Idyll: A Domain Specific Language for the Rapid Development of Interactive News Articles

Matthew Conlen
University of Washington
Seattle, WA
mconlen@cs.washington.edu

Jeffrey Heer
University of Washington
Seattle, WA
jheer@cs.washington.edu

ABSTRACT

It is becoming increasingly common for news outlets to publish rich, interactive digital stories. Often referred to as *interactives*, these stories have the potential to engage a large audience, but are expensive and time consuming to produce. The custom code required for an interactive — code that is often developed by a non-expert programmer under the stress of a deadline — can be error prone and suffer from performance problems, in addition to being difficult to understand and hard to reuse. In this paper we introduce *Idyll*, a compile-to-the-web language for creating interactive narratives. Idyll combines a human readable markup language with a framework for embedding reactive JavaScript components in-line with text. Drawing on experiences with newsroom graphics production, Idyll distills a set of best practices for creating and deploying interactive stories to the web. The goal of the project is to accelerate the production of interactives by reducing the amount of custom code required to create them. It also looks to enable a closer collaboration between technical and editorial contributors, and encourage the development of JavaScript components that can more easily be reused across articles.

KEYWORDS

interactivity, data visualization, tools for journalism

ACM Reference format:
Matthew Conlen and Jeffrey Heer. 2017. Idyll: A Domain Specific Language for the Rapid Development of Interactive News Articles. In *Proceedings of Computation + Journalism Symposium, Chicago, Illinois, USA, October 2017 (C+J 2017),* 5 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Publications like *The New York Times*, *Washington Post*, *The Guardian*, and *FiveThirtyEight* are known for producing high-quality multimedia narratives. Often referred to as *interactives*, these stories have the potential to engage a large audience, but are expensive and time consuming to produce.

In order to add interactive elements to an article, custom HTML, CSS, and JavaScript code needs to be written. This use of custom code can clash with the typical process of publishing an article, which involves entering text and images into a content management system (CMS). Some CMSs allow custom code to be added to posts directly, but this is a tedious and error-prone method of embedding code and so is usually undesirable. Instead, in order to support publication of interactives many newsrooms either (1) use an `<iframe>` tag to insert the code, or (2) develop a completely separate page that lives outside of their existing CMS.

There are problems with both of these approaches. Embedding interactive content using an iframe narrows the design possibilities afforded to the embedded content. Content that has been included in a page via an iframe cannot interact with text on the containing page, and requires additional code in order to proxy events from the top level page [13].

Maintaining a separate public facing page allows for more flexibility in design, but adds an additional technical burden. It also adds a new problem: because the CMS is eschewed, article copy needs to be inserted directly into HTML. This makes it more difficult to edit and make changes to the text, and also makes non-technical (editorial) staff reliant on those with programming knowledge, instead of being empowered to make text updates directly.

The custom code required for the interactives is often developed by non-expert programmers under the stress of a deadline. Because of this, the resulting web pages can suffer from performance issues, and the code itself may be difficult to understand and hard to reuse across articles. In order to improve the code quality, many newsrooms create internal frameworks that help to structure common tasks. These frameworks require a large up-front investment in development time, and also require maintenance over time as tools and techniques change.

In response to these issues, we created *Idyll*, a markup language designed for authoring interactive narratives. Idyll combines a human readable markup language with a framework for embedding reactive JavaScript components in-line with text.

The project attempts to accelerate the production of interactives by reducing the amount of custom code required to create them. Drawing on experiences with newsroom graphics production, Idyll eliminates the need for a large class of boilerplate code, and helps users adhere to a set of best practices for creating and deploying interactive stories to the web. It also hopes to enable a closer collaboration between technical and editorial contributors, and encourage the development of JavaScript components that can more easily be reused across articles.

2 RELATED WORK

Many projects attempt to make it easier to write clean text markup and publish it as a static web page. For example, Jekyll [1] is a project that allows users to write plain text or Markdown [7] files

C+J 2017, October 2017, Chicago, Illinois, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and deploy them to the web. The tool will take in a Markdown file, convert it to an HTML web page and publish it as a blog entry. These tools typically do not provide a mechanism that allows JavaScript to be tightly integrated with the text.

Visdown [8] extends the idea of compiling Markdown to HTML pages by allowing authors to specify data visualizations in their markup. A user can provide a declarative Vega-Lite [11] specification directly in the Markdown file, and this is used to render a chart in the final markup.

