.

DataSource

As we saw in some of the examples in the previous section, a data source are used to create observables that will query the data source, e.g. a web service, when subscribed to. A data source is selected via its unique `name`.

Template

A template tree consists of instances of the `Template` type. If a `Template` is at the root of a template tree we call it a *form*, however, in practice, a form is just a `Template` without a parent. All forms belong to an `App`, giving the `App` a reference to the root of each template tree in the application. All templates within a template tree must have a unique `name`, and their `type` property indicates the kind of component they will create at runtime.

Not surprising, templates is what creates the component tree (indicated by the dependencies arrows) by creating instances of the `Component` type. It groups the components it creates into bundles, and it provides *component properties* and *component events* to each component it creates. A template exposes a `Components` observable that it uses to push new components it creates to children. This is discussed in details in the coming section about the template tree.

TemplateProperty

The `TemplateProperty` type can create component properties as observables. It has a compiled version of a uVis formula that it uses to create a customized component property observable, tailored to the component it is creating it for. The `name` is the unique name of the template property. Each template property can only exist once per template.

Currently, there are three template component types.

- `TemplateProperty:` It creates a subject-based observable that can have its value changed directly at runtime. If the developer has specified it, it also has a default/initial value assigned to it.
- `ComputedTemplateProperty`: It creates an observable that depends on other properties or data from a data source, and the dependencies are based on the component it creates the observable for.
- `SharedComputedTemplateProperty`: It creates an observable that depends on other properties or data from a data source, where the dependencies do not change based on the component it is assigned to. The `SharedComputedTemplateProperty` will just create one observable that it will return to all components.

TemplateEvent

The `TemplateEvent` type can create either *event handlers* or observable *event properties*. The difference between the two is that an event handler is just a piece of code that executes, e.g. modifying the state of the component tree, as opposed to event properties, who are observables that will push data to observers when the event is triggered. We saw an example of an event observable in the Patient Search Form example. An event is created for each component.

There can be as many event handlers per event type as the developer needs, since they are not observables. However, there can only be one observable event property per event, since properties or data sources can subscribe to it. If there were more, the uVis.web kernel would not know which event property it should return to a would-be observer.

Bundle

The component tree consists of instances of the `Bundle` type. The `Bundle` type has a collection of components, a reference to the template that created the components it holds, and a reference to parent component. A bundle also has a `components` observable that it uses to push new components added to it, to interested parties. We see why this is needed in the coming sections.

Component

A component is an instance of a template. It has an `index` that matches its position in the bundle it is in, and it has a reference to its parent component and the template that created it. It also has a collection of component properties and component events. Each component will render a single visual component onto the canvas it has been designated, in the case of uVis.web, a DOM element, and it can read and write properties on the visual component and attach events to the visual component. It also has a collection of bundles it is the parent of and a reference to the bundle it is in itself.

To understand the relationship between the `Bundle`, `Template` and `Component` types better, it can help looking back on the component tree in Figure 4 on page 9. Here we see that each component is the parent of a bundle one level deeper in the component tree, and we see that a component belongs to one bundle. We also see that a template creates one or more bundles, depending on how many components its parent template created. These relationships are what we have modelled with the `Bundle`, `Template` and `Component` types.

ComponentProperty

The `ComponentProperty` type has a `name` that identifies it and an observable that will produce a value.

ComponentEvent

A `ComponentEvent` has `name` and either a JavaScript code snippet that should run when the event is triggered, or an observable that will push data to observers when the event is triggered (the data being pushed depends on the even type).

Now that we have overview of the major types in the uVis.web kernel, we can dive a little deeper and look at how we create the template tree.

3.4.3 Creating the template tree

Creating a template tree is a straightforward task that we can split up in two distinct subtasks:

1. Compile uVis formulas to JavaScript and extract template tree parent-child relationship for each form in the application.
2. Create an instance of the `App` type, set up data sources, and create a template tree for each form.

Step 1 is happens at the same time we compile uVis formulas to JavaScript. Since I have not implemented a uVis DSL to JavaScript compiler, I will skip step 1 and go straight to step 2.

Step 2 starts by downloading the `app.json` file for the selected uVis application. After the uVis.web kernel has downloaded the `app.json` file, it uses it to create an instance of the `App` type. It is a straightforward process of reading the application definition from the `app.json` file and creating the needed resources, one at the time.

When creating the data sources, it looks at the `type` identifier and downloads the corresponding JavaScript source code file. If, for example, a data source's type is `OData`, the kernel will look for a `ODataDataSource.js` file on the AWS, download it and its dependencies, and use it to create an OData

data source instance. For this purpose, uVis.web uses the *require.js* JavaScript library available from http://requirejs.org.

Each form is created in a similar way. The kernel looks at the form references in the `app.json` file and downloads the .VIS file for the forms that are marked as visible. Then it instantiates the template tree based on the downloaded .VIS file. This too is straightforward. It goes through each form/template definition in the .VIS file and creates the corresponding templates, template properties, and template events. A template also has a `type` identifier that identifies the type of the component a template creates. The source code for each component type is downloaded just as it is for data sources, on demand.

Extending uVis.web with new functionality

This dynamic loading of components and data sources makes it very easy to extend the functionality of uVis.web with support for new types data sources and components, without having to change a single line of code in the core kernel types. It is this easy because JavaScript is a dynamic scripting language. When new components or data sources become available, the AWS administrator just have to upload their JavaScript source code files to the AWS, and the local developers will be able to use them right away by referencing them in their application definitions.

Now that have a template tree, let us create the component tree.

3.4.4 Creating the component tree

As we know from section 2.1, a template cannot create components before it knows how many it should create. This is determined by two parameters:

1. What does the templates `Rows` observable return, e.g. a single object, an array of objects, or a number.
2. How many components does the parent template create.

A template need both because it must create N components (determined by the `Rows` observable) for each component the parent template creates. The special case is when the template is the form, i.e. the root of the template tree, in which case it just creates one set of N components since it has no parent.

To get the data we need from the `Rows` observable we must subscribe to it, and it will keep providing us with new data until it completes (or an error occurs). Getting components from the parent template is more complex, since the parent must first create the components. That means it must first subscribe to its `Rows` observable and get the components from its parent, and so on, all the way to the root of the template tree.

Things become even more complex because the `Rows` observable can continue to produce new data that we must react to, i.e. be able to add or remove components on demand from the component tree. This means that a template must somehow notify its child templates when it create or remove components from the component tree.

My solution is to have each template expose a custom `Components` observable that it uses to push components it creates to observers. Templates can then subscribe to the `Components` observable of their parent and that way be notified when their parent creates a component. However, signaling to observers that a component has been removed from the component tree is not done through the `Components` observable. Instead, we utilize the fact that a component knows who its children are in the component tree, so if a component is removed from the component tree, it will removes its children as well.

Let us look how the `Components` observable is created, and how it works, and then go through some examples that will show how the component tree is created and maintained.

Creating the Components observable

When the template tree has been constructed, the kernel will call an `initialize` method on each template in the template tree, starting from the root and working its way down.

It is in the `initialize` method that the `Components` observable is created. Listing 8 on page 44 has the code that creates the `Components` observable[11]. Let us look more closely at it, but first a quick introduction to creating custom observables.

Since the `Components` observable will be subscribing to other observables (`Rows` and another `Components` observable), we use the `Rx.Observable.createWithDisposable` function (line 1) to create our observable. This function requires us to give it a *create* function that takes an observer as input, and returns an object with a `dispose` method on it, that observer can use to signal it no longer needs our service and we can dispose of any resources we might be using, e.g. the observables we have subscribed to. The object we return in this case is an instance of the `Rx.CompositeDisposable` type (line 2). It allows us to add other "disposables" to it that it will dispose of, when its `dispose` method is called. The boilerplate code for creating an observable using this approach is shown in Listing 7 to the right. Whenever somebody subscribes to the observable we create, our create function is called with the observer that subscribed to it, and we can communicate with the observer as illustrated in Listing 7 through the `onNext`, `onCompleted` and `onError` callbacks the observer has. This is how we send new components to observers.

```
Rx.Observable.createWithDisposable(observer => {
    var disposables = new Rx.CompositeDisposable();

    // Send data to observer.
    observer.onNext(some data ...);

    // Tell observer that no more data is coming.
    observer.onCompleted();

    // Tell the observer that an error has occurred
    // and we cannot continue.
    observer.onError(error message ...);

    return disposables;
});
```

Listing 7: Boilerplate code for creating a custom observable.

Back to Listing 8 and the `Components` observable. Here are the highlights (from the top):

Line 8-16: This internal function is used to decide when to signal to the observer that we will not be producing more components (line 11). This happens when both the template's parent `Components` observable has signaled it is completed and when the `Rows` observable is completed.

We also tell all our bundles that the template is completed (line 14). This is because the collection of components each bundle has is also an observable, and they must also notify their observers that they too are completed.

Line 19-38: This internal function is used to add or remove components from a bundle. If there are too many components in a bundle, we remove the redundant components from the end of the bundle. If there are too few, we create and add the missing components to the end of the bundle, and push it to the observer (line 35).

Line 41-59: This code figures out if the template has a parent, and if it does, it subscribes to the parent's `Components` observable (line 42). If there is no parent, i.e. because the template is form, it creates a single bundle that it will store its components in (line 57-58).

[11] is also attached as a loose paper to the printed version of this report.

```
this._components = Rx.Observable.createWithDisposable(observer => {
    var disposables = new Rx.CompositeDisposable();
    var latestRowCount = 0;
    var parentCompleted = false;
    var rowCountCompleted = false;

    // Internal function that keeps track of the state of the subscriptions.
    var setCompletedState = () => {
        if (rowCountCompleted && (parentCompleted || this.parent.state === TemplateState.COMPLETED)) {
            this._state = TemplateState.COMPLETED;
            observer.onCompleted();

            // Mark all bundles as complete as well
            this.bundles.forEach(b => b.markCompleted());
        }
    };

    // Internal function that will update the number of components in a bundle.
    var updateComponentCountInBundle = (count: number, bundle: ub.uvis.Bundle) => {
        var orgCount = bundle.count;

        // Set state to ACTIVE, indicates this template has produced at least one component.
        if (count > 0) this._state = TemplateState.ACTIVE;

        // If count is lower than current number (orgCount), we remove the extraneous components and dispose of them.
        if (orgCount > count) {
            while (bundle.count > count) { bundle.remove(); }
        } else if (orgCount < count) {
            // Otherwise we add additional components to the bundle
            for (var index = orgCount; index < count; index++) {
                var component = Template.componentFactory(this.type, this, bundle, index, bundle.parent);
                bundle.add(component);

                // Send the newly created component to observer
                observer.onNext(component);
            }
        }
    }

    // Subscribe to parent's components observable, if there is a parent. Creates bundles based on parent's components.
    if (this.parent !== undefined) {
        disposables.add(this.parent.components.subscribe(component => {
            // Create a bundle for this component, if it does not already have one.
            var bundle = component.bundles.get(this.name);
            if (bundle === undefined) bundle = component.createBundle(this);
            // Add components to the bundle
            updateComponentCountInBundle(latestRowCount, bundle);

        }, observer.onError.bind(observer), () => {
            // Mark the parent template observable as completed.
            parentCompleted = true;
            nextTick(setCompletedState);
        }));
    } else {
        // If there are no parent, create a default bundle to hold this templates components. This template
        // is a 'form' since it is at the top of the template tree.
        this.bundles[0] = new ub.uvis.Bundle(this);
        parentCompleted = true;
    }

    // Subscribe to rowCount observable.
    disposables.add(this.rowCount.subscribe(count => {
        // Update the number of components in each bundle
        latestRowCount = count;
        this.bundles.forEach(bundle => { updateComponentCountInBundle(latestRowCount, bundle); });
    }, observer.onError.bind(observer), () => {
        // Mark the rowCount observable as completed.
        rowCountCompleted = true;
        nextTick(setCompletedState);
    }));

    return disposables;
}).publish();

// Then we use the connect method start creating components.
this._componentsConnection = this._components.connect();
```

Listing 8: The custom `Components` *observable from the Template.ts file (available at https://github.com/egil/uVis.web).*

The `onNext` callback function that is provided to the templates parent's `Components` observable (defined in line 44-47) will create a new bundle whenever a the parent template produces a new component, and then it will call the `updateComponentCountInBundle` function (from line 19-38) to add components to the bundle. There is a special case where the component coming from the parent has already created a bundle for the template's components. This happens when a `Rows` observable depends on a component that has not been created yet. We will discuss this scenario in details in an upcoming section.

