



Graduation thesis submitted in partial fulfilment of the requirements for the degree of  
Master of Science in Applied Sciences and Engineering: Computer Science

## **INTERTEXT**

### **The Everything App**

**OGUZ GELAL**

**Academic year 2020–2021**

Promoter: Prof. Dr. Beat Signer

Advisor: Prof. Dr. Beat Signer

Faculty of Sciences and Bio-Engineering Sciences



## Abstract

Today, we often see the same repeating patterns in front-end systems, from the components that make up the user interface to features and practices such as navigation, routing and state management. It is not uncommon for these patterns to be re-implemented time and again in various shapes or forms. Due to the lack of a systematic enforcement mechanism in the open market, these implementations' correctness and completeness rely primarily on the developer. A proper online representation nowadays requires accessible front-end applications with best practices optimised for various devices, browsers, screens and platforms, often resulting in high costs or inconsistent, insecure and low-quality byproducts. From a users perspective, this entails a poor user experience and bring potential security and privacy concerns.

To address these problems, we propose Intertext, a novel approach to how front-end systems are developed and consumed. It consists of Intertext User Interface Description Language (IUIDL): a device, design and style agnostic XML-based markup language; and a family of software clients for web, mobile, desktop and various other platforms that can render IUIDL into fully functional front-ends. IUIDL provides essential building blocks, such as a layout system, User Interface (UI) components and commands. It incorporates both input and output components, which can be used in conjunction with commands to make applications interactive. Commands enable sending, receiving and handling data, managing application state, routing and much more. It supports interpolations and views derived from the application state. IUIDL can be assembled and served from a generic backend, where the business logic of the application shall live. Once served from an endpoint, users can access it through any Intertext client on any device or platform in a similar fashion to an internet browser. Clients receive and render IUIDL most appropriately based on the host device and platform. It allows users to browse all their data sources through the same familiar, stable, robust, accessible screen/device optimised interface. IUIDL is agnostic of style, so users can customise the application's look and feel to their liking. Most importantly, clients do not accept executable code from external sources and all interactions with the device and external servers are controlled, which guarantees safety and privacy to the users. Intertext aims to stand as an alternative to traditional front-end systems that significantly benefits both the user and the data provider.

## Acknowledgements

# Contents

## 1 Introduction

1.1	Problem Statement . . . . .	2
1.1.1	User Problems . . . . .	2
1.1.2	Developer Problems . . . . .	4
1.2	Research Questions . . . . .	5
1.3	Contributions . . . . .	5
1.4	Methodology . . . . .	6
1.5	Structure . . . . .	7

## 2 Related Work

2.1	UIDLs . . . . .	9
2.2	Frameworks and Libraries . . . . .	12
2.3	Tools and Services . . . . .	12

## 3 Solution

3.1	Design Principles . . . . .	14
3.1.1	No Foreign Code Execution . . . . .	14
3.1.2	Transparency . . . . .	15
3.1.3	Black-box Components . . . . .	15
3.1.4	Shared Syntax . . . . .	18
3.2	Intertext UIDL . . . . .	18
3.2.1	Styling . . . . .	21
3.2.2	Syntax . . . . .	21
3.2.3	Terminology . . . . .	22
3.3	Intertext Clients . . . . .	24
3.3.1	Components . . . . .	25
3.3.2	Commands . . . . .	25
3.3.3	State Management . . . . .	26

## 4 Implementation

4.1	Engine . . . . .	31
4.2	Layout . . . . .	32

---

4.2.1	Layout System . . . . .	33
4.2.2	Unit System . . . . .	35
4.2.3	Properties . . . . .	35
4.3	Components . . . . .	36
4.3.1	Block . . . . .	36
4.3.2	Grid . . . . .	37
4.3.3	Typography . . . . .	37
4.3.4	Literals . . . . .	39
4.3.5	Collapse . . . . .	39
4.3.6	Button . . . . .	40
4.3.7	Input . . . . .	41
4.4	Commands . . . . .	41
4.4.1	State . . . . .	41
4.4.2	Alert . . . . .	41
<b>5</b>	<b>Evaluation</b>	
<b>6</b>	<b>Conclusion</b>	
<b>7</b>	<b>Discussion</b>	
7.1	Future Work . . . . .	47
<b>A</b>	<b>Appendix</b>	

# 1

## Introduction

In recent decades, the rise of smart interconnected devices not only changed how we interact with information, but also introduced all-new ways of engaging with the world. The rise of smartphones enabled users to consume content and perform tasks on-the-go, tablets made people rethink the ways they work and travel, wearable devices made new sets of data available to the users and so on. Ericsson Mobility Report of February 2021 reveals that the number of connected devices has been steadily increasing [6], and with new devices and form-factors underway, such as smart glasses, smart vehicles and folding displays, this trend is likely to continue in the foreseeable future.

The increasing demand in the diversity of devices and device families also increased the demand for front-end application development, as it became more challenging and costly to have a decent user-facing online presence. This led to advancements in front-end development; emerging new tools, techniques, frameworks, libraries, and SDKs made it possible to build consumer-facing products in ways that were never possible before. Nevertheless, the ever-growing front-end ecosystem and high user demands forced developers to spread their focus to overwhelming levels; a recent survey *State of JavaScript* reveals that around **TODO: This is wrong ->** 60% of developers use more than one Front-end framework, and for cross-platform frameworks, it is more than **TODO: This is wrong ->** 80% [15] (Fig 1.1). This transformation did help create a large and dynamic market of consumer products,

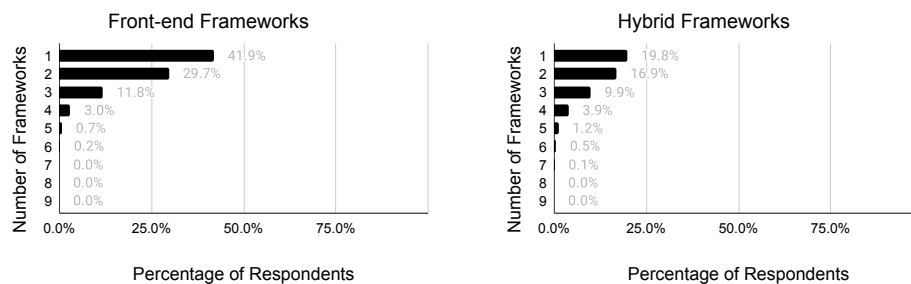


Figure 1.1: Number of frameworks used to Percentage of respondents

goods and services, though it did not come without some hurdles. In this thesis, we explore what these hurdles are and discuss how Intertext addresses them. subfigure env

## 1.1 Problem Statement

In this section, we focus on these hurdles from the users and developers perspectives:

### 1.1.1 User Problems

The simplicity of consuming data is lost within the complexity of modern-day applications; nowadays, something as simple as checking the weather, reading a news article, browsing an image gallery, buying a product or service and filling out a form can be frustrating and time-consuming. We have identified the fundamental problems that lead to user experience problems and compromise privacy/security.

### UI / UX Inconsistencies

One reason for this is the inconsistencies in front-end implementations. The visual presentation of components with the same functionality can often differ across front-end implementations, often confusing the end-users. Inconsistencies are often seen not only within a platform or cross-platform, but even a single front-end application can have inconsistencies within itself. This can arise from a variety of different reasons ranging from poor design to poor implementation.



**Intrusive Advertisement**

Another common source of frustration for the users is intrusive online advertisement and numerous other attention-grabbing call-to-action popups that diverts users attention from the content they are consuming. A recent study by Yahoo [17] states that as online advertisement became the primary source of income for most products, it has become increasingly intrusive and annoying even at the cost of hurting the user experience.

**Lack of Accessibility**

Due to its high costs of implementation and maintenance, accessibility is often overlooked by developers. A study reveals that on an average webpage, only 3.89% of the HTML elements were found to be fully accessible [9]. This paints a picture of how accessibility is a common problem among users.

**Lack of Customisability**

For many front-end applications, customisability is rarely a concern. The way of front-end ecosystem is, the developer decides on the look and feel of an application rather than the user. Although a recent trend in web design, dark mode support in front-end design [5] could be seen as a shift towards more customisability. However, very few front-end applications provide such features.

**Lack of Cross-platform Support**

Every now and then, we find ourselves in situations where we are trying to operate a web application with no responsiveness or touch support from our phone—or having to stand up from in front of our computer to reach for our phone because the messaging application we want to use is only available for web/desktop. Hardships in cross-platform application development often result in poor user experience due to compatibility issues and sometimes complete lack of availability.

**Privacy and Security Concerns**

In the open market, where everyone can create user-facing applications, there is little to no enforcement mechanism or quality control measures. At times, this results in a compromise in users security and privacy. Users, especially of old age, can be lead to downloading and executing harmful software. With-

out alerting the user, a foreign script can be executed on the users' computer/browser, leading to severe security and privacy concerns.

