Programming Spindle

A guide to vaporware

History (should probably be ignored)

I think that my love for invented languages began when I first read Tolkien - probably when I was ten or so. Later, I actually tried to do it myself in college with moderate success. It was by no means complete, but I took it as far as I really wanted. I wrote a short story, a myth about the language itself, in the language.

I have also always been one who feels the need to do it myself - to create instead of just learn. It has even prevented me from really learning how to play an instrument, because I constantly want to jump ahead to writing a song before I know how to play one...

...but I digress - programming has not been the same. I spent a long time learning before I ever concieved the idea of creating a language myself. Creating programs was enough for me. However, after I had been doing it long enough to form an opinion, the hubris and urges to try to do something better crept up again.

Templating languages are a gateway drug

Going back pretty early, to my first professional programming gig, I found myself cutting my teeth on some fancy Ajax stuff back when it was AJAX (and we actually used XML!). I found the thicker client approach liberating, but the environment pretty hacky. One of the things that killed me early on was the string concat. I searched for a nice simple JS based templating language, and coming up short thought I could probably whip up something really simple pretty quickly. I didn't even want to call it a templating language - just something to avoid string concat of simple strings. A friend of mine who was a bit of a perl hacker just smiled knowingly, "That's how it starts," he told me, "They start small and before you know it, you have a full templating language." He was right of course; not immediately, but in other projects to come.

I moved on from that project, that job, and did a bunch of other things - working with CMSs, fighting myself from making my own (including templating language), and got into other UI technologies like Flex. At the time, RIA was the buzzword, and Adobe was all over it. When I came to my next greenfield project,

HTML5 is not enough

Templating languages are a gateway drug

Write about Preface here.

Write about Roadmap here.

Write about Goals here.

Features

This language did not grow out of a desire to write a programming language (that part came later), it really just grew out of a need. I was not happy with the current solutions. After getting really close to actual language design with open spaces, I felt like I had something almost resembling a 0.1 a language by taking all of the different parts and stitching them together with some extra glue and features. OpenSpaces is part framework, part franken-language. There is an incremental step that I've also been looking into I've been calling StrongSpaces which stays very close to the original form, but cleaning it up to be a little less franken, and have better type safety.

When I actually decided to forge ahead and try to take all of the things I had learned and make a language, I did have to stop myself and try to focus my feature-set. I did not want a kitchen-sink language. To try and narrow down, I stopped and tried to look at the needs of applications. The following is where my train of thought led me.

Application Paradigms

When considering a web application language (not to be confused with a general purpose language), it is important to consider all of the different types of applications that might use it. I do not think it wise to assume that all types of applications can be considered upfront and catered to, because I think that is a little bit of the mentality that got us into this mess in the first place (html being a document format). However, considering the different types of applications and interactions required would help to make sure that there are not obvious gaps which need to be filled in.

Applications, Not Programs

This should not be a matter of considering all possible programs, but more specifically GUI applications that would use the internet and most likely be run in a browser, or browser like environment (Adobe Air, Appcelerator Titanium). Technically, a local only, or server only application should be possible, but is not the main target.

Paradigms

It seems to me that there are only a handful course grained application types or paradigms. By this, I do not mean "word processor" vs "instant messenger", but rather "document" vs "realtime".

- **Data-centric** - This really makes up the bulk of stuff on the web. From informational pages, blogs news sites, social websites, and ecommerce to CRUD applications and I think even iTunes falls into this category. Anything that would have a considerable amount of application logic and UI devoted to browsing, filtering, searching, and optionally data entry.
  Storage - Something using a database is likely at least a little data-centric. Usually multiple types of data or at least many of the same kind of data.
  Interface Patterns - master/detail, list/drilldowns, search/results, tags/tag clouds, related items, filtering controls
  Interaction Patterns - The reason that this really took off on the web was the whole

hyperlink/url thing. When people say web scale, its because we start talking billions of different documents. The complexity lies in the amount, as opposed to the richness of any individual document. Finding the right data in that mess is the main challenge of the UI and the typical interaction typically involves a lot of navigation to get what is needed. This could be multiple levels of browsing navigation like tabs and sidebars, searching, contextual links, etc.

- **Document** - This has always been more in the realm of the desktop app, typically because it requires more rich and custom UI, especially depending on what is rendered/edited. Typical sorts of applications would be word processors, IDEs, photo editors, movie editors, etc. The main differentiator is that the the UI devoted to consuming or editing the document is considerable in comparison to finding the document to edit.
  Storage -As opposed to a data centric paradigm with its database, the document paradigm is the land of the file. When it comes to the web, even when a database is used, if it feels like something that would typically be a file to the end user, thats the sort of thing that would fall into this category. There is one type of data and it can be encapsulated into this single document that would be stored in a file or database. The association between a file extension and its application typifies the mentality of this paradigm.
  Interface Patterns - palette/canvas, context menus, copy/paste, undo/redo, saving, tools and other types of advanced controls targeting a specific document.
  Interaction Patterns - Typically a single document is edited at a time unless multiple instances of the app are opened, or possibly multiple documents are opened at once with tabs. If it is networked, there might be locking or versioning.