Line 62-70: This code subscribes to a transformed version of templates `Rows` observable, called `rowCount`, that looks at the data produced by `Rows` and returns a number based on it, indicating how many components the template should create. The code for the `rowCount` observable looks like this:

```
this._rowCount = this._rows.select(result => {
    // If it is an array, produce as many components as there are elements in the array.
    if (Array.isArray(result)) return result.length;
    // If it is a number, produce as many components as the amount the number indicates.
    else if (typeof result === 'number') return result;
    // Anything else, we produce a single component.
    else return 1;
});
```

When `rowCount` provide a new number to the `onNext` callback function, we update the `latestRowCount` variable and call the `updateComponentCountInBundle` function for each bundle we have.

The neat thing about this setup is that does not matter if `rowCount` or the parent template's `Components` observable pushes data first. If it is `rowCount`, it will have no bundles to update, so nothing will happen. If it is the parent `Components` observable, we just create zero components until we have a row count.

Line 73: There is one extra functionality we need from our `Components` observable. Even if there is more than one observer, which will happen when the template has more than one child template, we only want `Components` observable to subscribed to once. Otherwise, it would end up producing a unique set of components for each observer that subscribes to it. To prevent this from happening, we use the Rx `publish` query operator. `publish` converts our observable into a special *connected observable*, that multiple observers can subscribe to, but the underlying observable, our `Components` observable, is only subscribed to once by `publish`.

Line 76: Here we tell our connected observable to connect to the underlying observable, using the `connect` method. This will start the component creation process, even if there are no observers. This turns out to be important, since there are templates that will never have observers – the templates at the bottom of the template tree. Without this feature, these templates would never start producing components.

By using the `publish`/`connect` combination like we do, we risk that an observer (template) will miss some of the created components, because they are created before the observer subscribes. To mitigate this issue, we use the `startWith` query operator on the `Components` observable before we return it to another template. `startWith` allow us to prepend data to an observable stream, in our case, the components that have already been created. The following code snippet is the `components` property from the `Template` type (the `apply` method is a special JavaScript method that exist on functions, that allow us to call a function, i.e. `startWith`, with arguments provided as an array). Note that the components property will initialize the template if it is not already.

```
get components(): Rx.IObservable<uc.uvis.Component> {
    // If the template has not been initialized, we do so now, so it starts producing components.
    if (this.state === TemplateState.INACTIVE) {
        this.initialize();
    }
    return this._components.startWith.apply(this._components, this.existingComponents);
}
```

Now that we know how templates produce components, let us look at some examples.

Creating and updating a component tree (no cross dependencies scenario)

In this example we will look at how a component tree is created when there are no cross dependencies between templates, i.e. when a templates `Rows` observable does not depend on a property on another template's component.

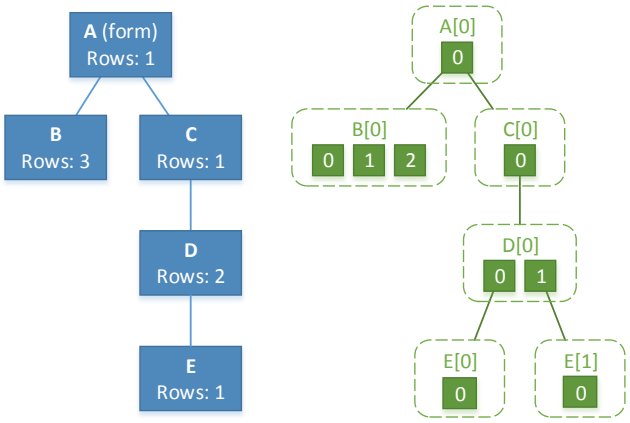


Figure 14: Simple template data tree on the left and the corresponding component tree on the right.

To the right in Figure 14 we have a template and a component tree. Each template in the template tree has a name and rows property, that is just a number, indicating how many components we want the template to create. In the component tree, we have bundles, named to match the template that created it, with a number indicating bundle number. E.g. bundle E[1] is the second bundle template E has created. Inside each bundle, we have the components that a template has created for the bundle, with a number indicating their index/position in the bundle. If a component is a parent to a bundle, a green line goes from the component to the bundles edge.

The first template the kernel initializes is the form template (A). Since A has no parent, the template creates a single bundle, A[0] (line 57), and adds one component to it, because `rowCount` returns 1 (line 65). Then B is initialized. It subscribes to A (line 39) and receives one component from A. It uses that to create the B[0] bundle that it adds three components to, since its `rowCount` returns 3. Template C and D follows and behave the same. Template E receives two components from D, making it to create two bundles, one for each component, E[0] and E[1], that it adds one component to.

Adding additional components

Now that the component tree has been created, let us try to modify it by changing the `Rows` property of template C to 2. This allows is to show what happens when a new row count is available, and what happens when new components arrive. Figure 15 to the right shows how the component tree is changed, with new components in light green.

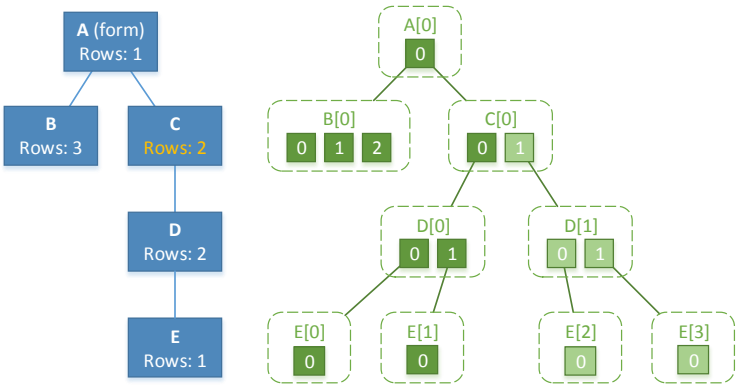


Figure 15: New component tree with branch added to the C[0] bundle.

When a new row count arrives in 62 in template C, it calls the `updateComponentCountInBundle` function with its only bundle, C[0], and the new count, 2. `updateComponentCountInBundle` sees that the new count is

higher than the old are, so it creates an extra component and adds it to the bundle. It also pushes the new component to template D, who is observing template C's `Components` observable. When D receives a new component (line 39), it creates a new bundle for the component, D[1], and creates two components for the bundle, since its row count is still 2. Template E receives the two new components from D, that it creates two bundles for, one for each (E[2] and E[3]), and it adds a single component to each new bundle.

Removing components

Let us see what happens if we turn row count on C back down to 1. As illustrated in Figure 16 to the right, component 2 (C[0][1]) is removed from the C[0] bundle, and the entire branch of the component tree below component 2 is removed as well.

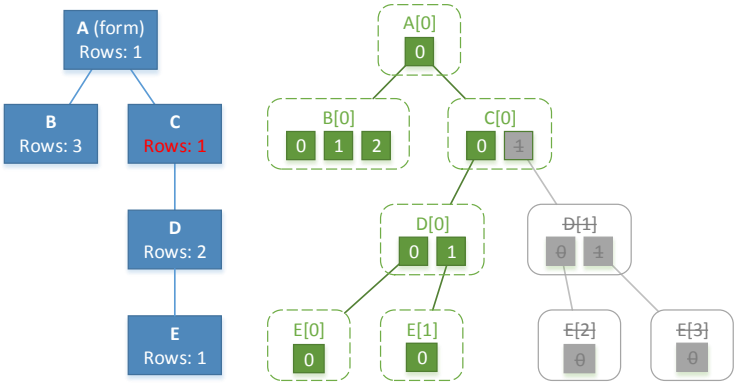


Figure 16: New component tree with branch removed from the C[0] bundle.

When C receives the new row count of 1, the `updateComponentCountInBundle` function sees that the row count is less that it was previously. It then calls the `remove` function on the bundle (line 23), that causes the bundle to remove the *last* component in the bundle, and call the components `dispose` method.

When a component is disposed, it will remove any bundles it is the parent of from the component tree. In this case, C[0][1] is the parent of the D[1] bundle that it removes from template D's bundle collection and then calls its `dispose` method. This causes the D[1] bundle to remove all components it holds, i.e. D[1][0] and D[1][1], and call their dispose method. This makes the components D[1][0] and D[1][1] remove the bundles E[2] and E[3] from template E and dispose of them. The E[2] and E[3] bundles in turn disposes of their component E[2][0] and E[2][1], which ends the remove operation.

It is critical that a component (C[0][1]) remove the bundle (D[1]) from the template (D) that created it. Otherwise, templates would keep maintaining disposed bundles.

Creating a component tree with cross template dependencies

What if template B depends on template D, i.e. its `Rows` observable depends on a property of a component that template D will create, e.g. the `Text` property. Figure 17 below illustrates this relationship.

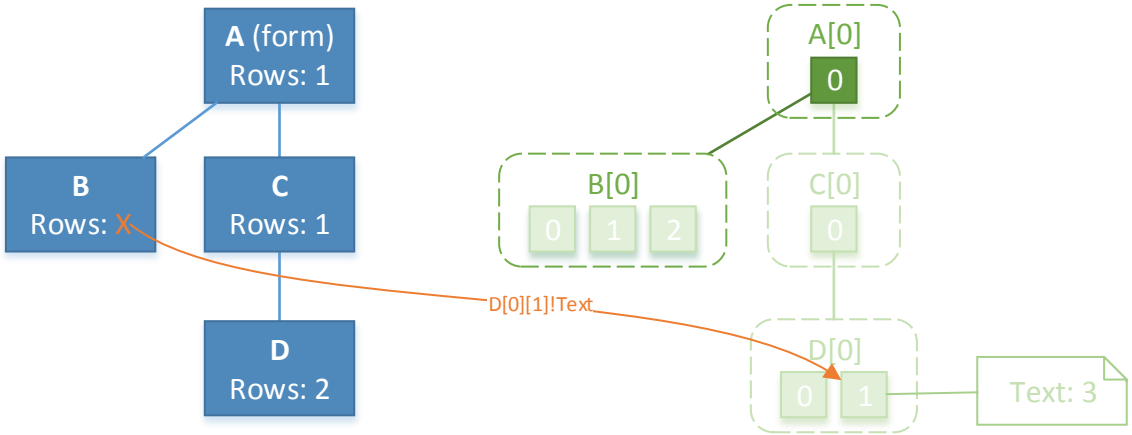


Figure 17: A template tree where template B depends on a property on a component in bundle D[0]. The bundles and components in light green have not yer been created.