Bret Victor has written persuasively in favor of adding certain types of interactivity to writing that would traditionally be presented as static text [14, 16]. His essay *Explorable Explanations* [15] illustrates some of the types of interactions that we hope to enable with Idyll. Victor released a JavaScript library, Tangle [17], that helps users add reactive variables to their documents. Chris Olah has written on using interactivity to explain complex topics, suggesting that a more interactive publishing platform could expedite the dissemination of new research ideas [9].

The typical process for creating interactive documents involves hand-writing a lot of custom JavaScript and HTML. It can quickly become difficult balancing the narrative portion of the project with the nitty-gritty details of code. To this end, The New York Times developed ArchieML [12], a markup language designed to make it easy to pull text into JavaScript code. A core idea with ArchieML is that code and text should be separated because they deal with very different concerns. Text needs to be edited for content and clarity, often by someone who doesn't care to look at code. Developers will need to integrate that text with their code at some point, but typically aren't concerned with grammar while they are writing JavaScript.

While ArchieML makes it easy to pull text into code, Idyll makes it easy to include JavaScript components in text. With this approach the relationship between code and text becomes much easier to reason about from an editorial perspective, and it becomes feasible to make nuanced changes to where components appear in text and how they interact with the page. It also eliminates the need for a large class of boilerplate code that binds JavaScript functionality to specific locations in the web page. With Idyll's approach the process of including an interactive component in text becomes much closer to, say, using a CMS to embed an image in a post.

Another project that addresses combining code and data with text is Stencila [5]. Stencila allows users to write text in a "what you see is what you get" (WYSIWYG) style text editor, and embed executable code cells in the text. The project focuses on streamlining the process to publish reproducible research, and uses ideas from "computational notebooks" such as Jupyter [10].

Idyll is not the first attempt to design a domain specific language (DSL) to solve a problem that newsrooms face. D3 [6] is a DSL that allows users to quickly create data visualizations inside of a JavaScript application. PersaLog [4] is a DSL for adding personalizations to news articles. Idyll attempts to enable a more generic type of customization than PersaLog, and considers documents in a more structured manner than D3. For example, a user might use Idyll to easily embed a visualization created with D3 in their article.

3 USAGE SCENARIOS

The section presents scenarios of envisaged usage. These examples do not encompass all possible uses of Idyll, however they are illustrative of the types of workflow that the tool enables.

3.1 The Newsroom Developer

Consider a newsroom developer tasked with building an interactive feature for an upcoming article. She is provided a text document that contains the final edited copy for the article, and needs to transform this into a standalone web page that can be published on a static web host. She also needs to add a custom chart to the article, along with a button that will toggle the mode of the chart when clicked.

The first thing the developer does is copy the text from the provided document and paste it into an Idyll markup file. She configures Idyll to open the markup inside of an HTML template that contains the organization's logo and branding.

At this point the developer is ready to start writing code. She runs the command to start Idyll, causing the HTML page to be opened in her web browser. She modifies the markup file to include following code:

```
[var name:"mode" value:1 /]

[CustomChart mode:mode /]
[Button onClick:`mode = 1 - mode`]
  Toggle Mode
[/Button]
```

This code creates a variable `mode` that toggles between 0 and 1 when a button is clicked. It also instantiates a component called `CustomChart` that accepts the `mode` variable as an input parameter.

She then creates a new file, `custom-chart.js`, where she will write d3 code to display the custom visualization. In her JavaScript she extends an `IdyllD3` component[1], and has it update the visualization based on the `mode` parameter. As she makes changes to the JavaScript, Idyll automatically refreshes her display so that the latest version of the article and her custom code is always on-screen.

The developer is able to focus on writing code specific to a single component on the page. She doesn't need to write any code that binds to an id in the article, or maintains application state, and doesn't have to write any code that listen for events, instead Idyll handles all of this.

After she finishes the component, the developer runs a command telling Idyll to compile the final output for publication. This command generates a folder containing an HTML, CSS, and JavaScript file. She takes these files and uploads them to her organization's static file server. The article is now live on the internet and the developer can move on to her next story.