### 1.1.2 Developer Problems

The complexity of building a decent, well designed, accessible front-end experience, not even once but once per every client built for various platforms, forces developers to choose between supporting multiple devices, following best practices, creating a good experience.

#### Challenges in Development

1. Cross-platform support: Cross-platform application development is a prominent topic even to this day. Even though several development techniques exists [10], building applications capable of running on multiple platforms is still a complex engineering challenge, and creating and maintaining multiple applications for different platforms is costly.
2. Accessibility support: Creating fully accessible applications is not always the priority for many development projects due to its costs and efforts. Accessibility implementations are often complicated as there are many different ways of creating accessible user interfaces for many different kinds of accessibility needs. A study shows that accessibility and complexity of web applications are reversely proportional, hinting that making a web page accessible hinders maintainability [9].

#### Challenges in Maintenance and Enhancements

Building a front-end application is a great effort, but maintaining and enhancing it over time could be even more troublesome. The world of front-end development is a fast-growing and evolving ecosystem. The changes are persistent, and at times breaking. There are thousands of tools, libraries, frameworks, SDKs available at the fingertips of developers at no cost. It is a common practice to make use of these libraries as they help with the development significantly. However, the diversity in the libraries used to build the software results in a diversity of maintenance problems. Even the most well-tested and maintained libraries could break after an update, causing a headache for the developers and hardship for the users.

## 1.2 Research Questions

We have listed the problems that we are interested to solve in the section above. In light of these problems, we aim to answer the following questions in our research:

**How can we re-imagine front-end, and improve it in such a way that:**

1. from the users perspective, for all applications it:
  - (a) presents a consistent User Experience (UX) ?
  - (b) works consistently on all supported devices and platforms ?
  - (c) allow users to customize the look and feel ?
  - (d) guarantees accessibility ?
  - (e) guarantees privacy ?
  - (f) guarantees security ?
2. from the data-providers perspective, it:
  - (a) eliminates the need to create visually-appealing front-end applications ?
  - (b) eliminates the need to create accessible front-end applications ?
  - (c) eliminates the need to create front-end applications for every device / platform ?

**How can we create an alternate front-end realm that:**

1. brings product development costs to a minimum?
2. enables more variability of software products in the market?
3. unifies the experience, so individual developers and large companies compete on the same ground?

## 1.3 Contributions

As discussed in the Problem Statement (1.1) section, Intertext ambitiously attempts to solve many different problems from various domains. A solution at this scale requires novelty, rethinking the current state of art instead of an incremental update. The biggest contribution of Intertext is to borrow existing concepts that are mostly academic, combine them in new ways, and

apply them to solve these real-world problems. With that said, here are some notable contributions:

- We reviewed existing research on User Interface Description Languages (UIDLs) and other similar topics to outline ways to utilise these concepts to solve the aforementioned problems.
- We designed IUIDL; while doing so, we decided:
  - What are the concepts that are crucial to modern front-end development that needs support out-of-the-box
  - What terminology should we use in order to create an optimal balance between the device-independent nature and developer familiarity
  - What the syntax should be like to maximise developer friendliness
  - How to overcome some limitation of XML in the most effective, clear and extendable way
- We created an engine using Javascript / Typescript to perform common tasks of Intertext clients, such as parsing IUIDL and managing the application state
- We created multiple software clients that can render IUIDL into functional front-ends optimised for the host device and are user customisable where possible

## 1.4 Methodology

To identify and effectively address all these problems, we utilized the Design Science Research Methodology (DSRM) [13] to carry out our research. DSRM is a commonly adopted research framework that provides six executable steps to guide the researcher in being consistent with prior work and provides a theoretical process for doing the research, and guides through presenting and evaluating the outcome.

The first step of DSRM is **identifying the problem and the motivation**. It is not uncommon for research and inventions to stem from a problem or a necessity. Similarly, in our case, the problem was out there; we only needed to focus on it to better understand the issues we are tackling. In the Problem Statement (1.1) section, we discussed and justified what problems are we attempting to solve with Intertext. Then we proceeded to the next step, **defining the objectives for a solution**. As we narrowed down the problems

that we attempt to solve in the Problem Statement (1.1) section, we drafted our objectives to approach this problem. We explained these objectives in detail in the Research Questions (1.2) section. Then, we were ready to take the next step of **design and development**. In the following sections **TODO: add sections**, we explained in detail our development process, all the challenges and design decisions that we have taken in order to address the problems in the best way possible.

Once we had a working prototype in our hands, we created a simple backend application that uses Intertext as its front-end. This sample application served two purposes; it demonstrated how IUIDL could be dynamically served from a backend, and it was used for the user evaluation. Consequently, we used this sample application to introduce Intertext to the users, communicate its goals and motivations, and then collect feedback. This last step combines the final three steps of **demonstration**, **evaluation** and **communication** into one, which is explained in details in **TODO: add section** section.

## 1.5 Structure

**TODO: add structure**



# 2

## Related Work

*the field of ...*  
This section explores and discusses the state-of-the-art research, tools, and technologies in ~~this field or related to this field~~. First, we focus on existing UIDLs and how they compare to Intertext. We categorise them to demonstrate the relevancy as coherently as possible and investigate <sup>as</sup> ~~ones~~ <sup>2 the</sup> that fall into the same category <sup>as</sup> ~~with~~ Intertext on a case-by-case basis. Then, we look into libraries and frameworks that aim to achieve a similar goal <sup>with as</sup> ~~with~~ Intertext. We will explain how Intertext is extending the existing solutions to do what it does. Moreover, we will explore commercially available tools and services and discuss why they are relevant to Intertext and what is to be learned from their efforts and success.

### 2.1 UIDLs

*→ did you mention the full name earlier?*  
A UIDL can be defined as having two separate parts: a syntax that describes the user interface characteristics, and semantics that defines what these characteristics mean. They share the common goal of describing a UI without targeting any particular programming language or platform; nevertheless, the end-goal of UIDLs often varies [16]. UIDLs typically use XML or a similar markup language/notation that later gets transpiled into a programming language or is processed by a software to automatically or semi-automatically be translated into a UI, a visualisation, or any byproduct depending on the

goal of the project. A UIDL can be thought of as a tool designed to achieve a particular goal or to achieve a common goal in a particular way.

Souchon, Nathalie and Vanderdonck et al., 2003 [16], and later Guerrero-Garcia, Josefina and Gonzalez-Calleros et al., 2009 [7] argued "there is a plethora of UIDLs that are widely used, with different goals and different strengths". There are many ways of classifying and categorising existing UIDLs, as can be seen in *A review of XML-compliant user interface description languages, 2003* [16], and *A theoretical survey of user interface description languages: Preliminary results, 2009* [7]. However, in this section, to stay relevant to the comparison to Intertext, we group UIDLs under two main categories: compiled and interpreted UIDLs. Most of them belong to the first category, allowing us to focus on the general picture instead of on a case-by-case basis. This separation narrows down the comparison to a select few UIDLs, which we investigate in more detail.

### Compiled UIDLs

Compiled refers to cases where UI descriptions are created at design time and are used to generate code or the final UI for different target platforms and environments. Most of the known UIDLs falls under this category. They often rely on a transformational approach; they utilise a reference framework for classifying UI elements with multiple levels of abstraction, and reify them into more concrete levels based on the target platform and context of use. UIML [4], XIML [14], TeresaXML [11], MariaXML [12] and UsiXML [8] are several examples of UIDLs that uses this model.

### Interpreted UIDLs

TODO: BOSS? TODO: SeescoaXML TODO: OpenUIDL?

### Comparison

We explored several of the existing UIDLs in this section. Some of them show similarities with Intertext in some respect, whereas others are radically different. The differences between many of them include but are not limited to:

1. Intertexts focuses on graphical user interfaces (GUIs), while many UIDLs try to cover different kinds of UIs. TODO: narrow down the mentions to focus on GUIs early on in the paper?

repetitive

still to be done? just listy then is not really explanatory



2. Intertext has a single level of abstraction, unlike many of the other UIDLs with multiple levels of abstraction, mostly due to their wider focus
3. Intertexts' IUIDL is interpreted. It relies on reifying the single level of its abstraction into the final UI on-the-fly, whereas the final UI for the UIDLs in the first category is generated during compile time.
4. The UI descriptions are created manually during design time for many UIDLs (or, in some cases, generated by several different means). However, for Intertext, it is meant to be generated and served on-the-fly from an endpoint based on the application logic and UI state.
5. Unlike Intertext, most UIDLs with a model-based approach incorporate heavy user interactions.