When B is initialized, and it subscribes to its rows count observable, the `Rows` observable will, metaphorically speaking, walk the component tree until it gets to component D[0][1], that it will ask for the observable to its `Text` property. In this case, neither D[0] or its parent C[0] have not been created yet, and neither has the components in those bundles, so that must happen first. This is where the algorithm for walking the component tree comes into play.

Walking the component tree

A template has a method called `walk` that takes a number as input. The number indicates which of the components in the root bundle (A[0]) of the component tree it wants to start its component tree walk from. In our examples, template A has only created one component, but a form template can create as many components as its row count specifies, so there can be more.

Once a template has a component from the root bundle, it can use that as its starting point to traverse down the component tree, until it finds the component it seeks. Components can also walk the component tree and they naturally have more starting options, i.e. they do not have start at the top, they can go up and down from their position, or to the sides, i.e. to access other components in its bundle.

The key to make this work is the `Components` observable exposed by the `Bundle` type. That makes it possible to request a component from a bundle, even before the component has been created, just as a template can subscribe to components from a parent template.

To make it a little easier create an observable that will walk down the component tree an find a property, we have two helper functions named `get` and `property`, both custom Rx query operators created for uVis.web, that hides some of the boilerplate code needed in every step. So instead of having to write…

```
componentObservable.select(component => {
    return component.bundles.get('c').components
        .where(childComp => childComp.index === 42);
});
```

… for each level we need to traverse in the component tree, we can just write `componentObservable.get('c', 42)`. Because `get` is a proper Rx query operator, more calls to `get` can be chained together, e.g. `componentObservable.get('a', 0).get('c', 3).get('d', 2)`. The `property` query operator works the same; it just selects a property on a component, e.g.:

```
get('d', 2).property('top').
```

Let us continue our example in Figure 17 using `get` and `property` to walk down to D[0][1]!Text.

Walking the component tree to D[0][1]

We start with the component A[0][0] and call its `get` method like so:

```
C00 = A[0][0].get('C', 0);
```

This tells the A[0][0] component that we want the bundle it is the parent of that is created by the template C, and we want the first component in that bundle. If the bundle does not exist yet, it is created, and template C is initialized. This will cause template C to create the bundle C[0] and fill it with components. As soon as the component with index = 0 is added to the bundle, it is returned to us and we can continue

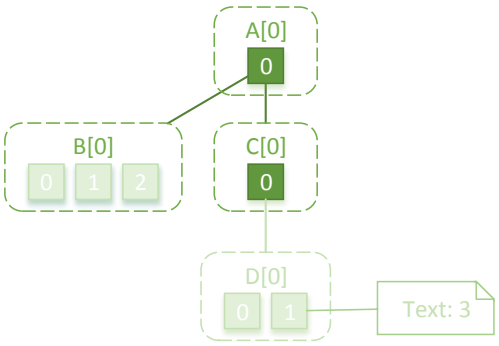


Figure 18: Component tree after template C has been initialized and bundle C[0] has been created.

to the next step. Figure 18 shows the updated component tree after bundle and component have been created.

The next step is to continue to expand the `C00` observable we received from `A[0][0].get` in the previous step:

```
D01 = C00.get('D', 1);
```

Here we tell the component C[0][0] that we want the bundle it is the parent of that is created by the template D, and that we want component at index 1 in the bundle. As in the previous step, template D is initialized and the bundle and components for the bundle is created. Now we can request the `Text` property from D[0][1]:

```
D01_text = D01.property('text')
```

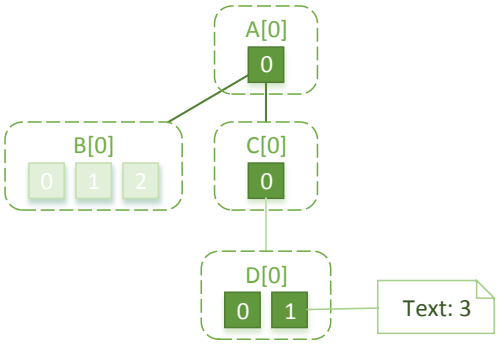


Figure 19: Component tree after template D has been initialized and bundle D[0] has been created.

`property` will check if the component property already exists, if not, it will ask the component to create the property. The component does this by contacting template D and retrieving the `Text` template property that it uses to create a `Text` component property. In general, all component properties are created lazily when they are first requested.

At this point, `D01_text` is an observable that will return the value of the `Text` property of component D[0][1], and all the components and bundles above it in the component tree has been created. Put together in one line, the full query against the component tree looks like this:

```
D01_text = A[0][0].get('C', 0).get('D', 1).property('text');
```

If we start in the template B, it looks like this:

```
D01_text = B.walk(0).get('C', 0).get('D', 1).property('text');
```

Because calling `B.walk(0)` returns component A[0][0].

An important take away here is that by chaining `get` calls together, we create an observable that will walk the component tree asynchronously, one level/step at time, getting the component or property it needs to proceed. It does not care when it receives the next component or property that it needs to take another step, since that can depend on many factors, such as network delay, user input, etc., as long as there is no cyclic dependencies. We talk about cyclic dependencies shortly.

Establishing dependencies between properties

As we know, component properties can dependent on properties owned by other components just as the special `Rows` property can. These dependencies are established somebody subscribes to the component property, just as we saw with the `Rows` property above.

To establish the dependencies between properties, we use the same technique discussed above to navigate the component tree until we find the property we need, then we can use Rx query operators to transform the property observables as demonstrated in section 3.4.1 and return an output that the visual components can use for rendering.

With both the rows property and component properties in general, there is a possibility for the developer to write a cyclic dependency. Let us look at have uVis.web handles these.

3.4.5 Detecting cyclic dependencies

It is almost impossible to create a cyclic dependency with Rx if you are writing the code manually, that is probably the reason why there is no built-in protection against such in the Rx library. A cyclic dependency between observables can result in two things. A stack overflow, because the observables will jump back and forth between each other when they try to observe each other (subscribe to each other), until the stack runs out of space. The alternative is that nothing happens. This is the case when both observables are able to indirectly subscribe to each other, e.g. because the observables themselves are created asynchronously, and both observables are simply waiting for the observable they have requested to show up. Neither is a good scenario for uVis, because the local developer can in fact inadvertently create a cyclic dependency without trying very hard, especially if the dependency is chosen based on data from a data source.

In uVis, there are three places where a cyclic dependency is possible.

1. Between template's `Rows` properties, used for creating and maintaining the component tree
2. Between component properties – property A depends on property B who depends on property A
3. Between canvas's – canvas A rendered on canvas B who is rendered on canvas A

The detection algorithm works by dynamically building *dependency graphs* where templates, component properties, or canvases are the *nodes*, and the *edges* are their dependencies. The three uVis query operators `walk`, `get`, and `property` maintain the dependency graphs. They do it by updating a list of dependencies stored with each node as they walking the component tree.

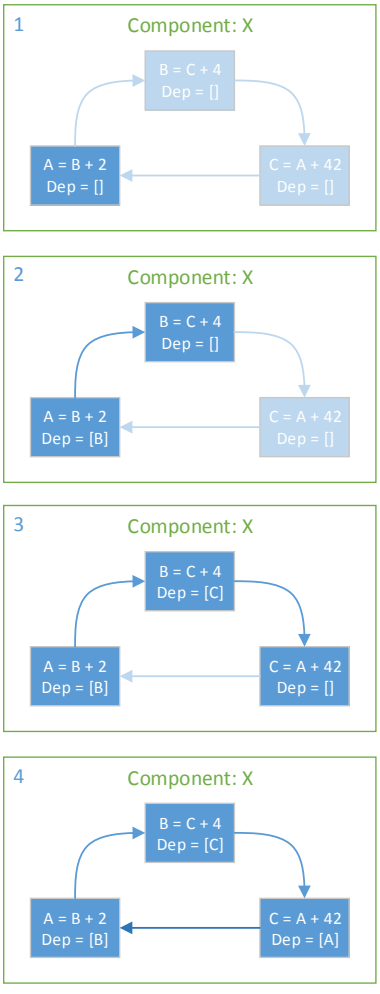


Figure 20: Four steps in detecting the cyclic dependency between three component properties A, B, and C, on component X.

Let us look at an example. Figure 20 to the right have the four steps we need to go through to detect the cyclic dependency between the three properties `A`, `B`, and `C` on the same component `X`. The observable code for each of the three properties looks like this:

```
AObs = X.property('B').select(b => b + 2)
BObs = X.property('C').select(c => c + 4)
CObs = X.property('A').select(a => a + 42)
```

Step 1: Somebody subscribes to property `A`. As we can see in Figure 20, the dependency list of property `A` is empty at this point and the dependency graph for property `A` contains just one node, property `A`.

Step 2: `property('B')` in `AObs` retrieves property `B` and adds property `B` to the dependency list of property `A`. This adds `B` to the dependency graph. It sees that the dependency list of property `B` is empty, so it just returns `BObs` observable and it is subscribed to internally.

Step 3: `property('C')` in `BObs` retrieves property `C` and adds it to the dependency list of property `B`. It then returns `CObs` since `C` does not have any dependencies. Now `C` is part of the dependency graph starting with property `A`.

Step 4: `property('A')` in `CObs` retrieves property `A` and adds it to the dependency list of property `C`. Then it sees that property `A` has

dependencies. That makes it start a depth-first search of the dependency graph, starting from property A, to see if it can find another property that also depend on property A. That is how it finds the cyclic dependency. First `property('A')` finds property B in the dependency list of property A, so it traverses the dependency graph to property B and checks the dependency list of property B, where it finds property C. It then continues to property C where it sees property A in the dependency list of property C, thus detecting the cyclic dependency. It then reports the cyclic dependency back to the original subscriber of property A.

`walk` and `get` work in the same way as `property`. The only difference is that they update template's dependency lists as they retrieve components from the component tree. Let us look at another example where `get` and `walk` is used.

Figure 21 illustrates our scenario. The `Rows` observable of template B depends on a property on a component created by template D that implicitly depends on template C, because template D cannot create any components before template C has done so. However, the `Rows` observable of template C depends on a property on a component created by template B, and thus, we have non-trivial cyclic dependency. This example illustrates a worst case scenario where none of the templates have been initialized, so no components exists on the path from template B to D[0][1], besides the component created by template A.

