Hypermedia In Action

3. Hyperview: A Mobile Hypermedia

In this chapter, we will:

- Discuss shortcomings with the current state of mobile app development
- Explain how a Hypermedia architecture can address those problems
- Introduce Hyperview as a Hypermedia for mobile application development
- Dive into HXML to show the power of its syntax

3.1. State of mobile app development

In web development, you deliver JS,HTML,CSS,JSON on-demand to a standard web browser. This is not the case with a native mobile app.

With native mobile apps, you compile code into an executable binary targeting the underlying operating system (Android or iOS). This binary gets uploaded and approved through app stores controlled by Google and Apple. When you install or update an app, you're downloading the binary and running the code directly on your device's OS. In this way, mobile apps have a lot in common with old-school desktop apps for Mac or Windows.

There is one important difference between PC desktop apps of yesteryear and today's mobile apps. These days, almost all mobile apps are "networked". By networked, I mean the app needs to read and write data over the Internet to implement core functionality. What's the easiest way to make a mobile app networked? Well, the app stores already require developer to write & distribute code that runs on the user's device. So why not have that frontend code make requests to a backend, and update the UI based on the response?

Thus, developers are naturally led into a SPA-like pattern for developing mobile apps:

- The frontend on the mobile device, runs code to create and update the UI
- The backend is an API, called by the frontend

Just like with SPAs on the web, this architecture has a big downside: the app's logic gets spread across the frontend and backend. Sometimes, logic needs to be duplicated (like to validate form data). Other times, the frontend and backend each implement one part of the

app's overall logic. To understand what the app does, a developer needs to trace interactions between two very different codebases.

There's another downside that affects mobile apps more than SPAs: API churn. Remember, the app stores control how your app gets distributed and updated. Users can even control if and when they get updated versions of your app. As a mobile developer, you can't assume that every user will be on the latest version of your app. Your frontend code gets fragmented across many versions, and now your backend needs to support all of them.

3.2. Hypermedia for Mobile Apps

We've seen that the hypermedia architecture can address the problems of SPAs on the web. But can hypermedia work for mobile apps as well? The answer is yes!

Just like on the web, we can use Hypermedia formats on mobile and let it serve as the engine of application state. All of the logic is controlled from the backend, rather than being spread between two codebases. Hypermedia architecture also solves the annoying problem of API churn on mobile apps. Since the backend servs a Hypermedia response containing both data and actions, there's no way for the data and UI to get out of sync. No more worries about backwards compatibility or maintaining multiple API versions.

So how can you use Hypermedia for your mobile app? There are two approaches employing hypermedia to build & ship native mobile apps today:

- Web views, which wraps the trusty web platform in a mobile app shell
- Hyperview, a new hypermedia format I designed specifically for mobile apps

3.2.1. Web views

The simplest way to use hypermedia on mobile is to make a web app! Both Android and iOS SDKs provide "web views": chromeless web browsers tbat can be embedded in native apps. Tools like Apache Cordova make it easy to take the URL of a website, and spit out native iOS and Android apps based on web views. If you already have a responsive web app, this gives you "native" Hypermedia apps for free. Sounds too good to be true, right?

Of course, there is a fundamental limitation with this approach. The web and mobile are different platforms, with different capabilities and UX conventions. HTML doesn't natively support common UI patterns of mobile apps. One of the biggest differences is around how each platform handles navigation. On the web, navigation is page-based, with one page

replacing another and the browser providing back/forward buttons to navigate the page history. On mobile, navigation is more complex, and tuned for the physicality of gesture-based interactions.

- To drill down, screens slide on top of each other, forming stacks of screens.
- A nav bar at the bottom of the app allows switching between various stacks of screens.
- Modals slide up from the bottom of the app, covering the other stacks and the nav bar.
- Unlike with web pages, all of these screens are still present in memory, rendered and updating based on app state.