3.2 The Internal Collaboration

This second use case considers Idyll in a larger organizational context. Imagine a collaboration between an editor, a writer, and a

[1]For a full example of using Idyll with D3, see https://idyll-lang.github.io/idyll-d3-component/.

newsroom developer. The three collaborators use Google Docs to write and edit the text of an article using Idyll markup. Google Docs provides functionality so that multiple people may edit and comment on a document in real-time.

The developer runs a small script that keeps the text from the document in sync with a local file, allowing him to pull in changes to the text to his local computer. He uses the output of this script as the input to Idyll, allowing changes to the document on Google Docs to immediately be reflected in Idyll's rendered output.

This group has used Idyll for several previous projects, and have maintained a repository of useful components that they've created. The developer has configured Idyll so that the components in this internal repository are always available, so any of the collaborators may add one of these components to the text by simply adding a component tag.

For example, they could have a specific style of map that they prefer to include. The writer could add it to the article by inserting a tag:

```
[InHouseMap lat:47.6062  long:-122.3321 zoom:8  /]
```

The three collaborators have a detailed discussion about which components would be most effective to include in this particular article, and where they should be displayed. It is easy for any of them to change the locations of the components in the document. They decide on a configuration and realize that another custom component will need to be written.

The developer writes the JavaScript for the new component, and inserts the corresponding tag into the document. He also checks that component into the internal repository so that it will be readily available for future articles. When the document is complete, the editor takes one final look at the text in the Google Doc. The developer then runs the command to compile the output, and pushes the final package to a static hosting server.

4 DESIGN

Idyll emerged from the goal of designing a markup language that would satisfy the following requirements:

- The markup should be clean enough that a non-technical writer or editor could understand and edit it, and be extensible via JavaScript by programmers or designers.
- It should integrate well into existing workflows, and integrate easily with popular existing tools (for example, D3). To facilitate this, the output should be a complete web page that can be published directly, or embedded in another page via an iframe.
- The tool should eliminate as much boilerplate code as possible for techniques common in interactive articles, such as reactive variables, and scroll-driven storytelling.
- The tool should eliminate the need for a complex JavaScript build setup, and enable basic best practices by default (for example, server-side rendering of static content, and code minification).

These goals were synthesized through informal conversations with journalists and web developers, and through the first author's experience developing news applications with *FiveThirtyEight* and several other publications.

4.1 Examples

Figure 1 shows an overview of Idyll's basic syntax. Idyll's markup allows users to concisely embed reactive variables in their text. This is an example from Tangle's documentation, recreated using Idyll:

```
[var name:"cookies" value:3 /]

When you eat [Dynamic value:cookies /] cookies,
you consume [Display value:`50 * cookies`] calories.
```

The above code displays the sentence "When you eat 3 cookies, you consume 150 calories." The number 3 is dynamic, meaning that a reader may change its value, updating the `cookies` variable, and in turn updating the displayed number of calories consumed.

Idyll also makes it easy to load datasets and instantiate JavaScript components. The following code parses a CSV file containing information about the water levels in Lake Huron. The first 10 rows of the file are rendered in a table, and then a Vega-Lite component is used to plot the water level of the lake over time in a line chart.

```
[data name:"lakeHuron" source:"lake-huron.csv" /]
[Table data:`lakeHuron.slice(0, 10)` /]
[VegaLite data:`lakeHuron` spec:`{
  mark: "line",
  encoding: {
    x: { field: "time", type: "temporal" },
    y: { field: "LakeHuron", type: "quantitative" }
  }
}` /]
```

The remainder of this section gives an overview of the Idyll language; more detailed documentation can be read online at https://idyll-lang.github.io/.

4.2 Text

Anything written in an Idyll document is treated as text unless otherwise specified. To make common formatting tasks easier, Idyll borrows some shorthand syntax from Markdown.

- Italic - Text surrounded by a single asterisk (*) or underscore (_) will be *italicized*.
- Bold - Text surrounded by two asterisks (**) or underscores (__) will be **boldface**.
- Headers - Lines starting with a pound symbol (#) are rendered as headings. The number of sequential pound symbols determines the level of the heading.
- Links - links can be rendered with the following syntax: [URL](link text).
- Images - Image can be rendered with the following syntax: ![Image URL](alt text).
- Code - Text placed between backticks (‘) will be displayed inline as code. Text placed between groups of three backticks

will be displayed as a code block. A language can be specified and Idyll will automatically provide syntax highlighting.

```
```js
moveForward(100);
```
```

- Lists - Consecutive lines starting with an asterisk (*) form an unordered list:. Consecutive lines beginning with numbers followed by a period form an ordered list.
- Comments - Text placed after two forward slashes (//) will be removed from the final output.