Nonetheless, if we were to take a step back to look at the big picture, Intertext has a fundamental difference that separates itself from the others; it is the purpose. The value added by the UIDLs is to improve the development process of user interfaces, reduce the cost and effort of creating UI descriptions that can target multiple platforms and environments with minimal to none additional effort. The byproduct of these UIDLs is a functioning user interface on each target platform. While we aim to take advantage of the nature of UIDLs to enable some of these advancements, we also have an equal focus on the user and improving the user experience. Unlike other UIDLs that double down on creating the best UI development experience and producing the most comprehensive outcome possible, we intentionally introduce calculated limitations to benefit users equally, if not more. Although we provide more details on Intertext clients and IUIDL in section **TODO: add section**, we can summarise our efforts that distinguish Intertext from other UIDLs as below:

1. Intertext is style-agnostic so that users can customise the look-and-feel of the UIs to their liking.
2. The final UI rendered for each platform and environment cannot be altered to ensure the quality and accessibility of the UI elements.
3. Intertext UIs are meant to be solely presentational with minimal user interaction for security reasons.
4. Intertext, in general, is a platform that comes with IUIDL and software clients that can interpret and render IUIDL. It has no intention of generating a UI that can run independently.

## **2.2 Frameworks and Libraries**

## **2.3 Tools and Services**

# 3

## Solution

Intertext is a platform based on a straightforward premise; it is a family of applications that can interpret IUIDL, an XML based UI description language, and generate appropriate front-ends for the host platform on the fly. Simply put, a developer wanting to create a front-end application uses a generic backend to generate IUIDL, and serve it from an endpoint, say at <https://intertext.example.com>. Users who wish to use this application use an Intertext client on their preferred device or environment and visit this domain just like in a web browser. The Intertext client then makes a request to this domain, fetches the IUIDL served by this endpoint and generates the user interface <sup>based on the</sup> as per the instructions received via IUIDL. (Fig 3.1) <sup>as illustrated in Figure 3.1.</sup>

Rather than rendering a simple static view, <sup>Intertext</sup> it performs some tasks such as accepting user input, navigating to different screens, making additional requests to fetch more data, keeping the UI updated and reading and writing some data to <sup>users</sup> local storage; all of which is again orchestrated based on the instructions received by the backend in IUIDL syntax. <sup>currently</sup>

Intertext aims to support multiple software clients (<sup>at the time of this paper,</sup> web and command-line interface clients are implemented, <sup>with more on the way</sup>) built natively for various platforms that can interpret IUIDL most appropriately to the host device or platform. For instance, users browsing an Intertext app through a smartphone receives an experience optimized for touch screens, command-line interface client users receive an optimized expe-

or text/html  
or browser  
code

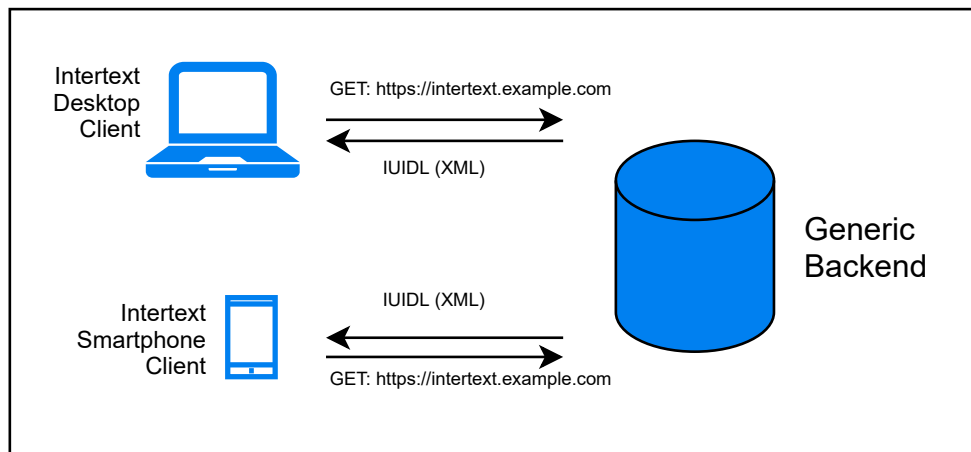


Figure 3.1: A diagram showing the general workings of Intertext

rience for a text-based interface or a user browsing from a low-end device with limited capabilities use the version optimized for low-performance devices to get a comfortable viewing experience and so on.

## 3.1 Design Principles

*problem statement*

In order to address the problems mentioned in the ~~Problem Statement (1.1) section~~, we adopted and implemented the following design principles.

### 3.1.1 No Foreign Code Execution

*if when* There is no foolproof way of entirely securing a piece of software; even the most mature platforms and operating systems sometimes end up vulnerable to security exploits. However, without the ability to execute code on a platform, it is not directly possible to take advantage of vulnerabilities, even ~~when~~ there is one. Furthermore, the nature of Intertext allowed us to adopt disallowing code execution as a design principle. Intertext clients only accept UIIDL code, which is in XML and is not executable. Should a server send anything else, Intertext clients will simply ignore it. Thus, we can guarantee security for the users, regardless of the platform they are using the Intertext client on.

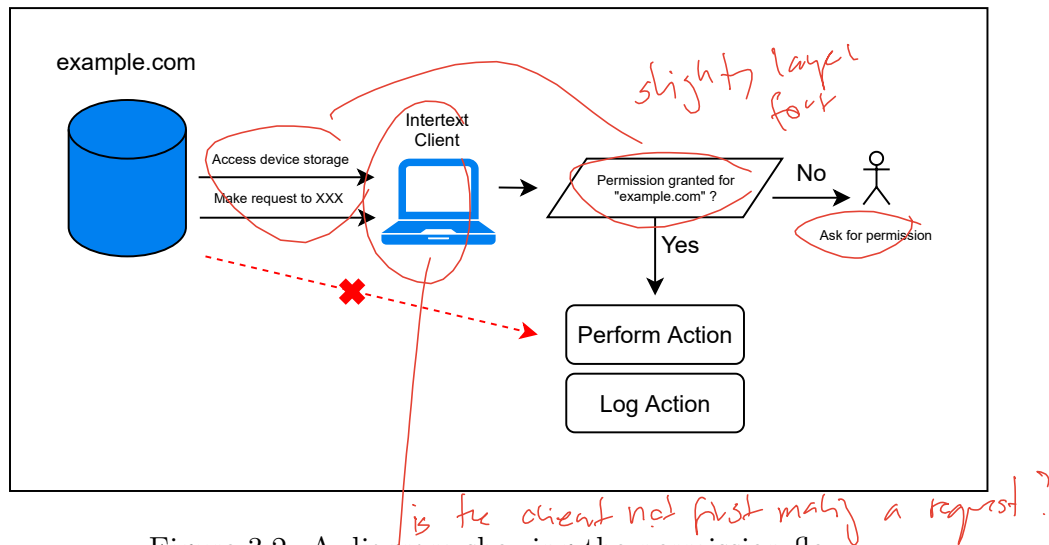


Figure 3.2: A diagram showing the permission flow

### 3.1.2 Transparency

Privacy is a common concern among users; primarily due to the recent scandals and data leaks, people started getting more conscious about their data. There is an increasing demand for users to be more in control and be aware of what is exposed and what is not. In order to address this burning need, we adopted transparency as a design principle.

The most prominent way of achieving this is to have Intertext clients ~~be~~ in control of all interactions with the device and with external sources and keep logs in order to make them transparent to the user. The above-mentioned principle that no foreign code will be executed goes hand-in-hand with achieving this, as it would be unrealistic to expect complete control as it would be a non-deterministic approach. A good analogy would be to think of Intertext clients as an API endpoint that runs on users devices and exposes fundamental interactions with the host device and external networks through an API in a fully controlled manner. An application could "instruct" the Intertext client via IUIDL to perform some actions, such as making a network request, storing data, and accessing the local storage. Intertext client will then block this action until the user grant permission and keep logs every time before performing an action (Fig 3.2)

as show in Figure 3.2.

### 3.1.3 Blackbox Components

Intertext adopts a component-based approach, we provide a set of components for developers to use to build their applications with. Each component

```

<h3>Buttons</h3>

<grid cols="[1,1]">
  <block>
    <button marginBottom="2">default</button>
    <button marginBottom="2" disabled="true">disabled</button>
  </block>
  <block>
    <button marginBottom="2" intent="default">default</button>
    <button marginBottom="2" intent="primary">primary</button>
    <button marginBottom="2" intent="error">error</button>
    <button marginBottom="2" intent="warning">warning</button>
    <button marginBottom="2" intent="success">success</button>
    <button marginBottom="2" intent="info">info</button>
  </block>
</grid>

```

Figure 3.3: UIUDL that renders buttons in several states

accepts a set of properties, which gives them certain functionality or appearance. Developers are to use these components through UIUDL. For example, the UIUDL code below (at Figure 3.3) and its output for Intertext web version (at Figure 3.4) shows some of the properties that *button* component accepts. Layout properties such as *marginBottom* allows positioning of the buttons to be customised, *intent* properties gives the button a different look based on the use case, properties like *disabled* can alter its behaviour and so on.