The navigation architecture is a major difference between how mobile and web apps function. But it’s not the only one. Many other UX patterns are present in mobile apps, but are not natively supported on the web:

- pull-to-refresh to refresh content in a screen
- horizontal swipe on UI elements to reveal actions
- sectioned lists with sticky headers

While these interactions are not natively supported by web browsers, they can be simulated with JS libraries. Of course, these libraries will never have the same feel and performance as native gestures. And using these libraries usually requires embracing a JS-heavy SPA architecture. This puts us back at square 1! We wanted to avoid using the typical SPA architecture of native mobile app development, so we turned to a Web view. The web view allows us to use good-old hypermedia-based HTML. But to get the desired look & feel of a mobile app, we end up building a SPA in JS, losing the benefits of Hypermedia in the process.

Many mobile apps on iOS and Android are implemented as web views, and they work perfectly fine. If you already have a responsive web app, perhaps wrapping it in a Web view and distributing it through app stores is an easy proposition. But I tend to believe: "let the web be the web, and let mobile be mobile". Your responsive web app already works perfectly fine when accessed through Mobile Chrome or Mobile Safari. Progressive web app support has come a long way on iOS and Android. Users can "install" your web app to their home screen, and launch it with a single tap. Does your web app need to be in the app store too?

To build a hypermedia-based mobile app that feels and acts native, HTML isn't going to cut it. We need a format designed to represent the interactions and patterns of native mobile apps. That's exactly what Hyperview does.

3.2.2. Hyperview

Hyperview is an open-source framework that provides:

- A pre-defined hypermedia format for defining mobile apps called HXML
- A hypermedia client for HXML that can be embedded in an app binary on iOS and Android.
- Extension points in HXML and the client to customize the experience on a per-app basis.

The Format

HXML was designed to feel familiar to web developers, used to working with HTML. Thus the choice of XML for the base format. In addition to familiar ergonomics, XML is compatible with server-side rendering libraries. For example, Jinja2 is perfectly suited as a templating library to render HXML, as we'll see in the next chapter. The familiarity of XML and the ease of integration on the backend make it simple to adopt in both new and existing codebases.

```
<doc xmlns="https://hyperview.org/hyperview">
  <screen>
    <styles />
    <body>
      <header>
        <text>My first app</text>
      </header>
      <view>
        <text>Hello World!</text>
      </view>
    </body>
  </screen>
</doc>
```

htmx fill in the "missing parts of HTML" to enable rich web app experiences. If HTML was designed today, I believe the ideas of htmx would be part of the standrd spec, and natively supported by every browser without the need for a 3rd party JS library. Well, HXML was

designed today! The HXML format has built-in support for htmx-like interactions. Specifically, HXML is not limited to "click to navigate" and "press to submit" interactions like basic HTML. It supports a range of triggers and actions for modifying the content on a screen. These interactions are bundled together in a powerful concept of "behaviors". Developers can even define new behavior actions to add new capabilities to their app, without the need for scripting. We will learn more about behaviors in later sections of this chapter.

The client

Web developers are lucky. They can assume their users have access to a web browser capable of rendering any web app. In Hypermedia terms, the Hypermedia (HTML) client is already built and distributed to users. Half the work is done! The developer has to only build the backend to serve Hypermedia responses.

< diagram showing many browsers, pointing to one backend >

This is possible because the web is an open ecosystem built on standards. Any developer can build and host a web app, and any user can access it directly.

As we know, that’s not the case with mobile platforms. There is no open standard for building and distributing native mobile apps. And there’s definitely no widely distributed "HXML browser". So how can a developer deliver a Hypermedia mobile app using HXML? Well, unlike on the web, the mobile developer must provide both the backend to serve HXML, and a mobile client app to render those HXML responses.

< diagram showing one Hypermedia client, pointing to one backend >

It would be a lot to ask from developers to write their own HXML client. That’s why Hyperview provides an open-source client library, written in React Native. This library can be used to bootstrap a new mobile app, or it can be embedded in an existing app. In either case, developers get a full "HXML browser" without needing to write it from scratch.