4.3 Reactive Variables, Datasets

Idyll implements a *reactive variable system*, meaning that users can instantiate variables, and any time the value of one variable changes, any values in the document depending on that variable will immediately be updated as well.

Variables can be declared at any point in an Idyll file. Variables are used to drive the behavior of *components*, which are introduced in the next section. The following code defines a variable `x`, with the initial value of 10.

```
[var name:"x" value:10 /]
```

Derived variables behave similarly to ordinary variables, but their value is tied directly to the other variables and cannot be set directly. Derived variables can be used like formulas in spreadsheet cells.

The following code defines a derived variable `xSquared` that depends on the value of `x`. The value of `xSquared` is automatically updated based on the value of `x`.

```
[derived name:"xSquared" value:`x * x` /]
```

Datasets are similar to variables, except their initial value is populated from the contents of a CSV or JSON file.

```
[data name:"myData" source:"myData.csv" /]
```

4.4 Components

The final constituent of an Idyll document is the *component*. Components provide the mechanism through which interactivity can be added to an Idyll document. Adding a component to the document will cause a corresponding JavaScript component to be instantiated.

Components are denoted with brackets, and can be invoked in one of two ways: they can be self-closing, or have a closing and opening tag surrounding content.

4.4.1 Built-in Components. Idyll includes a variety of basic reusable components. These components may be divided into three categories:

- *Presentation components* render something to the screen. Examples of this type include the `button`, `chart`, `equation`, and `slideshow` components.
- *Layout components* manipulate how content is displayed on the page. For example, the `fixed` component renders its contents fixed to a specific position on the page as a reader scrolls.
- *Helper components* help with behind-the-scenes tasks like adding social media share images and meta tags to the compiled output.

Figure 1: Example Idyll Markup

```
# Article Title
##  Subtitle

Normal body text. **Bold text**.
A self-closing component:
[SelfClosingComponent /]

[OpenComponent]
An open component can modify the content
between its opening and closing tags.
[/OpenComponent]
```

The components that were included with Idyll attempt to cover a wide range of common use cases. They aim to allow users to implement a variety of techniques common in interactive articles without needing to write JavaScript. A full list of the available components is available on the Idyll documentation website.

4.4.2 Third-Party Components. Users may also choose to use components not included in the Idyll standard library. Custom components may either be installed using `npm`, a popular JavaScript package manager, or created locally. Because Idyll is implemented using the React framework [2], any JavaScript modules that functions as a React component will function as an Idyll component. Users may choose to incorporate components from an existing set of tens of thousands of open source options.

4.4.3 Component Properties. Properties may be passed to a component in order to parameterize their behavior. Properties can be passed into components in the following ways:

- *Number*, *string*, and *boolean* literals may be used:

```
[Component propName:10 /]
[Component propName:"propValue" /]
[Component propName:false /]
```

- A *variable or dataset* can be passed directly. If a variable is passed, this will create a two-way binding between the variable and the component. This means that if the variable changes, then the component will immediately reflect that change, and that the component is provided a mechanism to manipulate the value of the variable and affect the document's state.

```
[Component propName:myVar /]
```

- An *expression* may be passed by using backticks.

```
[Component propName:`2 * 2 * 2` /]
```

```
[Component propName:`{ an: "object" }` /]
[Component propName:`2 * myVar` /]
```

All of the document's variables are available in the scope of the expression, so an expression can act as a function of a variable. Because Idyll is reactive, if a variable changes any expressions that reference that variable will immediately be recomputed.
If the property defines an event handler, the expression is evaluated each time the event occurs. This is convenient for updating variables on events, for example.

```
[Component onClick:`myVar++` /]
```

4.5 Scroll Events

Idyll includes two utilities for allowing your document to respond to the reader's active location on the page. Any component that Idyll renders will automatically have access to two events, `onEnterView`, and `onExitView`. If you pass an expression to these handlers, the expression will be evaluated when the component enters or leaves the viewport of the reader.