- **Realtime** - Also typically in the desktop realm, these applications basically have a process running in an event loop to do some operation as it is being interacted with and then give feedback or data from other processes. Some are passive and only receive, some only send, and some do both. This is a little vague, but I think it is a good way to categorize applications like chat, video chat, games (both networked and non), stock information, etc. A game is an interesting example because even non-networked games really fall into this category because of the game loop. User interactions and application logic have to be able to handle the continuous input, update loop.
  Storage - A great distinguishing aspect of this paradigm is that data is typically much more ephemeral. While it may get stored, what makes it realtime is that the data is constantly new, and storing old stuff is more historical. While chat apps typically have a chat log, creating it is not the purpose of the app. With games, the bulk of the interaction may get communicated as data for network play, but it is never stored. Only a reduced set of information is really stored like game progress, but that is really just so the game can be stopped and resumed later.
  Interface Patterns - There aren't really any common interface patterns other than it is updating all the time. Also, notification are common for this sort of thing.
  Interaction Patterns - Instead of the more discrete steps common with data centric, or the discrete edits one would make with document apps, realtime really has an aspect of time, or at least this notion of an an interactive loop, where really the function of the app is based on the interaction itself, as opposed to the interaction being a means to an ends.

Blurring the lines

Do applications fit into perfect boxes, of course not. Applications may fall into more than one category pretty easily. While a blog may be data centric, a blog admin is a bit of a mix. It is data centric in basically the same ways as the blog is, but because of the admin functionality, operating on a single blog entry can be quite sophisticated and really be considered a document app in its own right. Likewise, Google Docs falls pretty clearly into the document paradigm, but when used collaboratively, is also realtime.

Social apps like Twitter and Facebook can also be a little bit of a blur between data and realtime. Twitter falls much more heavily on the realtime aspect, while Facebook likes to keep track of and share data, while also trying to have some of the realtime aspects of Twitter. Both of them allow for heavy amounts of linking and network effects so typical of the data centric paradigm.

It is not so much important that applications belong to one paradigm, as much as each paradigm is important to think about when looking at features and programming paradigms available in Spindle.

Language Features for each Paradigm

Each of these paradigms has certain challenges that come with it, and therefore also language/ platform features which can help overcome those challenges. The question of whether some of these things belong in the language or just libraries is perhaps the most important question of all, but there is a reason why I think a new language could be very potent here, and it is largely because of what happens when a less general purpose language can take platform constructs and make them first class. I just have to make sure that what I choose to include does not become limiting.

Organized by paradigm:

Data Centric

- Good integration with databases - This may seem entirely like a platform feature, but there are ways a language can be more amenable here. For example, all databases really have a notion of a key, or id. I think a first class notion of this might make sense. Also, the ability to serialize to common data formats like JSON would be helpful working with document databases like MongoDB or CouchDB. Declarative language constructs can go a long way towards allowing first class DSLs for query languages also.
- Data-binding - Document and realtime apps tend to have a richer editing model, but fewer inputs or pieces of data that need wiring. Data-binding is amazing for this and really needs to be first class.
- Schemas - Data-centric apps typically have many kinds of data and get a lot of user input. That data needs to be validated, and additionally certain static guarantees would be useful from that information. Completely dynamic typing might be a little too loose.
- Restful interactions - Making restful interactions easy and first class should be easy to do. Not sure what kind of language features I can do with this.
- Data transformations - Lets face it, most web applications mostly just shuffle data back and form worrying only about two things, storing/retrieving the data and transforming it into something visual.

Document

- Access to more graphical primitives - This really falls into platform features, but I can't

help but feel like some language features could be helpful here. One thing that could be useful to get kick started, is language parity to allow a 1 to 1 transformation to and from SVG. This also supports declarative language constructs.
- Undo support – One way I can think of making undo support easier is through the ability to easily describe an edit and a first class transformation that can be reversed.

Realtime

- Socket support – Definitely a platform feature, but also need to make sure that these types of interactions are easy to do. Simply having first class notion of events could be enough to do well here.

All – Obviously there are also many features that will be important for any GUI focused language/ platform. These are the ones that make sense to me:

- Declarative Syntax – Most platforms/languages are going in this direction even if they have to use XML as a crutch. It just make sense for reasons I won't even go into. Note that making this first class instead of just for the UI like XAML will make it really widely useful.
- Data binding – Already mentioned, but this works amazingly well with the declarative syntax to avoid imperative style code in the UI. It makes data wiring declarative, and not just layout/visual component hierarchy.
- Pattern matching/Data transformation – CSS/XPath/XSLT/jQuery – enough said? Especially in the data centric paradigm, there is just a lot of data transformation, however as shown with XSLT, if you can think about the UI like data, matching and transforming can be extremely powerful as means of being concise, reusing code, creating abstractions, and putting code in the right place.
- Immutability – This is still up in the air speculation, but there is a big move towards immutability by default, and I think it makes a lot of sense for all the reasons mentioned by everybody else. Also interacts well with transformations (see XSLT).
- Controlled Side Effects – Going hand in hand with immutability, controlling side effects can just help programs be easier to reason, but also allow for security with untrusted code, and also many kinds of optimizations.
- Security – I think that something resembling the object capability model is likely the best way to achieve the kind of holistic security that web apps really need. Also immutability and controlled side effects help here. It will be extremely important for mash ups with untrusted code, but also just needs to be considered because it is such a big threat. One intention for Spindle is to allow plugins using untrusted code, and I don't know if it can be done with anything less than language level security (similar to Facebook's FML)
- Rich, integrated templating language capabilities – I think this is really key. Let's keep the 'templating' type tasks inside the language, in other words, the UI code, and not the just the end result like HTML. There may be room for different libraries or frameworks or something in this space, but it should never require a separate templating language living external to the language like velocity/django templates.

Write about Spindle as a Data Format here.