However other than properties that components accept, they cannot be modified or altered by the developer. The main motive behind this principle is standardisation. When all Intertext applications use the same set of components, we can control how they look and how they are implemented, and ensure certain behavioural and visual aspects to bring the benefits in order to solve some of the problems mentioned in the *Problem Statement* section.

Consistency is one of the most important benefit that this principle enables. A standard look and feel for components helps users to never get disoriented across applications, and to recognise similar patterns easily. Another benefit is customisability, it is only possible to create one-size-fits-all themes that applies to every component in every application when all the applications use components that are built in the same way. Another thing

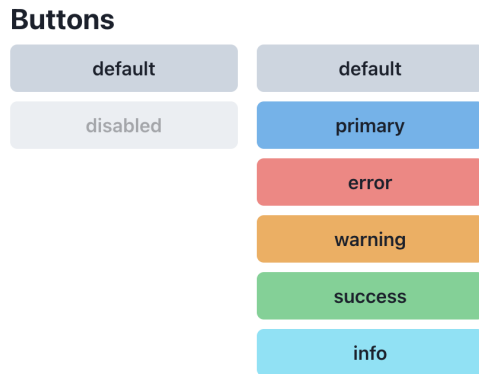


Figure 3.4: Output of IUIDL at Figure 3.3 on Intertext web client

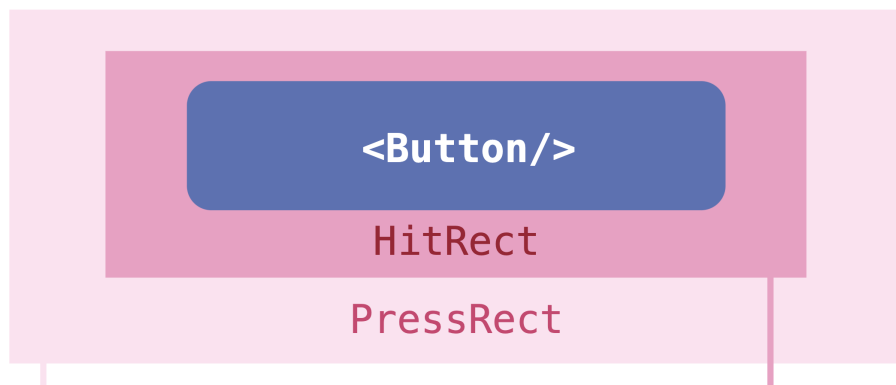


Figure 3.5: Diagram that shows the implementation of Pressable component in React Native (reactnative.dev/docs/pressable) *fairly!*

that comes to mind is accessibility. We established in section 1.1 that accessibility is a big issue that not all developers choose to address. This principle allows us to take this responsibility from the developers' hands by providing accessible components as building blocks.

Last but not least, this principle was crucial to achieve cross-device compatibility. The appearance and behaviour of components differs between implementations of the Intertext clients. For instance, the button component on the web version needs not to be too big as clicks are precise, it requires a hover and focus state and so on. For a touch interface however, it needs to be bigger, and handle the caveat of having less precision by accepting hits on an area around it as seen in figure 3.5. Every feature that components have needs to be supported as much as possible on different platforms that

*code!*

has different requirements. Having components with a fixed set of features allowed us to implement them all for different Intertext clients.

### 3.1.4 Shared Syntax

We designed IUIDL to be a generic markup language, agnostic of any platform or interaction type, and be based purely on XML so it could be consumed by all clients. This approach has many advantages, both for developers and users. It allows developers to create universal applications; once they start serving an Intertext application through an endpoint, any Intertext endpoint can consume the application through that endpoint. Moreover, it helps create a continual experience for the user; given that there is a shared layout system, UI elements will look and feel the same between Intertext clients. Figure **TODO** and **TODO** shows the similarities between Intertext web client and the command-line client.

**TODO Add screenshots that shows similarities between the web version and the cli version**

Also, this approach allows existing Intertext applications to adopt to new Intertext clients as they are built in the future. At the time of this paper, Intertext web client and command-line client is ready, but as mentioned in the Future Work section, more Intertext clients are on the way. Thanks to this principle, Intertext clients will have immediate availability for the upcoming Intertext clients.

## 3.2 Intertext UIDL

Intertext UIDL (IUIDL) is an XML-based markup language that can be used to describe user interfaces. It features a layout system and UI components that could be used as building blocks to put together a user interface. IUIDL is meant to be served from a generic backend, assembled on the backend based on the application logic. It is designed to be unopinionated, as it doesn't restrict how you assemble it or serve it, or where you serve it from. For instance, below is an example To-do list UI served in IUIDL from a Node.js server that uses Express.js framework, as it can be seen in Figure 3.6. Once hit, the endpoint `/todo` passes the To-do item data to a template file which uses Handlebar as a template engine, and it renders the UI in IUIDL format. The output XML is served from the endpoint.

*code*



```
router.get('/todo', function(req, res, next) {

  const todos = [
    {
      title: 'Buy milk',
      done: false,
    },
    {
      title: 'Call mom',
      done: false,
    },
    {
      title: 'Prepare for presentation',
      done: true,
    },
  ];

  res.render('todo', {
    itemsToDo: todos.filter(item => !item.done),
    itemsDone: todos.filter(item => item.done),
  });
});
```

Figure 3.6: A simple Express endpoint that serves the render output of the Handlebars template at Figure 3.7

*shown in*

```

<h3>To do ({{ itemsToDo.length }})</h3>

{{#each itemsToDo}}
  <block intent="default" flexDirection="row" alignItems="
    center" paddingLeft="4">
    <text flexGrow="1">{{this.title}}</text>
    <button intent="error">Remove</button>
    <button intent="success" marginLeft="2">Done</button>
  </block>
{{/each}}

<h3>Done ({{ itemsDone.length }})</h3>

{{#each itemsDone}}
  <block intent="default" flexDirection="row" alignItems="
    center" paddingLeft="4">
    <text flexGrow="1">{{this.title}}</text>
    <button intent="error">Remove</button>
  </block>
{{/each}}

```

Figure 3.7: Handlebars template that renders To-do items in IUIDL format



Figure 3.8: IUIDL served from the endpoint at Figure 3.6 rendered on Intertext web client

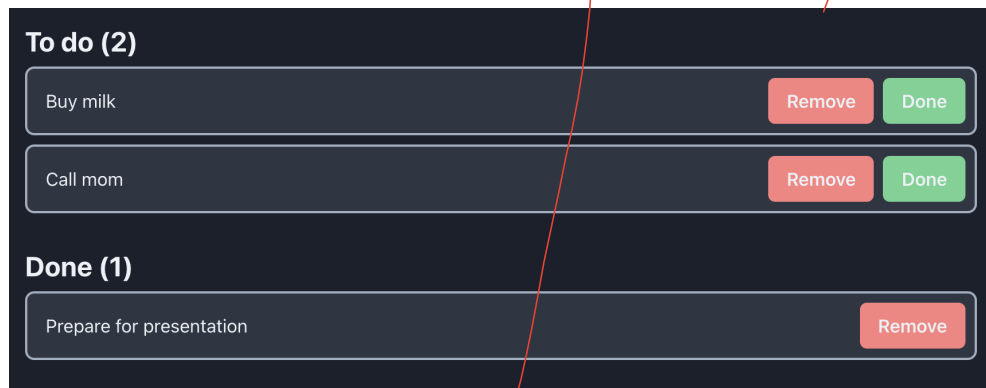


Figure 3.9: Dark version of the to-do list UI from Figure 3.8 rendered on Intertext web client

### 3.2.1 Styling

IUIDL is agnostic of the styling. There are several *intent*'s that components can accept, which renders the component most appropriately based on its use-case. For instance, the "Remove" buttons in Figure 3.8 as can be seen is red, and "Done" button is green. That is because *error* intent was given to the Remove button and *success* was given to the Done button. However, this configuration, there is no way to specify how the components look like in particular. The main idea is customisability. Intertext provides themes that users can choose from, and for each theme UI components look and feels differently. As of this thesis, only light and dark themes are available for the web version of Intertext (the dark version of the To do list UI can be seen on Figure 3.9), but as mentioned in the Future Work section, more themes will be made available. Moreover, users will also be able to create custom themes that matches exactly to their liking.