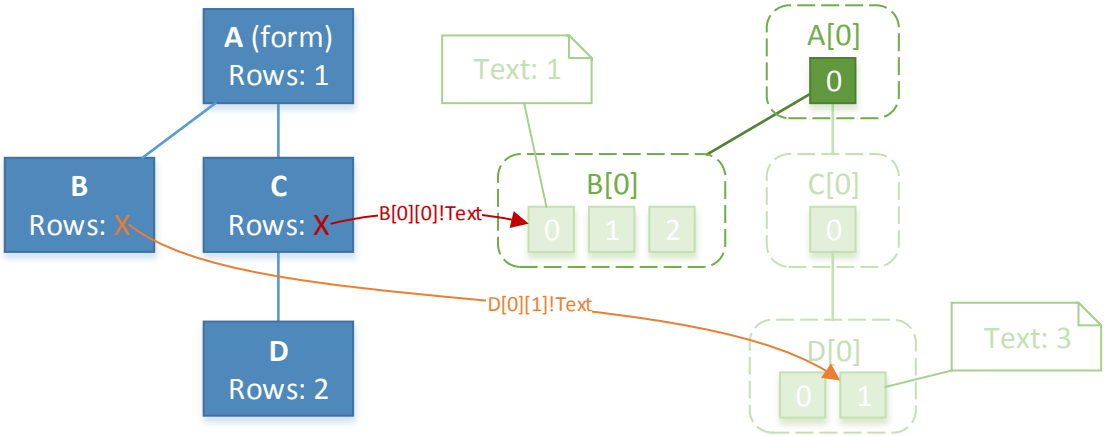


Figure 21: A cyclic dependency between templates B, D and C exists in the template tree.

The code for the two `Rows` observable for template B and template C looks like this:

```
BRowsObs = B.walk(0).get('C', 0).get('D', 1).property('text')
CRowsObs = C.walk(0).get('B', 0).property('text')
```

As we can see in the component tree in Figure 21, template A has been initialized and it has created its one component. Now the turn has come to template B. For template B to be fully initialized, it needs both components from its parent and data from its `Rows` observable. Since A has already delivered on its part, we start from where template B subscribes to its `Rows` observable. Figure 22 on page 52 has a step by step illustration of how dependency graphs develop as walk and get move about the component tree.

Step 1: Template B subscribes to its `Rows` observable, its dependencies list is empty and it is the only node in its dependency graph.

Step 2: `B.walk(0)` in `BRowsObs` updates the dependency list of template B as it retrieves the bundle A[0] from template A. It does this because template B is depending on template A to create component A[0][0],

otherwise it would not be able to walk down the component tree to find the elusive component D[0][1] and its `Text` property. Template A does not have any dependencies so nothing further happens.

Luckily, component A[0][0] already exists, so it is passed to `get('C', 0)` in `BRowsObs`.

Step 3: `get('C', 0)` in `BRowsObs` receives component A[0][0] that it asks for bundle C[0] that holds its child components created by template C. Then it adds template C, which is accessible through bundle C[0], to the dependency list on template B.

Notice how the previous dependency on template A is removed from the dependency list of template B. That is because template B and its `BRowsObs` observable no longer depend on template A to progress towards its final destination.

Then it checks if template C has any dependencies, and since it does not, template C is initialized and a subscription to bundle C[0] is made, waiting for component C[0][0] to show up.

Step 4: Now that template C is initialized, it subscribes to its `Rows` observable that uses `C.walk(0)` to get bundle A[0]. It uses A[0] to get component A[0][0]. It also adds template A to the dependency list of template C, creating a new dependency graph that contains template C and A.

Step 5: `get('B', 0)` in `CRowsObs` is passed A[0][0] that it asks for bundle B[0]. When it receives the bundle, which has a reference to template B, it adds template B to the dependency list of template C. Then it sees that template B has dependencies already, so it searches the dependency graph of template B. In the graph, it finds template C, who now has a dependency on template B. This detects the cyclic dependency and it is reported it to both template C and template B.

Detecting cyclic dependencies this way is thankfully not too costly in terms of performance, since the dependency graphs tend to stay very small, so the overhead of testing for cyclic dependencies is minimal in most cases. The is algorithm similar to Tarjan's strongly connected components algorithm, since it also makes sure to only visit nodes once. That should keep the running time at $O(|V| + |E|)$.

3.4.6 Adding a component to a canvas

Now that we know how to create a template tree and a component tree, the last piece in the puzzle is getting the components a canvas to add their visual components to. In uVis.web, a canvas is just a component that implements the `ICanvas` interface, which has two methods `addVisualComponent(vc)` and `removeVisualComponent(vc)`. A canvas

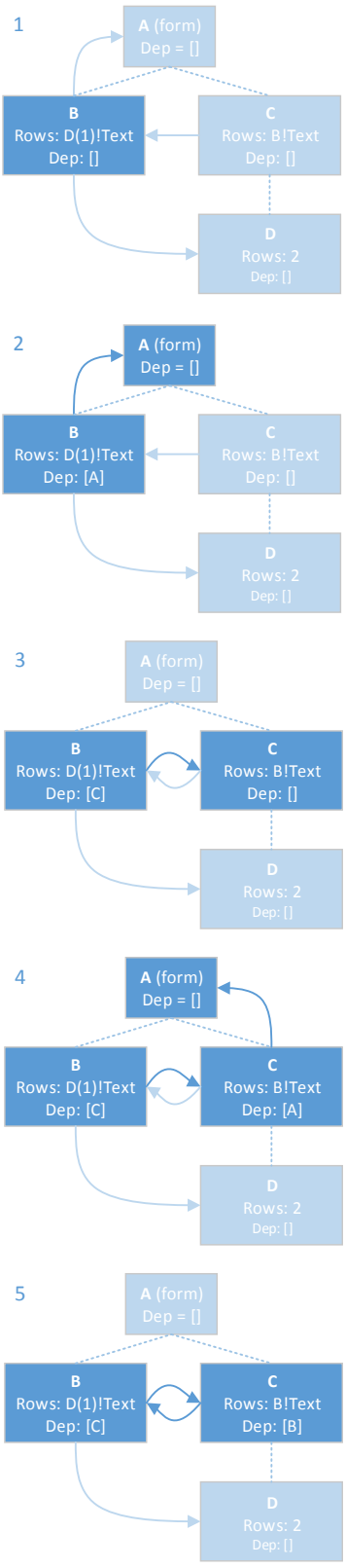


Figure 22: The five steps it takes to detect the cyclic dependency between template B and template C. Dotted lines indicate the implicit dependency that exists between templates.

can be inside another canvas, but usually the “canvas tree” is quite short, as most forms only have a few canvases.

Components get their canvas via the special `Canvas` component property that they subscribe to when they are instantiated. If the developer has not assigned a specific canvas to a component via `Canvas` template property, the component will use the default, which is the form component at the top of the component tree. The form component always has a specific canvas assigned to it, since it cannot add itself to itself. It is not controlled by the developer though, but by the kernel. The kernel provides the form component with a `ICanvas` object that will add visual components it receives to the browser’s document object model (DOM), which makes the browser render it on screen. The kernel pushes its `ICanvas` to the form component when it has finished initializing all templates.

The code in Listing 9 shows the `onNextCanvas` callback function that receives the canvases assigned to a component. It is from the abstract `Component` class that all component types must extend. Those that do must have the methods `createVisualComponent`, `setVisualComponentProperty`, `attachVisualComponentEvent` in addition to `addVisualComponent` and `removeVisualComponent`.

```
1   private onNextCanvas<T>(canvas: ICanvas) {
2       // If the visual component does not exist yet, create it.
3       if (this._visualComponent === undefined) {
4           this._visualComponent = this.createVisualComponent();
5
6           // Subscribe to properties for visual component
7           this.template.properties.forEach((name, prop) => {
8               // Skip internal properties that are not used by visual component.
9               if (prop.internal) return;
10
11              // Here we subscribe to each property. When a property returns a
12              // value the abstract method setVisualComponentProperty,
13              // that will set the value of the actual visual component.
14              this._subscriptions.add(this.property(name).subscribe(
15                  value => {
16                      this.setVisualComponentProperty(name, value);
17                  }, (err) => {
18                      console.error('Error with property observable (name = ' + name + '). ' + err);
19                  }));
20          });
21
22          // Subscribe to properties for visual component
23          this.template.events.forEach((name) => {
24              // Here we attach the component events to the visual component.
25              // The attachVisualComponentEvent returns a disposable that
26              // when triggered will detach the event from the visual component again.
27              this._subscriptions.add(this.attachVisualComponentEvent(name, this.events(name)));
28          });
29      }
30
31      // If there is a current canvas, remove the visual component from it
32      if (this._currentCanvas !== undefined && this._currentCanvas !== canvas) {
33          this._currentCanvas.removeVisualComponent(this._visualComponent);
34      }
35
36      // If this component is a canvas, notify subscribers that they can
37      // add themselves to this components visual component.
38      if (this._canvasSource !== undefined) {
39          this._canvasSource.onNext(this)
40      }
41
42      // Add it to new canvas
43      canvas.addVisualComponent(this._visualComponent);
44
45      // Save canvas for later
46      this._currentCanvas = canvas;
47  }
```

Listing 9: onNextCanvas callback function use to add a visual component to the provided canvas.

When a canvas is received, we check if the visual component is already created (line 3-17). If it is not, we create it (line 4). Then we subscribe to all the components properties and attach all events the visual

component (line 7-28). Then we check if the visual component is already added to a previously received canvas, and if it is, we remove it from that canvas (line 32-34). This allows us to move a visual component from one canvas to another at runtime.

Then we check if this component is a canvas for other components. If it is, its `canvasSource` variable will be defined and we send this component to any subscribes (line 38-40). That is how an implicit "render yourself" command from the kernel to the form component flows down through the component tree. If a component is added after the initial push of `ICanvas`'s, it will just receive it as soon as it subscribes to its `Canvas` observable.

Last, we add the visual component to the canvas we have been provided (line 43). The reason why we wait until the end, after we have pushed this component to subscribers, is performance. Web browsers generally perform much better if all visual components, i.e. DOM elements, have been added to their containers (canvases) before the container is added to the DOM.

This concludes the discussion of how the uVis.web kernel is implemented. Now I will discuss how the uVis concept can be extended to better take advantage of the strengths of the web.

4 Extending uVis with web concept

By taking uVis to the web, we also get access to the advanced layout engine available in modern web browsers. Currently, all layout in uVis is pixel based and visual components are absolutely positioned inside the canvas they are in. That is also how uVis.web works. However, there is nothing preventing uVis.web from supporting the other layout modes available through CSS. In fact, uVis.web does this already. uVis.web can create DOM elements and set their attributes, and that is all it takes. The local developer just has to specify the necessary template properties in his .VIS file to change the layout mode.