At first, it might seem like the Hyperview approach requires extra work to write and maintain the mobile app client. But there is a benefit when the developer controls both the server and client. Did you ever wish you could fix a web browser bug? Or maybe add a new HTML element or features to the browser itself? The open nature of the web means that progress happens slowly. New features go through a lengthly standardization process. Browser vendors may prioritize bugs and features that don’t match your individual

priorities. As a web developer, you may need to wait years until browsers support the feature you need. Or, you can try to work around it with some kludgy JS.

Well, with Hyperview, there is no standards body or lengthly process for new features. As a Hyperview developer, you control your backend and mobile app client. Do you want to add a new element to HXML? Go right ahead! In fact, the Hyperview client library was built with extensibility in mind. There are extension points for custom UI elements and custom behaviors.

By extending the format and client itself, there's no need for Hyperview to include a scripting layer in HTMX. Features that require client-side logic get "built-in" to the client browser. HTMX responses remain pure, with UI and interactions represented in declarative XML.

3.2.3. Which Hypermedia architecture should I use?

We've discussed two approaches for creating mobile apps using Hypermedia architecture:

- create a backend that returns HTML, and serve it in a mobile app through a web view
- create a backend that returns HXML, and serve it in a mobile app with the Hyperview client

I purposefully described the two approaches in a way to highlight their similarities. After all, they are both using the Hypermedia architecture, just with different formats and clients. Both approaches solve the fundamental issues with traditional, SPA-like mobile app development:

- The backend controls the full state of the app.
- Our app's logic is all in one place.
- The app always runs the latest version, there's no API churn to worry about

So which approach should you use for a Hypermedia-driven mobile app? Based on my experience building both types of apps, I strongly believe the Hyperview results in a better user experience. The web-view will always feel out-of-place on iOS and Android; there's just no good way to replicate the patterns of navigation and interaction that mobile users expect. Hyperview was created specifically to address the limitations of thick-client and web view approaches. After the initial investment to learn Hyperview, you'll get all of the benefits of the Hypermedia architecture, without the downsides of a degraded user

experience.

Of course, if you already have a simple, mobile-friendly web app, then using a web-view approach is sensible. You will certainly save time from not having to serve your app as HXML in addition to HTML. But as I will show at the end of this chapter, it doesn't take a lot of work to convert an existing Hypermedia-driven web app into a Hyperview mobile app. But before we get there, we need to introduce the concepts of elements and behaviors in Hyperview. Then, we'll re-build our contacts app in Hyperview.

3.3. Introduction to HXML

3.3.1. Hello World!

HXML was designed to feel natural to web developers coming from HTML. Let's take a closer look at the "Hello World" app defined in HXML:

```
<doc xmlns="https://hyperview.org/hyperview"> ❶
  <screen> ❷
    <styles />
    <body> ❸
      <header> ❹
        <text>My first app</text>
      </header>
      <view> ❺
        <text>Hello World!</text> ❻
      </view>
    </body>
  </screen>
</doc>
```

❶ The root element of the HXML app

❷ The element representing a screen of the app

❸ The element representing the UI of the screem

❹ The element representing the top header of the screen

❺ A wrapper element around the content shown on the screen

❻ The text content shown on the screen

Nothing too strange here, right? The base syntax should be immediately familiar. Just like HTML, the syntax defines a tree of elements using start tags (`<screen>`) and end tags

(`</screen>`). Elements can contain other elements (`<view>`) or text (`Hello World!`). Elements can also be empty, represented with an empty tag (`<styles />`). However, you'll notice that the names of the HXML element are different from those in HTML. Let's take a closer look at each of those elements to understand what they do.

`<doc>` is the root of the HXML app. Think of it as equivalent to the `<html>` element in HTML. Note that the `<doc>` element contains an attribute `xmlns="https://hyperview.org/hyperview"`. This defines the default namespace for the doc. Namespaces are a feature of XML that allow one doc to contain elements defined by different developers. To prevent conflicts when two developers use the same name for their element, each developer defines a unique namespace. We will talk more about namespaces when we discuss custom elements & behaviors later in this chapter. For now, it's enough to know that elements in an HXML doc without an explicit namespace are considered to be part of the `https://hyperview.org/hyperview` namespace.