### 3.2.2 Syntax

As explained in detail in the Implementation section, IUIDL gets converted from XML syntax to JSON syntax. However, the conversion to JSON is only an implementation detail that takes place during the rendering phase, and developers ~~need not~~ <sup>do not need</sup> to be aware. XML was chosen at the cost of creating an additional transpilation layer, to achieve optimum developer friendliness.

When it comes to building UI's, XML and XML-like languages are very common. The main benefit of such languages <sup>is</sup> ~~are~~ that they are easy to read and write, while at the same time they are easy to parse and process as data. IUIDL is meant to be served from a backend, but developers still need

```

<block intent="error">
  <h3 intent="error">Not Found</h3>
  <collapse>
    <collapse.handle>
      <text intent="error">
        Show Details
      </text>
    </collapse.handle>
    <p intent="error">... stack trace here ...</p>
  </collapse>
</block>

```

Figure 3.10: The XML representation of the 404 page

to write IUIDL code by hand on the server side. JSON is far from being practical to write in large by hand. The difference could be observed in Figure 3.11 and 3.10, the IUIDL code for a custom 404 page could be seen in both XML and JSON syntax.

One issue that raised from XML syntax was the inability to assign complex children to attributes of a specific tag. As it can be seen from the JSON representation of the 404 page at Figure 3.11, the component *collapse* requires two sets of children: one for its direct children (specified as *collapse*), and one for the children to be rendered in its handle (specified as *handle*). Both *collapse* and *handle* properties takes other UI components as their children. However, XML syntax only allows attribute values of a tag to be of type string. In order to solve this problem, we extended XML to create a special syntax to handle such cases. We introduced a special tag name convention: when an XML tree is placed as the children of a component with the tag name being the parent components name concatenated with an attribute name joined together with a dot ("."), its gets interpreted as an attribute of the parent, with its value being the children of that XML tree (Figure 3.12).

### 3.2.3 Terminology

A Last thing to mention in this section is the terminology used in IUIDL. IUIDL syntax is shared between multiple devices and environments, which includes desktop interfaces as well as touch interfaces, non-graphical user interfaces and more. As mentioned in the Future Work section, as more Intertext clients gets released, this variety will increase further. This brings up a problem with

code  
 so not really XML anymore!

```
[
  {
    "block": [
      {
        "h3": "Not Found",
        "intent": "error"
      },
      {
        "collapse": [
          {
            "p": "... stack trace here ...",
            "intent": "error"
          }
        ],
        "handle": [
          {
            "text": "\n Show Details\n ",
            "intent": "error"
          }
        ]
      }
    ],
    "intent": "error"
  }
]
```

Figure 3.11: The JSON representation of the 404 page

```
<parent>
  <parent.complex>
    <text>
      This is under 'complex' attribute of parent
    </text>
  </parent.complex>
  <text>
    This is the real children of parent
  </text>
</parent>
```

*why attribute?*

Figure 3.12: Complex attributes

the terminology. For instance, the "click" that normally would make sense for a desktop environment does not make sense for touch interfaces, as the interaction for a touch screen interface would be "touch" or "tap". In the case of a command line, the primary interaction is focusing on an item and hitting the "enter" key.

Our immediate reaction to solve this problem was to create custom jargon that is agnostic of device / interaction type, just like IUIDL is by nature, and find terms for every component / interaction that applies globally. For instance, instead of "button" we used "callToAction", and instead of "onClick" we used "onPrimaryInteraction". However we later abandoned this approach for the sake of developer friendliness. As we custom-named components and their respective actions, we realised that their names were getting very unusual and non-familiar, to a point where it was very difficult to even recognise them. Should we have chosen the global naming convention, we would have introduced a steep learning curve; developers would have needed to get familiar with a long list of terms that they have never heard of before, and they would strictly need to follow the documentation while building Intertext applications.

Instead, we decided to adopt the terminology for desktop / web environments. The rationale behind this decision was that these environments have been around for a very long time, that the terms used to build desktop applications and web applications has a strong recognition among many developers. Also, terms used for desktop environments were mostly generic enough that it was easy to represent on other environments. For instance, a "button" is a component to be interacted with that performs an action, and "click" is the method of primary interaction with that component. We can easily take this as a given to replicate it on any environment and create an optimised version taking the limitations of the host platforms in mind.

### 3.3 Intertext Clients

Intertext clients are the user facing products of Intertext, that build the bridge between the user and servers that serve IUIDL. They are meant to be implemented natively for the host platform in the most optimal way possible.

For example, on mobile platforms such as iOS and Android, UI elements would be larger and more suitable for touch interactions. Primary interaction would be translated as "tap" while secondary interactions, if any, would be translated as "tap and hold". The layout would adjust itself based on the screen size. The client would be implemented with either native technologies such as Swift/Objective-C for iOS and Kotlin/Java for Android, or with

hybrid technologies that gets compiled into native code, such as React Native or Flutter for maximum responsiveness and efficiency.

At the time of this paper, a web client and a command-line client is implemented. Native clients for iOS and Android, and desktop clients for Mac OS, Windows and several Linux distributions are among the ones that are planned, more could be found in Future Work section.

Intertext clients consists of two main parts; components and commands. Each Intertext client implement these separately, based on the host platform and its capabilities. Each client also implements the their own local storage, state management strategy, and communication technique with the server.

### 3.3.1 Components

Components are anything that has a visual representation. All component stem from a generic IUIDL definition, and is interpreted based on the host platform requirements. They can be organisational or presentational. Organisational components are the layout components, they are used to position presentational components on the screen. Intertext implements a version of css flexbox specification as a layout system, which could be used to implement responsive interfaces for all screen sizes. For non-browser-based platforms, we use the popular Yoga layout engine, which is a standalone implementation of the css flexbox specification in C++, allowing us to target almost any platform. More details on the layout system is given in Implementation section.

Most presentational Intertext components share a concept, "intent", which is a way to communicate the intention of the UI element based on its use-case. Intertext clients renders the UI elements differently based on its intent, allowing a way for the developer to convey an intention of an UI element to the user without being able to intercept with the styles. For instance, an error message could be shown in a block component that has the intent "error", and it will be rendered with a red background, indicating to the user that it is an error message. 3.13 shows an example of intents on block component for Intertext web client.

### 3.3.2 Commands

Commands are IUIDL statements that are used to instruct Intertext client to take an action. These actions can be triggered upon interaction with some components (such as clicking a button or submitting a form), a life-cycle event, with a timeout, or on page load. Commands are very primitive by design, they are not meant to build application logic with. They are merely

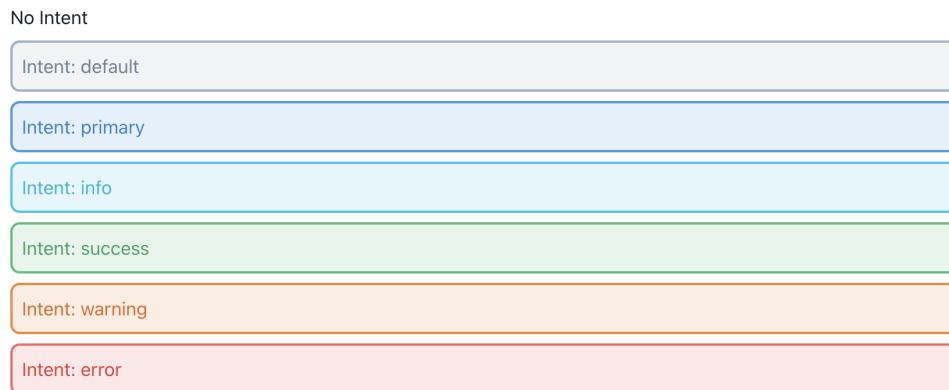


Figure 3.13: Intents for Block components

```
<timeout delay="1000">  
  <request endpoint="/refresh"></request>  
</timeout>
```

Figure 3.14: Post command on Timeout

for asking the Intertext client to communicate with the server, store something, read something that is stored, or retrieve some user data. Command system is designed in order to ensure Intertext client is aware of what it is being asked to do, so that it can control and restrict the entire flow, ask for permission from the user when necessary, or simply refuse to take an action if needed.

Figure 3.14 shows an example of how a network request action can be executed upon a delay, and Figure 3.15 shows how a form can be submitted with reference to input fields. More on commands will be on Implementation section.

### 3.3.3 State Management

Intertext clients offer basic state management, making it possible to serve stateful Intertext applications using serverless/stateless backends. Front-end state can be fully driven from the backend via IUIDL. Intertext client offers two types of storage, persisted and volatile.