Changing layout mode and switching to the HTML mindset might be a stretch for the target audience for uVis. The big difference is between uVis component properties and CSS properties. Both are used to specify how visual components look, but where uVis component properties are calculated at runtime based formulas plus data from a data source, CSS properties are static and do not change. Instead, the HTML/DOM changes based on data, usually generated via a templating system on the server. Different web pages with different content has a different DOM, but the CSS is generally reused all pages on a website. This means a very tall DOM tree, compared to the short and simple canvas tree in uVis.

In the web world, there are many JavaScript libraries that support changing the CSS and the DOM on the client at runtime, so the web world have and are seeking some of the functionality uVis provides. Going the other way, taking in some of the layout concepts that CSS has to offer, will make it easier to create certain visualizations in uVis, especially if these visualizations should adapt or change depending on screen size or device that is used to view them. It would also mean fewer formulas would be needed in certain situations; there are after all no formulas in CSS, so utilizing other CSS layout modes, would mean more static properties in uVis applications as well.

Whether the advantages outweighs the added complexity is hard to tell without usability studies with uVis audience. However, I do think there is an interesting opportunity if a good compromise between the two worlds can be found.

Let us look at a few of the opportunities, the changes required to uVis and to the local developers workflow. Going forward, I will use the terms visual component and DOM element interchangeably, since in the world of uVis.web, visual components are DOM elements.

The visual tree must become deeper and more complex

The flow-based positioning modes in CSS require DOM elements to be nested correctly inside each other, as the nesting controls how the elements are positioned. This will result in a deeper and more complex visual tree than now; however, combining absolute positioning with the flow-based positioning should make it possible to keep the complexity of the visual tree down. The bigger problem with adapting CSS's flow-based positioning modes is that the *order* elements are placed inside each other matter and that is not the case in uVis. In fact, since uVis only works in absolute positioning and there is a z-order property that controls what components are visible if components overlap, the order templates are defined in does not matter at all. To get around this difference, we can change uVis such that the order of templates does matter, however that would make uVis a little more complex. Alternatively, we could add another property called e.g. flow-order, that would control the order of components.

Flow-based positioning mode

The default layout mode in CSS is called `static`. In this mode, DOM elements flow on screen after each other, in the same order as they appear in the visual tree. All DOM elements have a *display mode* that controls how the element will flow among its peers on screen. The two primary display modes are `block` and `inline` and we call elements *block elements* and *inline elements* depending on their setting. A block element will be positioned below the previous element on screen and any element coming after it will be placed below it. Block elements stack on top of each other like boxes. Inline elements will flow next to each other, like words on a page, and will automatically break to the next line when the current line runs out of screen space, similar to how text breaks to a new line in a text editor. See Figure 23 to the right for an example of both `inline` and `block` mode.

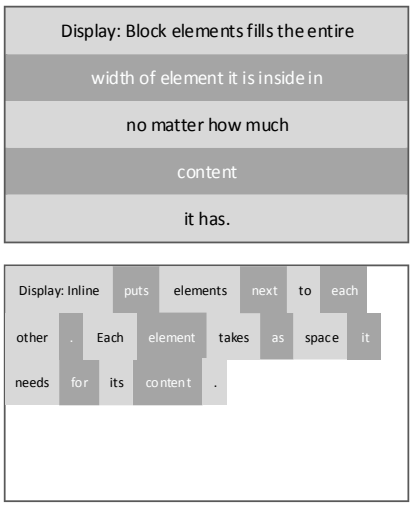


Figure 23: Top, a box with block elements inside it. Bottom, a box with inline elements inside it.

To give the developer more control over elements position in the flow-based layout mode, we have the CSS properties `width`, `height`, `margin`, `padding` and `border`. `Width` and `height` control the width and height of the content area of an element. `Margin` allows us to add a margin to the outside of an element, and padding allows us to specify how much whitespace/padding there should be added to their inside. Padding is useful if an element contains other elements. `Border` allows us to specify a border between the margin and padding. All properties are optional. See Figure 24 to the right for a visual representation of the HTML box model, as it is called. More information on the HTML box model can be found at http://www.w3.org/TR/CSS2/box.html.

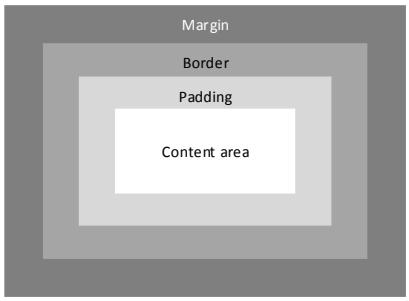


Figure 24: The HTML box model.

Some of the latest layout modes and element display modes that have introduced in the new CSS3 specification offers more complex element flow than `static` and `block`/`inline`. For a quick overview of all other options, I refer to www.learnlayout.com.

Different measurement units

One of the reasons why CSS layouts is able to change and adapt easily to different browser widths and screen sizes is that there are numerous measurement units which sizes can be specified in, most noticeably % (percentage), that offers some interesting possibilities that pixel units are missing. It gives us the option to define e.g. the width of an element as a percentage relative to another value, typically an enclosing element's width.

Support for different layouts depending on screen size or device type

If a uVis application is going to be used on devices of different types, it could be beneficial to adapt the concept from responsive web design discussed briefly in section 3.2.2. In responsive web design, the designer defines different layout configurations, i.e. **views**, that are dependent on the screen size and/or the orientation of the device, e.g. landscape or portrait.

In practice, I imagine the workflow for creating views will be similar to how a web designer creates views for a responsive web site. First, the designer creates a default layout with styling for all the components on

screen, a **default view**, and then specifies other views, e.g. one for tablets and one for mobile devices, that contains customizations to the layout from the default view. Customizations does not have to large, it could be that a visualization works equally well on both a laptop and on a 10-inch tablet, but that some of the controls on screen are adjusted slightly to be more usable with a finger instead of a mouse.

An additional view, e.g. for a tablet, would reuse all formulas and templates from the default view and just replace a few formulas where it make sense, e.g. override what is needed. The advantage to this approach over just having separate .VIS files for each view is that if the developer adds new templates to the default view or changes existing, the tablet view will inherit the new additions automatically. So as long as the changes between the two views are not big, and generally, display the same content on screen, this approach can lessen burden of keeping multiple .VIS files in sync.

Now that we have introduced the concepts of different views and a flow-based layout mode, let us look at a development scenario where we apply a few of the concepts discussed while creating the patient medicine chart we saw in the Background section using an mock-up of uVis.web Studio.

4.1 Development scenario with uVis.web Studio

In this scenario, we will recreate the patient medicine chart, used as the example in the Background section, using a few different layout modes from HTML. To help us go through the development scenario, I have created a mock-up of uVis.web Studio. Figure 22 below shows uVis.web Studio and the final output of the patient medicine chart.

Let us start from the beginning to see how the developer firsts gets to uVis.web Studio and creates a new uVis application, then how he creates the the patient medicine chart with an additional mobile view.

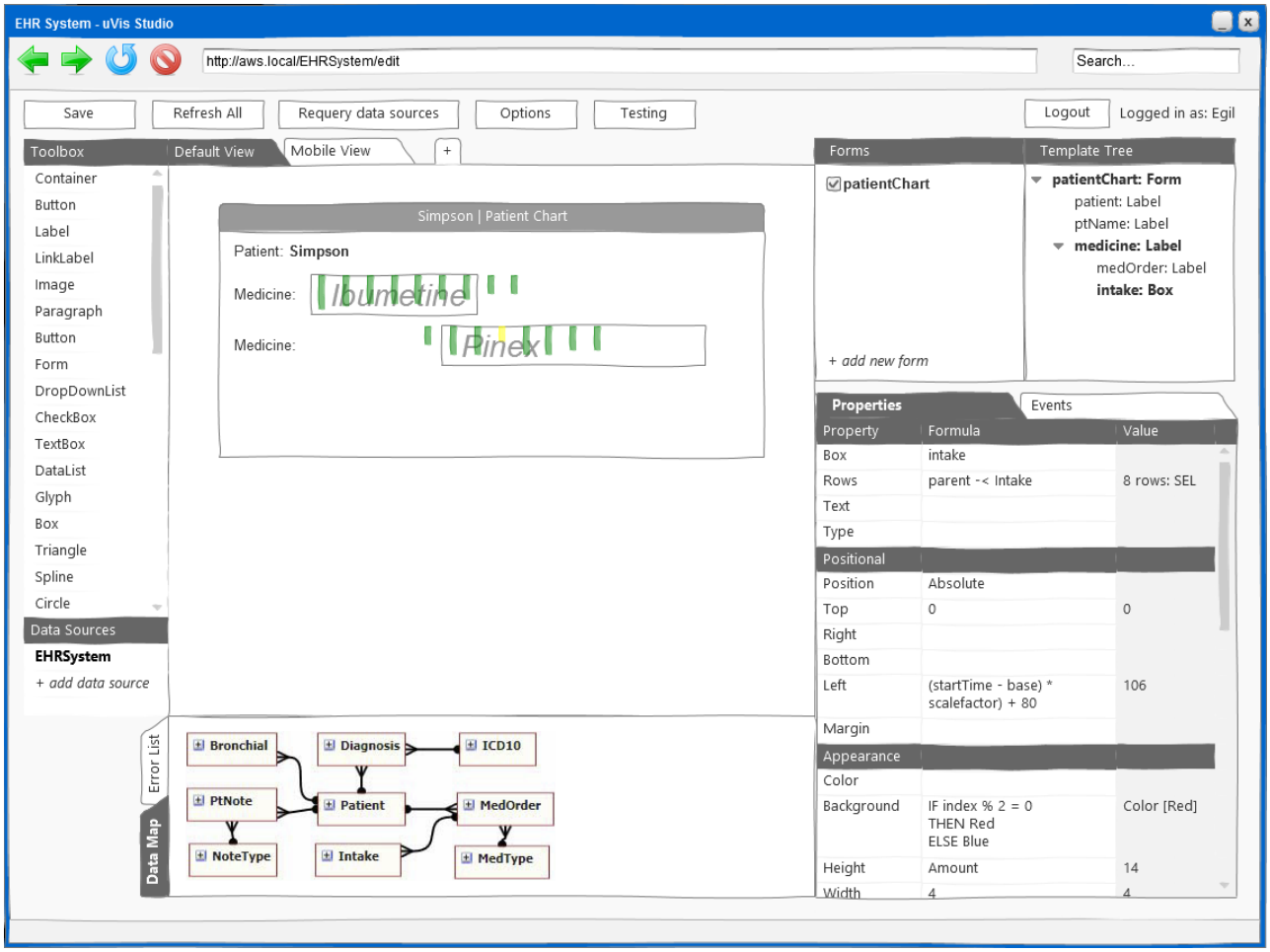


Figure 25: Mock-up of uVis.web Studio. To the left we have a toolbox that contains the components available to the local developer. Below the toolbox there is a list of available data sources. In the center, we have the currently selected view (Default view) that contains the form(s) that is being edited. It is possible to switch between views and create new views using the tabs at the top. Below that, there is box showing the data map of the selected data source, and it is possible to switch to an error list. To the right we have list of the forms added to the application, and the template tree for the currently selected form. Below there is the property list for the currently selected template, i.e. the intake box. It is possible to switch to a list of events that is/can be attached to the template.