The following code updates the style of a component when a second component comes into view.

```
[var name:"x" value:0 /]
[Component style:`{opacity: x}` /]
[Component onEnteredView:`x = 1` /]
```

The *Document Object Model* (DOM) is an interface used to provide programmatic access in JavaScript to the elements rendered from an HTML document. Idyll allows users to store references to components' corresponding nodes in the DOM, and access these references in handler expressions. When a reference is created, Idyll performs some basic calculations about the size and position of the component, and where it is on the page relative to the viewport.

To create a reference, one can define the `ref` property on a component. The following code updates the value of a property of the second component based on how far the user has scrolled through the contents of the first.

```
[Component
  ref:"firstComponent" /]
[Component
  propValue:`refs.firstComponent.scrollPropgress.y` /]
```

Each ref object has the following properties[2]:
`domNode`, `scrollProgress`, `size`, and `position`

4.6 Theming and Customization

Idyll exposes two methods for styling the final compiled output. The layout option defines CSS styles that determine how content is laid out on the page: article width, column position, and so on. The theme option allows users to choose from several stylesheets that modify the style of the content itself (text color, font, and so on). Idyll provides several basic defaults of both layouts and themes, but these are fully customizable by users.

Users can also customize the HTML container that the output is rendered into. This is useful for users who wish to provide a branded container that will wrap their content.

[2]See https://idyll-lang.github.io/components-refs for a more detailed explanation.

5 CONCLUSION

We have developed Idyll, a compile-to-the-web language for creating interactive narratives. Idyll was designed according to a set of goals that were distilled from informal conversations with web developers and journalists in addition to experiences working with newsroom graphics production teams.

Idyll combines a human readable markup language with a framework for embedding reactive JavaScript components in-line with text and encodes a set of best practices for creating and deploying interactive stories to the web. A goal of the project is to reduce the amount of code that needs to be written to produce interactive articles. It also aims to enable a closer collaboration between technical and editorial authors, and encourage development of components that can more easily be reused across articles.

The project has been released as free and open source software, and has been downloaded over 10,000 times [3] in the four months between its initial release in April 2017 and the writing of this paper.

REFERENCES

[1] 2008. Jekyll. (2008). Retrieved August 1, 2017 from https://jekyllrb.com/
[2] 2015. React. (2015). Retrieved August 1, 2017 from https://facebook.github.io/react/
[3] 2017. Idyll download count. (2017). Retrieved August 1, 2017 from https://npm-stat.com/charts.html?package=idyll
[4] Eytan Adar, Carolyn Gearig, Ayshwarya Balasubramanian, and Jessica Hullman. 2017. PersaLog: Personalization of News Article Content. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. ACM, New York, NY, USA, 3188–3200. https://doi.org/10.1145/3025453.3025631
[5] Nokome Bentley. 2016. Stencila. (2016). Retrieved August 1, 2017 from https://stenci.la/
[6] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2011). http://vis.stanford.edu/papers/d3
[7] John Gruber. 2004. Markdown. (2004). Retrieved August 1, 2017 from https://daringfireball.net/projects/markdown/syntax/
[8] Amit Kapoor. 2016. Visdown. (2016). Retrieved August 1, 2017 from http://visdown.amitkaps.com/
[9] Chris Olah and Shan Carter. 2017. Research Debt. *Distill*. Retrieved August 1, 2017 from http://distill.pub/2017/research-debt
[10] Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Computing in Science and Engineering* 9, 3 (May 2007), 21–29. https://doi.org/10.1109/MCSE.2007.53
[11] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)* (2017). http://idl.cs.washington.edu/papers/vega-lite
[12] Michael Strickland, Archie Tse, Matthew Ericson, and Tom Giratikanon. 2015. Archie Markup Language (ArchieML). (2015). Retrieved August 1, 2017 from http://archieml.org/
[13] NPR Visuals Team. 2014. Pym.js. (2014). Retrieved August 1, 2017 from http://blog.apps.npr.org/pym.js/
[14] Bret Victor. 2006. Magic Ink: Information Software and the Graphical Interface. Retrieved August 1, 2017 from http://worrydream.com/MagicInk/
[15] Bret Victor. 2011. Explorable Explorations. Retrieved August 1, 2017 from http://worrydream.com/ExplorableExplanations/
[16] Bret Victor. 2011. Scientific Communication As Sequential Art. http://worrydream.com/ScientificCommunicationAsSequentialArt/.
[17] Bret Victor. 2011. Tangle: a JavaScript library for reactive documents. Retrieved August 1, 2017 from http://worrydream.com/Tangle/