Volatile storage is kept in the memory and it is persisted across the session. The purpose of this storage is to keep temporary values, such as users' progress in a long form distributed across multiple pages. Intertext clients



```

<input name="item_title"></input>
<input name="item_description"></input>
<button>
  <text>Submit</text>
  <button.onClick>
    <request endpoint="/items/save"></request>
  </button.onClick>
</button>

```

Figure 3.15: Form command with reference on server side to internal input values

passes the entire application state on the volatile storage to the backend on every request, allowing backend to build logic around the current state of the front-end application, and serve the UI accordingly. Figure 3.16 and 3.17 show this through an hypothetical sign up form with steps distributed across multiple steps. On figure 3.16, we see the pseudo-code implementation of the stateless backend for this scenario. It can be seen that the current step user is at on the form, and the input state data collected thus far is kept in the volatile state on the front-end. The state gets passed on to the backend on every request made to the `/signup` endpoint. We retrieve the current step, and serve different responses accordingly. If user completed the last step, then we execute our application logic (which in this case is signing user up), and render the result page. Otherwise, we increment the current step, combine the form data collected thus far, and render the form template with these. On the form template on figure 3.17, we use the `<state>` block to instruct the Intertext client to record these new set of data to the front-end, so it could be passed on to the backend on the next request. Moreover, we render the correct form UI based on the step user is currently at.

Another type of storage is the persisted storage. As the name suggests, values stored this way is persisted across sessions. Unlike the volatile storage, in which the data is kept in the memory, persisted storage gets stored locally using the storage option available to the host platform. By default, data available in the persisted storage is not passed on to the backend in every request automatically, but can be configured to do so.

In order to protect user privacy, Intertext clients block cross-origin storage reads/writes, preventing users to be tracked across Intertext applications. In other words, state management is bound to the origin. Lets say `"sub1.example.com"` wrote data to the storage. This data can only be read or manipulated by the very same domain. `"sub2.example.com"` or

```
router.get('/signup', async (req, res, next) => {

  const currentStep = req.body.state?.current_step ?? 0
  const currentInputState = req.body.inputState
  const previousInputState = req.body.state.input_state

  // check if user finished the form
  if (currentStep == LAST_STEP) {

    // execute signup logic
    const error = await signupUser(req.body.state?)

    // render success or error page based on
    // the signup result
    res.render(error
      ? 'signup_form_error'
      : 'signup_form_success',
      { error });

  } else {

    // render signup form
    res.render('signup_form', {
      // increment the current step
      step: currentStep + 1
      // merge current form data with previous
      inputState: merge(
        previousInputState,
        currentInputState
      )
    });
  }
});
```

Figure 3.16: Handling a sign-up flow with multiple steps

```
<!-- signup_form -->

<!-- set front-end state to read on later requests -->
<state key="current_step">{{ step }}</state>
<state key="input_state">{{ inputState }}</state>

{{#ifEquals step 1}}
  <!-- form elements for step 1 -->
{{/ifEquals}}

{{#ifEquals step 2}}
  <!-- form elements for step 2 -->
{{/ifEquals}}

<!-- ... -->

<!-- signup_form_success -->

<h1 intent="success">Sign Up Successful!</h1>
<p intent="success">Please check your verification email</p>

<!-- signup_form_error -->

<h1 intent="error">Something went wrong</h1>
<p intent="error">{{error}}</p>
```

Figure 3.17: Template files for multi-step sign up flow at Figure 3.16

"sub.example2.com" or any other domain will not have access to this data. While this behaviour protects users from cross-origin tracking, it may also inconvenience some users, especially in scenarios such as services that shares the same user login that is distributed across multiple domains/subdomains. As mentioned further in the Future Work section, we plan to add a feature where Intertext clients would ask for user permission to share data between domains/subdomains instead of directly blocking it.



# 4

## Implementation

The primary language used to implement Intertext is Typescript. Typescript is a superset of Javascript that brings it typing support for safety and a better developer experience. Typescript gets transpiled into Javascript during build time, therefore the byproduct is plain Javascript. Sharing the same language between Intertext clients brings a number of advantages, mostly code sharing.

### 4.1 Engine

Code sharing between Intertext clients has been an important priority while creating the architecture for several reasons. First of all, it is crucial for Intertext clients to behave consistently for some tasks that are common to every Intertext client, such as parsing of IUIDL, rendering of the components, behavioural aspects of the clients, the data structure and so on. These are the cornerstone of Intertext, and they must behave exactly the same across clients. In other words, everything that isn't client-specific needs to be consistent and predictable. This is where Intertext engine comes in play.

Intertext engine implements several aspects of Intertext clients; most notably it parses IUIDL, internally converts it into JSON, and uses the output to render the components. Intertext clients are only concerned with the view layer, but they use the engine as a black box system. The engine also defines the data types of every component, and internal mechanics of the build

and rendering process. It exposes the types for Intertext clients built with Typescript to consume. Intertext engine is built as a standalone library, it exposes the main functions and types, and it is designed to be used in several different ways.

At the time of writing, all Intertext clients that are implemented so far were built with Typescript; they simply import and use the engine as a dependency. Their build system picks up the engine and adds it to their final bundle. However this approach is not possible for potential future clients that do not built on a Javascript runtime. To overcome this issue, the engine can be compiled as a Node module, and be deployed to a Node.js server. This gives us the ability to use the engine on any software system that can send or receive HTTP requests. Moreover, with the same logic, the engine can be deployed locally alongside any Intertext client that runs on a platform that can support Node.js runtime. It can then be deployed and served locally, and used to communicate with the local Intertext client using any protocol that is supported by the host platform as long as there exists a data exchange.

For instance, let's say a car manufacturer wanted to build an Intertext client for their in-car entertainment system using their own components. They could potentially install Node.js on their computer system, or even install a simple raspberry-pi-like chip that can run Node.js. Then, they can install the engine as a standalone application in it, and utilize any protocol to build a communication channel between the Intertext client they built on their computer system and the chip. Then, they could serve the IUIDL code either through a network connection (if and when available), or from a locally running server, potentially from the same one the client uses to interact with the engine.

## 4.2 Layout

The premise of Intertext is that there will be a unified implementation of UIs in a more or less abstract format, which will then be reified into fully functional front-ends on many different environments. These environments have different requirements and limitations; and an all-in-one solution is needed that will handle all these different requirements and limitations with minimal compromises. One of the aspects that needed consideration was the variation in screen sizes. We had to use a layout system that can be used to build a UI once, and would render properly on various screen sizes.

The obvious solution to this problem was using web technologies, as responsive design is one of the cornerstones of web development. There are many well defined specifications in place to build UIs for different screen

sizes. But we cannot directly use web technologies as many of the clients are not browser based. Therefore, we had to use an hybrid approach that can work universally.

### 4.2.1 Layout System

IUIDL offers a powerful layout system based on the css flexbox specification. It allows developers to build responsive layouts and position UI elements with ease. Intertext web client makes use of the browser implementation of CSS flexbox, whereas for other non-browser based clients, we use Facebook's open source library Yoga [3]. Yoga is a layout engine that implements css flexbox specifications in C++, therefore it can be used on any ecosystem that can execute C++ to calculate the positioning of the UI elements. We currently use Yoga layout engine indirectly on our command-line client, for it was built with a framework that internally uses the Yoga engine.

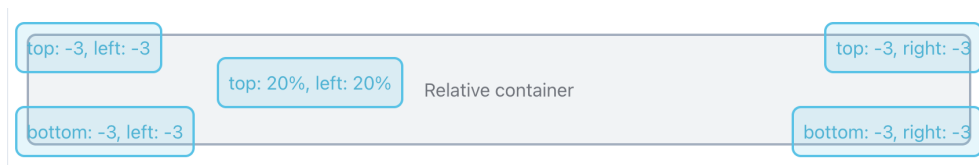


Figure 4.1: Absolute positioning

Apart from flexbox-based positioning, Intertext layout system also supports absolute positioning. Absolute positioning, as opposed to relative positioning, detaches a component from the flow of the page and positions it relative to its first relatively-positioned parent, having it be floating over everything else, as shown in Figure 4.2.