4.1.1 Step 1: Connect to AWS and login

The developer navigates to the AWS inside his organization via his web browser (e.g. `http://aws.local`), and if required by the AWS administrator, logs in with his developer credentials.

At this point, the developer is presented with a list of existing uVis applications and the option to create a new uVis application. He chooses to create a new application, and is asked to provide a name and a URL for the application, e.g. "EHR Web System" and `http://aws.local/ehrsystem`.

When done, the developer sees an empty uVis.web Studio screen (depicted in Figure 26), with no forms, data sources or additional views defined.

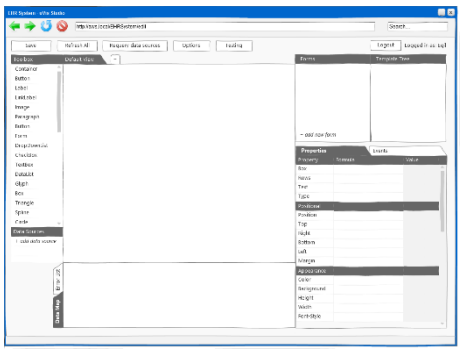


Figure 26: uVis.web Studio mock-up with no views, forms, or data sources defined.

4.1.2 Step 2: Choose data sources

Now the developer adds the EHRSystem web service to the application by providing uVis.web Studio with the URL to the web service and a name/identifier for the data source. The AWS administrator can provide the developer with predefined data sources, directly in the uVis.web Studio interface, that the developer can choose between.

4.1.3 Step 3: Create default view

First, the developer drags a `Form` template from the toolbox into the default view and names it `patientChart`. Then he configures the `Form` template to use the standard browser layout method by setting the `Position` property to `static`. He also sets the `Rows` property to `Patient where ptID = QueryString[ptid]`. That way, he can get the patient ID from the query string part of the URL. For example, the URL `http://aws.local/ehrsystem/patientChart?ptid=42` would result in the patient with ID 42 to be used.

Then he drags a `Label` template from the toolbox into the `patientChart` form, names it `patient`, and sets its `Text` property to "Patient: ". He verifies that its `Display` property is set to `inline`, which means that other components that have the `Display` property set to `inline` will be placed next to it.

Now he adds another `Label` template from the toolbox into the `patientChart` form and names it `ptName`. Then he sets the `Text` property to `parent.ptName` and finally he makes sure the `Display` property is also set to `inline`, so the patient name will be placed on the same line as the previous label. However, since no patient ID has been defined via the query string, no patient name is displayed yet.

Figure 27: After adding `patientChart` form, `patient` label, and `ptName` label.

To test what he have created so far, he uses the testing functionality of uVis.web Studio to create a dummy query string parameter named `ptid` with a value of `1`, that he knows belongs the test patient Simpson.

Now uVis.web Studio queries the web service and retrieves the patient's name, and the output rendering is automatically updated. Figure 27 shows the progress so far.

With the basic patient info done, the developer now drags another `Label` template into the `patientChart` form, and names it `medicine`. He sets its `Text` property to "Medicine:" and its `Display` property to `block`, making the label fill up the entire line. The he sets its `Rows` property to `parent -< MedOrder`. That make uVis.web Studio query the web service and update the output rendering with two rows containing the text "Medicine:" text. He sees that they are a little too close to each other, so he adds 20 pixels of margin above and below both.

Now it is time to add name and prescription period of each medicine to the form. The developer does this by dragging a Label template to the form, adding it as a child to the `medicine` label, and naming it `medOrder`. He then goes on to set the `Position` property to `absolute`, so it can be positioned relative to its container, i.e. the `medicine` label. Then he sets the `Text` property to `parent.medName`, and uVis.web Studio updates the rendering so it now displays the medicine names. The names are however located on top of the text "Medicine:", so first he figures out that he needs to offset the `medOrder` label with `80` pixels to the left, giving the `medOrder` label a new starting point (zero) that does not cover the "Medicine:" text. Then he sets the `Left` property of the `medOrder` label to `startTime + 80` and the `Width` property to `endTime - Left`, and finally he adds a border around the `medOrder` label. Now each medicine order shows up in its own row, and it is positioned a certain amount from the left based on the start time of the prescription, and its width indicate the period the prescription last. Figure 28 shows the progress so far.

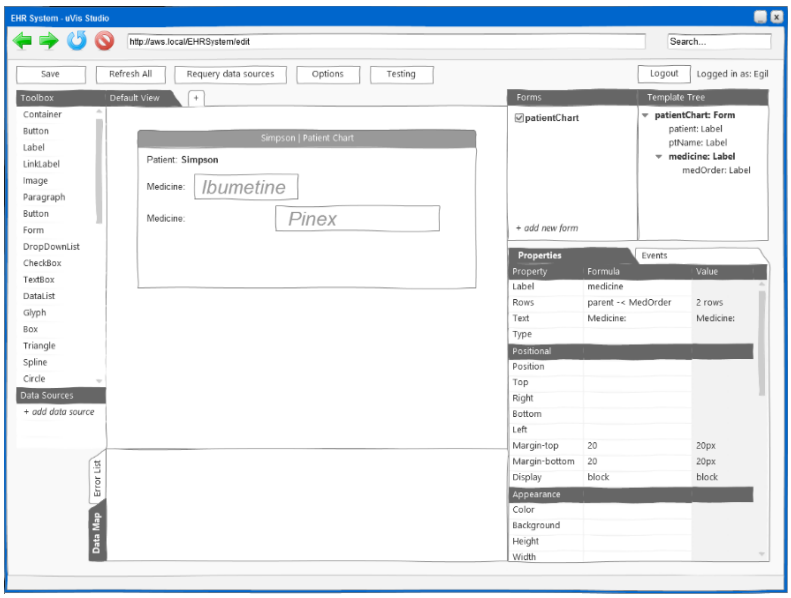


Figure 28: After adding `medicine` label and `medOrder` label.

Finally, it is time to add the intake bars. To do this, the developer drags a `Box` template form the toolbox onto the `medicine` template, and names it `intake`. The `intake` template is now a child of the `medicine` template. Then he sets the `Position` property to `absolute`, so each intake can be positioned relative to the medicine row it is in and the `Rows` property to `parent -< Intake`, causing uVis.web Studio to output all the intake boxes on top of each other in to top left corner of each medicine row. To get them to their right place, he sets the `Left` property to the formula `startTime + 80` that calculates their position. The last piece of the puzzle is the `Height` property, that he sets to the `Amount` value for each intake, making the height of each box illustrate the intake amount. The result is shown in Figure 22.

4.1.4 Step 4: Creating a mobile view

To create a mobile view, the developer opens the *add view* dialog, where he chooses the predefined "Mobile view" and makes sure to select that the new view should inherit from the default view. By doing this, the developer does not have to recreate everything he did in step 3, instead, he can simple override

any properties he wants in the mobile view. In addition, uVis.web Studio will also update the mobile view, if for example the developer adds a new template to the default view or changes a property on an existing component in the default view. Only if the property is overridden in the mobile view will it be left alone. This makes it easy to keep the views in sync and develop multiple views at the same time.

The patient chart form is quite mobile friendly already, but the developer wants to optimize its use of space by removing the “Medicine:” text next to each medicine row. Therefore, he finds the medicine label in the template tree, selects it, and clears the `Text` property. uVis.web Studio prepends a blue `+` to the property’s name to indicate it is overridden.

To finish the mobile view, the developer updates the `Left` property of the `medOrder` and the `intake` templates, so they start from `0` pixels instead of `80` pixels. Figure 30 shows how the mobile view turned out.

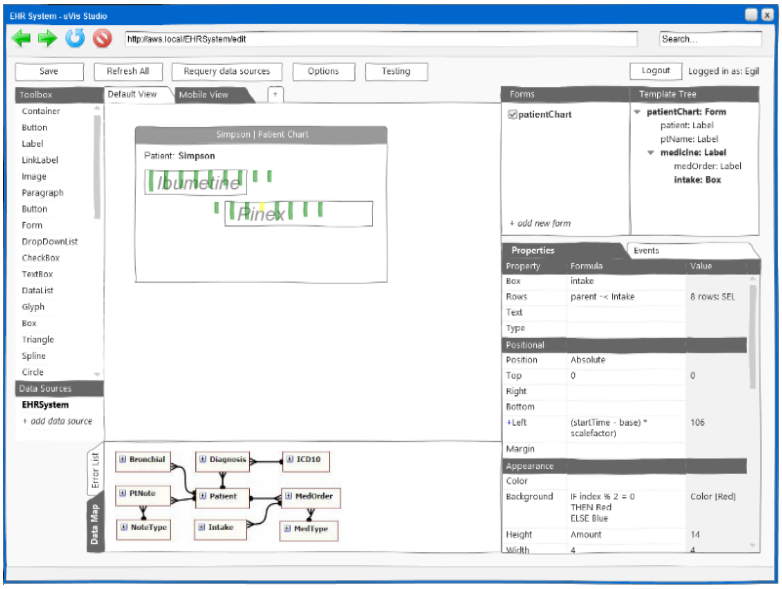


Figure 30: The mobile view of the patient chart form.

With the mobile view done, our medicine intake chart example is done.

5 Conclusion

Let us conclude this report look at other options for implementing uVis.web, with a review of the performance of uVis.web and a quick discussion on the merits of reactive programming.

Alternatives

The two big design decision that separate this implementation of uVis.web and that of uVis was using asynchronous programming and reactive programming. The question is can uVis.web be implemented using more classic methods?

Dropping asynchronous programming, i.e. doing web requests synchronous, is possible, but it is very hard to get a good consistent user experience across browsers with interactive uVis applications. Thus, I would not recommend going away from asynchronous programming. It is however possible to implement uVis.web without reactive programming or Rx. The Promises/A+ pattern (Cavalier, et al., n.d.) can be used bridge the gap between the asynchronous web requests and the more classical procedural style code in the rest of uVis.web. The Promises pattern enables a more lean way of handling callbacks and error messages from asynchronous code. One of my initial prototypes worked in this manner; it did however lack reevaluation of formulas and out of order initialization of templates.

Then the decision would be whether to reevaluate all formulas when changes happen or adapt a similar approach as afforded by Rx. If the choice is to reevaluate, the reevaluate code would most likely have to yield the UI thread from time to time so the browser does not think that the JavaScript is stuck in an endless loop and terminate it (browser differ on how sensitive they are to this). It is also possible to add a small "*Updating . . .*" message to the browser window if a reevaluation is taking long, such that the user knows the program has not crashed.

A middle way between the full-blown Rx version and the reevaluate everything version could be to establish dynamic dependency graphs between formulas, and that way keep track of what formulas needed to reevaluated when something is changed. What complicates this solution is the fact that dependencies can change as formulas are being evaluated, thus changing the dependency graphs. If I had to try a different approach, this would probably it.

Performance

The rudimentary performance testing I have done so far shows that all the additional objects created to support uVis.web does have a cost both in raw rendering performance and in memory. In the tests, it took uVis.web approximately 0.5 seconds to create 1000 visual components and add them to the DOM, 2.5 seconds for 5,000 visual components and 5.5 seconds for 10,000 visual components. That included five properties for each components. Memory usage peaks at around 250 MB for 10,000 visual components. Update speed does seem to be decent. In a simple test application, changing the height, width and border-radius properties on 5000 visual components simultaneously finished within an acceptable 1 second.