One advantage to this approach is that all our clients will render UI elements using the same layout, which will give users a sense of familiarity when switching between the clients. For instance, when a user accustomed to using the web version of Intertext switches to, say the command-line version, they will immediately recognise the layout and be able to navigate without having to adjust. This approach will also allow developers to build responsive layouts to some extent. Given that the same UIDL code will be used to render on multiple clients including ones with narrow and small screens, responsiveness goes a long way. At the time of writing, only flexbox features such as "flex-wrap" can be used for achieving a degree of responsiveness. However we plan to support screen-size aware variants of component properties, even conditional rendering of components based on the screen size. This can be

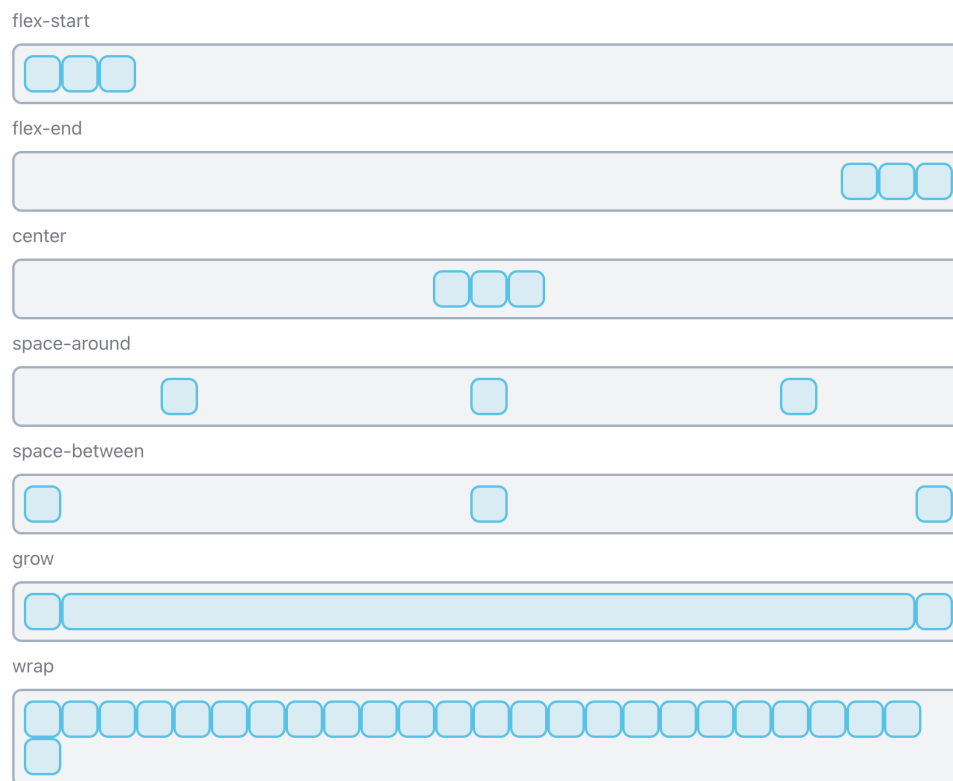


Figure 4.2: Flexbox system



thought of as "media queries" in css terminology. More on this will be on the Future Work section.

### 4.2.2 Unit System

One of the compromises we had to make for our an all-in-one layout solution was the unit system. We refrained from using definitive units that are specific to one platform, such as pixels. Most platforms use a different measurement system, for instance 1 unit for command-line applications represents a single space; character width for the x-axis and line height for the y-axis.

As a solution, we offer three different types of units; percentages, spaces and sizes. These units can be used for every property that accepts a unit; such as for width, height, margins, paddings and so on. Percentages are universal and can be calculated on any platform that we can get the screen dimensions of. We are able to use it in web environment natively, also the Yoga engine supports percentage values. Spaces are numeric values that starts from 0, increments by 0.5 until 4, by 1 until 12, by 2 until 16, and keeps increasing exponentially until 96. These values are meant to be used for spacing; for paddings, margins, gutter sizes, and other cases where spaces between UI elements needs to be determined. They are mapped to values on different platforms that are reasonable and look approximately same. Finally, we have sizes, which are used for determining the size of UI elements; in properties such as width, height, minWidth/maxWidth, minHeight/maxHeight and so on. While it also makes sense to use percentages for sizes, in cases where sizes should be fixed and not be changing based on the screen size, the size units can be used. Sizes are named values and are mapped to an appropriate value on every platform. They start from 3xs (xsmall) - up to xs, sm (small), md (medium), lg (large), xl (xlarge), 2xl and goes all the way to 8xl.

### 4.2.3 Properties

```
<block height="24" alignItems="center" justifyContent="center">  
  <text>This is centered</text>  
</block>
```

Figure 4.3: Layout properties

IUIDL defines some properties that can be used to leverate the layout system to determine sizes, spacing and positioning of UI elements. Figure 4.3 shows an example of how layout properties can be used to center a text

in a block. For positioning, *position* prop can be used, which takes *relative* or *absolute*. The default is *relative*. For spacing, supported values are *top*, *bottom*, *left*, *right*, *margin*, *marginTop*, *marginBottom*, *marginLeft*, *marginRight*, *padding*, *paddingTop*, *paddingBottom*, *paddingLeft* and *paddingRight*. As for sizing, accepted values are *width*, *minWidth*, *maxWidth*, *height*, *minHeight* and *maxHeight*. All these values can take all supported unit types. To leverage the flexbox properties, the accepted properties are *alignContent*, *alignItems*, *alignSelf*, *flexDirection*, *flexWrap*, *flexGrow*, *flexShrink*, *flexBasis* and *justifyContent*. Flexbox properties follow the flexbox specification [1] [2]. All layout properties mentioned above are accepted by and can be used with every Intertext component.

## 4.3 Components

Components are the building block of Intertext applications, they are a set of pre-defined building blocks provided out of the box, which can be assembled to create a functional front-end application. They define the view layer, and they are handled differently on every Intertext client. While the rendering logic and flow of the components are fully driven by engine for consistency, it exposes functions for Intertext clients to register renderers for each component. The renderer function gets called by the engine during the render flow; it receives the props that are the given to the component, the children of the component (if any) that the function is expected to be passing on to the next call level of the recursive rendering flow, and are expected to return the component instance that needs to be rendered, which will later be used to render the UI. Figure 4.4 shows an example snippet of the block renderer being registered to the engine on Intertext web client.

### 4.3.1 Block

Block component is the most basic and primitive building block of Intertext. It could be thought of as the *div* element of HTML, its main purpose is to enclose components and other blocks. It is primarily a layout component, as it can group and position elements together. Moreover, it can also double as a display component by accepting the *intent* property. When an intent is provided, it renders the block appropriately, thus it can be also used to convey a message. They also accept all layout properties. Figure 3.13 shows an example of the block component on the web client, with and without intents.

```
import Engine from "@intertext/engine"
import Block from "../components/core/Block/Block"

const engine = new Engine()

engine.renderer.registerBlockRenderer(({
  index,
  children,
  props,
}) => (
  <Block key={index} {...props}>
    {engine.renderer.render({ branch: children })}
  </Block>
))
```

Figure 4.4: Registering a component renderer

### 4.3.2 Grid

Grid is another layout component that can be used to render its children in grid formation. It accepts *cols* property, which can be used to define how many columns should the grid have, and what the size of the column should be. It accepts numeric values inside curly brackets delimited with a comma. Number of items in curly brackets represents the number of columns, and each number represents units that the columns should take up, where the total number of units is the sum of the numbers inside curly brackets. For instance, if *cols* value was `[1, 1, 1, 1]`, it would mean that the grid has four columns that are divided equally, where `[1, 3]` would represent two columns with widths being 25% to 75%. Grid component also takes a *gap* property to describe the gutter size. It can take any one of the spacing values. Any number of children can be provided to the grid component, without the need to nest them in any way, it will automatically position its immediate children based on the given configuration. 4.5 shows an example of how grid component can be used, and Figure 4.6 shows some render outputs of the grid component for the web client.

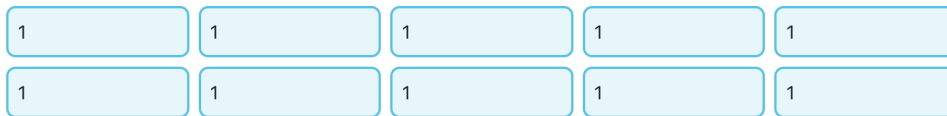
### 4.3.3 Typography

There are multiple components for typography, most of which are based off of the *text* component. All components for typography stem from the *text* component, but for ease of use, we defined aliases that utilize the *text*

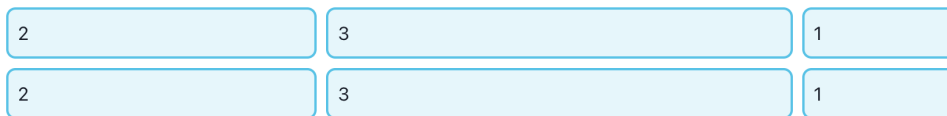
```
<grid cols="[2, 3, 1]" gap="2">
  <block intent="info">2</block>
  <block intent="info">3</block>
  <block intent="info">1</block>
  <block intent="info">2</block>
  <block intent="info">3</block>
  <block intent="info">1</block>
</grid>
```

Figure 4.5: How grid component is used

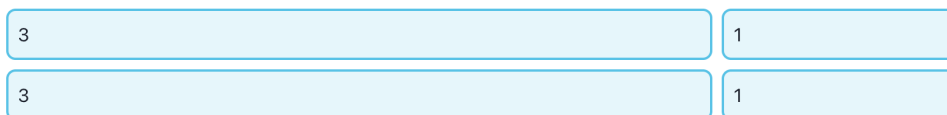
cols: [1, 1, 1, 1, 1]



cols: [2, 3, 1]



cols: [3, 1]



cols: [1, 3]

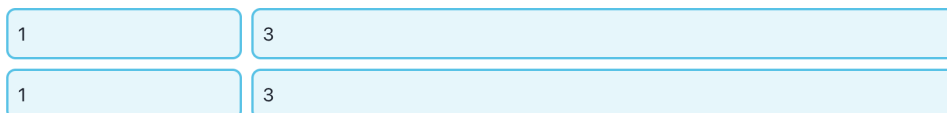


Figure 4.6: Grid component

component with a set configuration. The properties that the *text* component accepts are *p*, *h1*, *h2*, *h3*, *b*, *i*, *u*, *block*, *muted*, *code* and *intent*. *p* stands for "paragraph", which is used to define a paragraph block. *h1*, *h2*, *h3* represents headings, and their respective number represents the heading level. The behaviour is similar to HTML. *b*, *i*, *u* are styling options, represents "bold", "italics" and "underlined" respectively. *muted* renders the text with a muted color, useful for descriptions, subtexts or any secondary text that needs to be less attention grabbing. *code* renders the text with a light background and with monospace font family, should be used with codes.

By default, the *text* component renders inline, meaning, the render output of `<text>1</text><text>2</text>` will appear side by side, as "12". However, this behaviour changes for block-level properties; *p*, *h1*, *h2* and *h3*. *text* components that has these properties will take up the full line, and any other text component before or after them will fall into a new line. *block* property is a way to prevent text from rendering inline, without giving it any particular style. Text component also accepts *intent* property, which renders the text with the a color given its intent. Last but not least, text component can accept all layout properties just like any other component. The aliases that can be used in place of *text* are *p*, *h1*, *h2* and *h3* - which will render a *text* component with the respective styles applied. All properties that the text component accepts can be used together to combine styles. For instance, a configuration such as *intent="error"*, *b="true"*, *u="true"*, *h2="true"* *muted="true"* will result as a red h2 tag, muted, that is bold and underlined.

### 4.3.4 Literals

Literals are standalone string (or number) values provided to components (other than the *text* component) as children. Figure 4.7 shows the difference between a literal and a text value. While literals aren't exactly a component themselves, they are nonetheless values that rendering needs to be handled by each client, thus they need their own renderer. The way literals are handled differs from platform to platform, for instance on the web client they can be rendered as is, while on the command line client, they are required to be wrapped within a local Text node.

### 4.3.5 Collapse

Collapse component can be used to hide long-form content within a collapsible panel that is closed by default, and can be toggled by clicking the handle.

```
<text>Handled by text renderer</text>  
<block><text>Handled by text renderer</text></block>  
<block>Handled by literal renderer</block>
```

Figure 4.7: Difference between text and literal nodes

```
<collapse>  
  <collapse.handle>  
    <text>This is the handle</text>  
  </collapse.handle>  
  <text>This is the content</text>  
</collapse>
```

Figure 4.8: How grid component is used

It accepts the complex attribute *handle* that takes the contents of the collapse handle. Its children will be rendered inside of the collapse. Collapse components can be nested, in which case each collapse component will operate individually. Figure 4.8 shows how it can be used, and Figure 4.9 shows the render output of a collapse component on the Intertext web client.

### 4.3.6 Button

Button component is simplest and the most primitive way of triggering commands. It accepts the *onClick* complex attribute, which can accept one or many commands under it.

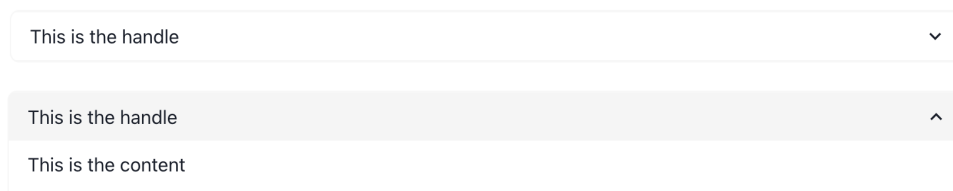


Figure 4.9: Collapse component opened and closed state

### 4.3.7 Input

## 4.4 Commands

Similar to the components, commands also are registered to the engine. However the registration of the commands are more dynamic compared to the components; while components are static and are immediately available right after the initialization, some commands rely on other artifacts to be available before they are able to be registered. For instance on the web client, the commands related to the client state requires the root component to be mounted, therefor they can only be registered after the mounting occurs. These requirements changes from client to client based on the platform requirements.

### 4.4.1 State

State command is used to set the state or retrieve read a value from the state. This command is expected to be registered to the engine by each Intertext client based on how the state is handled on the host platform. For instance, for the web client, the client state is controlled by the root component, thus it can only be initialized once the component is initialized and mounted on screen. State command accepts a *key* attribute, which takes the key of the state value to be set. The children of the state command takes the value of the state to be set. Unlike other commands, state command can also be used as a component to display state values on the screen. When the state command is used without providing a value, Intertext engine reads the state value, and interprets it as a literal value, uses the *literalRenderer* to display it on the screen. Figure 4.10 shows the syntax of how setting and reading the state works.

### 4.4.2 Alert

```
<!-- set the 'name' state value -->  
<state key="name">John</state>
```

```
<!-- read the 'name' state value -->  
<!-- renders "Hello, John!" -->  
<block>  
  <text>Hello, </text>  
  <state key="name"></state>  
  <text>!</text>  
</block>
```

Figure 4.10: How grid component is used



# 5

## Evaluation

- User evaluation - Questionnaire - Video / Prototype ask questions



# 6

## **Conclusion**



# 7

## Discussion

### 7.1 Future Work

- limitations
  - more intertext clients - more themes - user-made themes
  - refined control on persisted storage (expiration dates) - cross-origin storage read
  - media queries
- Sum up main contributions - How you address the research questions - Future work





# Appendix





# Bibliography

- [1] Css flexible box layout module level 1.
- [2] Mdn basic concepts of flexbox.
- [3] Yoga layout engine.
- [4] Marc Abrams, Constantinos Phanouriou, Alan L Batongbacal, Stephen M Williams, and Jonathan E Shuster. Uiml: an appliance-independent xml user interface language. *Computer networks*, 31(11-16):1695–1708, 1999.
- [5] Henriette Eisfeld and Felix Kristallovich. The rise of dark mode: A qualitative study of an emerging user interface design trend, 2020.
- [6] Ericsson. Ericsson mobility report, 2021.
- [7] Josefina Guerrero-Garcia, Juan Manuel Gonzalez-Calleros, Jean Vanderdonckt, and Jaime Munoz-Arteaga. A theoretical survey of user interface description languages: Preliminary results. In *2009 Latin American Web Congress*, pages 36–43. IEEE, 2009.
- [8] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. Usixml: A language supporting multi-path development of user interfaces. In *IFIP International Conference on Engineering for Human-Computer Interaction*, pages 200–220. Springer, 2004.
- [9] Rui Lopes, Daniel Gomes, and Luís Carriço. Web not for all: a large scale study of web accessibility. In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A)*, pages 1–4, 2010.
- [10] Tim A Majchrzak, Andreas Biørn-Hansen, and Tor-Morten Grønli. Progressive web apps: the definite approach to cross-platform development? 2018.

- [11] Fabio Paternò and Carmen Santoro. A unified method for designing interactive systems adaptable to mobile and stationary platforms. *Interacting with Computers*, 15(3):349–366, 2003.
- [12] Fabio Paterno’, Carmen Santoro, and Lucio Davide Spano. Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16(4):1–30, 2009.
- [13] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.
- [14] Angel Puerta and Jacob Eisenstein. Ximl: a common representation for interaction data. In *Proceedings of the 7th international conference on Intelligent user interfaces*, pages 214–215, 2002.
- [15] Sacha Greif, Raphaël Benitte. State of js, 2020.
- [16] Nathalie Souchon and Jean Vanderdonckt. A review of xml-compliant user interface description languages. In *International Workshop on Design, Specification, and Verification of Interactive Systems*, pages 377–391. Springer, 2003.
- [17] Catherine E Tucker. The economics of advertising and privacy. *International journal of Industrial organization*, 30(3):326–329, 2012.