There are however room for improvement. Creating the same 10,000 visual components on screen using only plain optimized JavaScript takes just 0.5 second and used between 60-80 MB of memory (Listing 10 has all the code for the plain JavaScript solution). While this is by no means a fair comparison, since none of the uVis features exists in the plain JavaScript solution, it does underscore that there is an overhead with the current solution and space to improve, especially considering that the performance has not been a driving design factor with the current prototype.

It is hard to draw any real conclusions on the performance merits of the prototype based exclusively on synthetic tests without real data and a real uVis application. If for example a uVis application were to render 10,000 visual components on screen, it would likely also have to download almost as many database records and that might not be a realistic scenario. Add to that network and/or DWS delay, and the rendering time of uVis.web might not be significant. Memory usage is the biggest concern right now, especially if uVis.web should be able to run on tablets and smartphone type devices due to the limited amount of memory available in such devices, so that would be the first place I would spend time on optimizing.

```
var count = 10000;
var fragment = document.createDocumentFragment();
for (var i = 0; i < count; i++) {
    var is = '' + i;
    var elm = document.createElement('span');
    elm.setAttribute('class',
        (i % 2 === 0 ? 'odd' : 'even'));
    elm.setAttribute('title', is);
    elm.innerHTML = is;
    fragment.appendChild(elm);
}
document.querySelector('body')
    .appendChild(fragment);
```

Listing 10: Plain JavaScript code for adding 10,000 SPAN HTML elements to the DOM.

To summarize, performance is not great but not bad either, considering it is a prototype, and there are certainly potential for improvements without removing functionality.

Reactive programming and bringing uVis to the web

Using reactive programming and in particular Rx has opened the door to things that are very hard to do in uVis, e.g. the live search example. What the ultimate potential is with the marriage of uVis and Rx is hard to tell at this point. I do think that this project and the prototype created has proven that using reactive programming is a viable solution to the challenges facing uVis, and that it has potential to enable uVis to expand in other directions.

However, it can be a big leap to take, getting to grip with reactive programming – it certainly was for me. Since I started using Rx in my prototype three months ago, I have rewritten it from scratch four times, each time reducing the amount of code significantly, using more and more of the features of the library, ending up with cleaner more maintainable code, and I still discover new features and ways improve.

Concerning discovering the feasibility of the uVis concept on the web and with implementing a prototype that demonstrates this, I believe I have been successful.

Going forward, I would add these things to uVis.web. First, would be to add the missing features to the kernel, especially a uVis DSL to JavaScript compiler and then create an AWS. That would result in a complete client side environment for uVis, making it possible to create more realistic test scenarios. Adding support for SVG would also be high on the priority list. It would make it possible to create any type of graphic on screen. Then the biggest thing missing would be uVis.web Studio for the developer.

The experience designing and implementing uVis.web tells me that while the uVis concept is solid, certain parts would probably be different had uVis been created for the web from the beginning. A production ready version of uVis.web would likely break with the uVis concept in some places and bring ideas from the web into the mix. If it did not, it would not utilize the web as a platform to its fullest, and that would be a shame. I find it particular intriguing to think of the possibilities the layout engines available in web browsers can bring to uVis. Looking to the web could certainly provide the uVis team inspiration to innovate an already innovating product even more.

References

Campbell, L., 2012. [Online]
Available at: http://www.introtorx.com/

Cavalier, B. et al., n.d. *Promises/A+.* [Online]
Available at: http://promises-aplus.github.io/promises-spec/

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1994. *Design Patterns - Elements of Reusable Object-Oriented Software.* s.l.:Addison-Wesley.

Kjær, O. J., 2007. *Timing and Synchronization in JavaScript.* [Online]
Available at: http://dev.opera.com/articles/view/timing-and-synchronization-in-javascript/
[Accessed 12 May 2013].

Lauesen, S., 2011. *Requirements specification for VisTool - A development tool for complex data visualization - version 2.0.* s.l.:s.n.

Lauesen, S. et al., n.d. *A drag-drop-formula tool for custom visualizationq,* s.l.: s.n.

Microsoft, 2013. *Reactive Extensions.* [Online]
Available at: http://msdn.microsoft.com/en-us/data/gg577609.aspx

uVis Reference Card v1.3

Text in gray: Maybe implement later

Paths in tables - Join paths

Rows: Map.tblA Get tblA from the data map. Generate a component for each data row.

Rows: tblA Where zip = 35
Get only the rows where the zip field is 35.

tblA Where zip = txtCrit!Text
Get only the rows where the zip field matches the text in txtCrit.

tblA Where name Like txtCrit!Text & "%"
Wildcarding: Get only the rows where the name field matches the text in txtCrit followed by any characters (%).

tblA-<relB **(Left) join many**
Get all B rows that relate to tblA. (Usually there are more B rows than A rows.) Add the fields from tblA to each row. If a tblA row has no related B rows, include a null row and add the fields from tblA to it.

tblA=<relB **Inner join many**
As left join many, but omit tblA rows without a related B row.

tblA>-relB **(Left) join one**
Get all tblA rows, add fields from the related B row. If there is no related B row, add null fields.

tblA>=relB **Inner join one**
As left join one, but omit A rows without a related B row. (*Inner join one* is not allowed if the path has an earlier left join.)

Join tails

A join path can have an SQL tail with Where, Order By and other SQL parts. The parts can be in any sequence.

Note: Joins in uVis have no Select part. uVis decides what to select based on the property formulas.

tblA **Where** id=2 And time > CDate(txtCrit !Text)
Get only the rows where id=2 and time > a dialog value.

tblA -< relB **Where** tblA.id=2 And relB.code>txtCrit !Text
Get rows based on fields in tblA as well as the related B.

tblA **Where** name>"ab" **Order By** name **Desc,** id
Get the rows where ... Order them by name in descending order and for those with the same name, id in ascending order.

tblA **Group By** zip
Divide all tblA rows into groups according to their zip. Calculate aggregates for each group according to formulas using Min(x), Count(x), etc. The result has one row for each group.

tblA -< relB **Group By** tblA.zip **Where** relB.name>"ab"
Get all B rows related to the A rows and include the A fields. Include only rows with name>"ab". Divide into groups according to tblA's id field.

tblA -< relB **Group By** tblA.zip **Having** Min(id)>12
Group by zip, but include only groups where the smallest id is larger than 12.

tblA Where time>txtCrit !Text **Top** 10
Get the rows where ... and include only the first 10 ones.

Statements - event handling

Event handler formulas. Done when event triggered.

OpenForm("F", 1, "ab") Opens form F with two params.

RefreshForm("F")	Recalculates all form F instances.
Refresh()	Recalculates all open forms.
RequeryForm("F")	Requeries and recalculates.
Requery()	Requeries and recalculates.
MsgBox("Continue")	Shows a message box and waits for the user to click OK.
MsgLog("Done")	Writes the message to the log.
Me!Top=3	Sets the value of my Top to 3.
Me.addr = txtAddr!Text	Sets the addr field of my data row to the text in txtAddr.
Commit( )	Saves all changes to data rows in the database.

SQL aggregate functions

SQL aggregate functions need a Rows formula with a Group By. Each row links to a group of source rows. SQL aggregate functions count the source rows in the group, sums them, etc.

Null handling: Omits rows where the field is null.

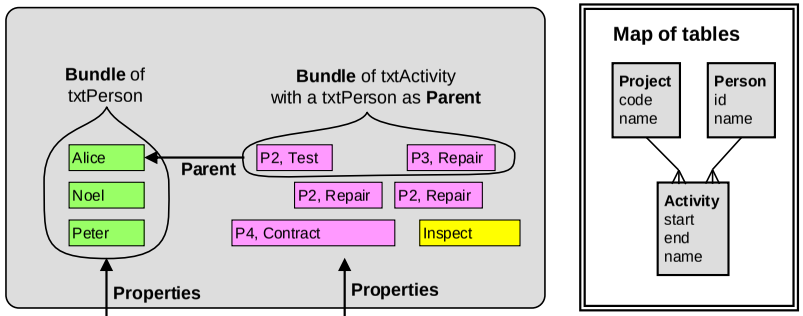
Me.Count(x)	The number of rows in my group, omitting rows with field x = null.
.Sum(x)	The sum of the x fields in my group.
Min(x)	The minimum of the x fields in ...
Max(x)	The maximum of the x fields in ...
Avg(x)	The average of the x fields in ...
Var(x)	Variance of x fields (divide by Count-1)
VarP(x)	Variance (divide by Count)
StDev(x)	Standard deviation (square root of Var).
StDevP(x)	Standard deviation (square root of VarP)
First(x)	A non-null x field in the group (there is no "first" since the rows in the group are unordered).

Bundle aggregate functions

component bundle: Each component belongs to a component bundle with one parent component. Bundle aggregate functions count the components in the bundle, sum their properties, etc.

Null handling: Omits components where the property is null.

Me!bCount(p)	Number of components in my bundle, omitting components with property p = null.
.bSum(p)	Sum of the p properties in my bundle.
bMin(p)	Minimum of the p properties in ...
bMax(p)	Maximum of the p properties in ...
bAvg(p)	Average of the p properties ...
bVar(p)	Variance of ... (divide by Count-1)
bVarP(p)	Variance (divide by Count)
bStDev(p)	Standard deviation (square root of Var).
bStDevP(p)	Standard deviation (square root of VarP)



Template for TextBox: txtPerson
Rows: Map.Person
Top: 10 + Index * 25
Left:: 10
Width: 100
Text: Person.name
BackColor: Color.LightGreen
Click: Requery( )

Template for TextBox: txtActivity
Rows: txtPerson -< Activity >- Project Where start > #1-1-2012#
Top: Parent ! Top
Left:: (Activity.start - #1-1-2012#) * 2.5
Right: (Activity.end - #1-1-2012#) * 2.5
Text: Project.code & ", " & Activity.name Default Activity.name
BackColor: Project.code=null ? Color.Yellow : Color.Purple

Paths from the Me component

Prefixes (Me, Map . . .) can be omitted if unambiguous. The first dot or bang can be omitted if unambiguous.

Me.id .id id tblA.id	**Dot:** Field in my data row. Omitting prefixes. **Table prefix:** Walk to the tblA part of my row. Get its id Field.
Me!Top !Top Top	**Bang:** Property in my component. Omitting prefixes.
!HPos(t)	**Function:** Call property function.
Index Me[Index-1]	My number in my bundle. **Indexing:** The previous component in my bundle.
Me[0] Me[Last]	The first component in my bundle. The last component in my bundle.
Form !Top Param[1]	**Form:** Walk to my form. Get its Top. **Param:** The second form parameter.
Me!txtID !Top Me!txtID[Last] !Top	**Component walk:** Walk to my related txtID box. Get its Top property. Walk to my related txtID box. Get the last in its bundle. Get its Top property.
Me>-relB.zip	**Join one:** Walk to my related relB row. Get its zip field.
Me>-relB -= txtID target: Me -= txtID Me! target!Top	**Component join:** Walk to my related relB row. Get the txtID box bound to it. Find a txtID box bound to a row related to me. Let the target property point to this txtID box. **Indirect reference:** Walk to the txtID box that target points to. Get its Top.

Path prefixes

Prefixes can resolve any ambiguity. May be omitted if there is no ambiguity.

Me.	Start in my data row.
Me!	Start in my properties.
Parent	Start in my parent component
Form	Start in my form component.
Forms	Start in the collection of all forms.
Map	Start in the map of tables.
System	Start in the collection of system items, e.g. System.Sin(a) or System.OpenForm(…).
Templates	Start in the collection of form templates.

Paths from Parent

Parent.ptId	Field in my parent's data row
Parent !Top	Property in my parent component

Paths to another Form

Forms !frmF !Top **Walk to a form:** From the open forms, get frmF. Get its Top value.

Forms !frmF(1)!Top **Form bundle:** There may be several open bundles of frmF. Get the second bundle. Get the Top of the first form in this bundle.

Forms!frmF(1)[1]!Top **Form instance:** The second open bundle may contain several open forms. Get the second form. Get its Top.

Text in gray: Maybe implement later

Line continuation and comments

```
Top + index*20          ' Comment
' Comment before line
"long text A" &  _      ' Line continuation and comment
    "long text B"       ' Comment in last line too
```

Constants

23, -23, 0, -4.9E-20	**Decimal numbers**
&h09A0FF, &o177	**Hex and Octal**
Color.Red	**Predefined colors**
"Letter to:"	**Strings**
Chr(65), Chr(vbKeyA)	The text "A"
"John" & Chr(10) & "Doe"	Two lines, Chr(10)=new line
"John" & NewLine() "Doe"	Two lines
"Don't say ""no"" "	Don't say "no"
True, False	**Booleans**
Null	**Null and DBnull**
In local format	**Date/time**
#24-12-2011#	24th Dec 2011
#24-12-11 14:15:00#	24th Dec 02 at 14:15

Operators, decreasing precedence

Nulls: Null operands give Null results and error log.

^	Exponentiation
-	Unary minus, 2*-3 = -6
*	Multiply, Result type is Integer, Double, etc.
/	Divide, Single or Double result, 5/2 = 2.5
\	Integer divide, result truncated, 5\3 = 1
Mod	Modulus (remainder), 5 Mod 3 = 2
+ -	Add and subtract numbers and DateTime. DateTime as number of days: Now( ) – 0.5
& +	Concatenation, String result
= <> < > <= >=	Equal, unequal, less than, etc.
s Like "s%n"	Wildcard compare. % any char sequence here. _ any char here. [cz] c or z here. [!cz] not c or z here.
Not	Negation. Bit-wise negation for integers
And	Logical And. Bit-wise And of integers
Or	Logical Or. Bit-wise Or of integers
Xor	Exclusive Or. Bitwise on integers
A Default B	A, but B if A is null or error. Errors are not logged. Not allowed in SQL; use IsNull(A).
A ? B : C	If A is true, B else C.
Init B	B when the form is opened. User actions may change the value later.
Partition(22, 0, 100, 10)	= "20:29" Only in SQL
a Between 3 and 9	Only in SQL
a IN (2, 3, 5, 7)	Only in SQL

Iif and Choose

Iif(a=a, b, c)	= b	**Immediate If**
Iif(a<>a, b, c)	= c	
Iif(Null, b, c)	= c	
Choose(2, a, b, c)	= b	**Choose**
Choose(4, a, b, c)	= Null	
Choose(Null, a, b, c)	= Null	

String functions

Nulls: Null operands give Null results and error report.

Chr(65)	= "A", a one-letter string with this ascii character
Asc("AB")	= 65, Ascii code for first character
Len("A_B")	= 3, length of string.
Left("abc", 2)	= "ab", leftmost two characters
Left("abc", 8)	= "abc", as many as available
Right("abc", 2)	= "bc", rightmost two characters
Mid("abcdef", 2, 3)	= "bcd", three chars, chars 2-4
LTrim(" ab ")	= "ab ", leading spaces removed
RTrim(" ab ")	= " ab", trailing spaces removed
Trim(" ab ")	= "ab", leading and trailing removed
Lcase("A-b")	= "a-b", lower case of all letters
Ucase("A-b")	= "A-B", upper case of all letters
Space(5)	= String of 5 spaces
NewLine()	= String of one new line char

Date-time functions

Null parameters: Always give a Null result.

Now( )	= current DateTime (maybe simulated)
Date( )	= current date, 0:00 (simulated)
ToDay()	The same as Date()
Time( )	= current time (since midnight)
TimeOfDay()	The same as Time()
Day(#25-12-2012#)	= 25, the day as Integer
Month(#25-12-2012#)	= 12, the month as Integer
Year(#25-12-2012#)	= 2012, the year as Integer
Weekday(#25-12-2012#)	= 3 (Sunday=0)
Hour(# ... 13:14:15#)	= 13
Minute(# ... 13:14:15#)	= 14
Second(# ... 13:14:15#)	= 15

DateAdd("d", 4, #30-12-2012#) = #03-01-2013#

"y" "m" "d" "h" "n" "s"

Year, month, day, hour, minute, second.

Timer( )	= Number of seconds since midnight, with fractional seconds.
DateSerial(2002, 12, 25)	= #12/25/2002#
TimeSerial(12, 28, 48)	= 0.52 (Time 12:28:48)

Math functions

Sqr(x)	Square root of x. Sqr(9) = 3.
Sin(r), Cos(r), Tan(r), Atn(x), Acos(x), Asin(x)	Trigonometric functions. R measured in radian (180 degrees = π = 3.141592 radian) Sin(0) = 0, Sin(3.141592 / 2) = 1.
Pow(x,y)	X to the power of y. Pow(2, 3) = 8.
Log(x, y)	Logarithm of x with base y. Log(8, 2) = 3.
Rnd( )	A random double number between 0 and 1.
RndInt(n,m)	A random integer between n and m-1.
Abs(x)	Returns x for x>=0, -x otherwise.
Hex(x)	Returns a string with the hexadecimal value of x. Hex(31) = "1f"
Oct(x)	Returns a string with the octal value of x. Oct(31) = "37"
Sgn(x)	Returns 1 for x>0, 0 for x=0, -1 for x<0
Int(x)	Rounds x down to nearest integral value
Fix(x)	Rounds x towards zero

Format function

Converts a value to a string, based on a format string. Format characters that are not placeholders, are shown as they are. Backslash+character is shown as the character alone, e.g. \d is shown as d.

Numeric placeholders

0	Digit, leading and trailing zero okay here
#	Digit, no leading or trailing zero here
.	Decimal point (shown as the local variant)
,	Thousand separator (shown as the local variant)
e- or e+	Exponent or exponent with plus/minus
%	Show number as percent

Format(2.3, "00.00")	= "02.30"
Format(2.36, "#0.0")	= "2.4"
Format(0.3, "##.0#")	= ".3"
Format(32448, "(00)00 00")	= "(03)24 48"
Format(32448, "##.#e+0")	= "32.4e+3"
Format(32448, "##.#E-0")	= "32.4E3"
Format(0.5, "#0.0%")	= "50.0%"
;	Separator between formats for positive, negative, and zero values:
Format(-3, "000;(000);zero")	= "(003)"

Predefined numeric formats

g (general), c (currency), f (fixed), p (percent), e (scientific), n (with thousand separator), x (hex)

Date/time placeholders

Example: DT = #3-2-2002 14:07:09# (Sunday)

Format(DT, "yyyy-MM-dd HH:mm:ss")
= "2002-02-03 14:07:09"

Format(DT, "dddd yy-MMM-d a\t HH:mm")
= "Sunday 02-Feb-3 at 14:07"

yy	Year, two digits "02"
yyyy	Year, four digits "2002"
M	Month, no leading zero "2" (Interpreted as minutes after h)
MM	Month, two digits "02" (Interpreted as minutes after h)
MMM	Month, short text "Feb"
MMMM	Month, full text "February"
d	Day of month, no leading zero "3"
dd	Day of month, two digits "03"
ddd	Day of week, short text "Sun"
dddd	Day of week, full text "Sunday"
H	Hour, no leading zero, 24-hour clock
HH	Hour, two digits, 24-hour clock
h, hh	Hour, 12-hour clock
tt	Show AM or PM here, 12-hour clock only
zz, zzz	Show time zone, +1, -8.30
m	Minutes, no leading zero "7"
mm	Minutes, two digits "07"
s	Seconds, no leading zero "9"
ss	Seconds, two digits "09"
f, ff ...	Fractions of seconds

Predefined date formats

G (local date and/or time), D (long date), d (short date) T, t (local long/short time), u (local GMT sortable) ...

Conversion and test functions

Nulls: Null operands give Null results and error log.
System prefix can be omitted if unambiguous.

System.CInt("2.6")	= 3
CInt("2.6")	= 3, omitting prefix.
Round(2.6)	= 3.0000 (Double)
	Rounding down: See Math functions Int, Fix.
CByte("37")	=37. Overflow outside 0..255
CLng("99456")	= 99456
CDbl("-2.6")	= -2.6
CCur(1/3)	=0.3333 (always 4 dec)
CDate("23-10-03")	= #23-10-2003#
	Uses local settings for input format
CDate(1)	= "31-12-1899"
CStr(23)	= "23". No preceding space.
CStr(#23-10-2003#)	= "23-10-03 00:00:00"
	Converts to local date format
IsNull(A)	True if A is null. See also operator Default.
IsDate(v)	True when v is a date or a string that can be converted to a date
IsNumeric(v)	True when v is a number or a string that can be converted to a number.

Common component properties

Rows: tblA	For each row in tblA, create a component.
Rows: txtB	For each txtB component, create a child component. Its parent component is txtB.
Rows: txtB -< tblC	For each txtB, create a bundle of child components. The bundle has a component for each tblC row that is related to txtB's row. The bundle's Parent is txtB.
Name	The name of the template.
Form	My Form component. Read-only.
Parent	My bundle's parent component. Read-only.
Index	0, 1, 2 ... My index in my bundle. Read-only.
Top	Pixels from my canvas top to my top.
Bottom	Pixels from my canvas top to my bottom.
Height	Pixels from my top to my bottom.
Left	Pixels from my canvas left to my left.
Right	Pixels from my canvas left to my right.
Width	Pixels from my left to my right.
BackColor	Color of my inner area.
BorderColor	Color of my border.
Weight	Number of pixels across my border.
Visible	False if I am not visible. **Default:** True.
Canvas	The component where I am located. It may scroll and clip me. My left and top don't change when it scrolls. **Default:** My form.
ZOrder	Integer. On my canvas, I am above components with a lower ZOrder.
Layout	Can compute Top, Left, etc.

Event handlers Act when the event occurs

Click, DoubleClick, KeyUp, KeyDown
MouseDown, MouseUp
MouseMove, MouseOver, MouseLeave