`<screen>` represents the UI that gets rendered on a single screen of a mobile app. It's possible for one `<doc>` to contain multiple `<screen>` elements, but we won't get into that now. Typically, a `<screen>` element will contain elements that define the content and styling of the screen.

`<styles>` defines the styles of the UI on the screen. We won't get too much into styling in Hyperview in this chapter. Suffice it to say, unlike HTML, Hyperview does not use a separate language (CSS) to define styles. Instead, styling rules such as colors, spacing, layout, and fonts are defined in HXML. These rules are then explicitly referenced by UI elements, much like using classes in CSS.

`<body>` defines the actual UI of the screen. The body includes all text, images, buttons, forms, etc that will be shown to the user. This is equivalent to the `<body>` element in HTML.

`<header>` defines the header of the screen. Typically in mobile apps, the header includes some navigation (like a back button), and the title of the screen. It's useful to define the header separately from the rest of the body. Some mobile OSes will use a different transition for the header than the rest of the screen content.

`<view>` is the basic building block for layouts and structure within the screen's body.

Think of it like a `<div>` in HTML. Note that unlike in HTML, a `<div>` cannot directly contain text.

`<text>` elements are the only way to render text in the UI. In this example, "Hello World" is contained within a `<text>` element.

That's all there is to define a basic "Hello World" app in HXML. Of course, this isn't very exciting. Let's cover some other built-in display elements.

3.3.2. UI Elements

Lists

A very common pattern in mobile apps is to scroll through a list of items. The physical properties of a phone screen (long & vertical) and the intuitive gesture of swiping a thumb up & down makes this a good choice for many screens.

HXML has dedicated elements for representing lists and items.

```
<list> ❶
  <item key="item1"> ❷
    <text>My first item</text> ❸
  </item>`
  <item key="item2">
    <text>My second item</text>
  </item>
</list>
```

❶ Element representing a list

❷ Element representing an item in the list, with a unique key

❸ The content of the item in the list.

Lists are represented with two new elements. The `<list>` wraps all of items in the list. It can be styled like a generic `<view>` (width, height, etc). A `<list>` element only contains `<item>` elements. Of course, these represent each unique item in the list. Note that `<item>` is required to have a `key` attribute, which is unique among all items in the list.

You might be asking, "Why do we need a custom syntax for lists of items? Can't we just use a bunch of `<view>` elements?". Yes, for lists with a small number of items, using nested `<views>` will work quite well. However, often the number of items in a list can be

long enough to require optimizations to support smooth scrolling interactions. Consider browsing a feed of posts in a social media app. As you keep scrolling through the feed, it's not unusual for the app to show hundreds if not thousands of posts. At any time, you can flick your finger to scroll to almost any part of the feed. Mobile devices tend to be memory-constrained. Keeping the fully-rendered list of items in memory could consume more resources than available. That's why both iOS and Android provide APIs for optimized list UIs. These APIs know which part of the list is currently on-screen. To save memory, they clear out the non-visible list items, and recycle the item UI objects to conserve memory. By using explicit `<list>` and `<item>` elements in HXML, the Hyperview client knows to use these optimized list APIs to make your app more performant.

It's also worth mentioning that HXML supports section lists. Section lists are useful for building screens like a list of contacts, where contacts are alphabetized and grouped in a section by their starting letter.

```
<section-list> ❶
  <section> ❷
    <section-title> ❸
      <text>A</text>
    </section-title>
    <item key="1"> ❹
      <text>Aaron</text>
    </item>
    <item key="2">
      <text>Adam</text>
    </item>
  </section>

  <section> ❺
    <section-title>
      <text>B</text>
    </section-title>
    <item key="3">
      <text>Bart</text>
    </item>
  </section>
</section-list>
```

❶ Element representing a list with sections

❷ The first section of contacts with names that begin with the letter "A"

❸ Element for the title of the section, rendering text of the letter "A"

❹ An item representing a contact that belongs to this section.

❺ A section for contacts with names that begin with "B".

You'll notice a couple of differences between `<list>` and `<section-list>`. The section list element only contains `<section>` elements, representing a group of items. A section can contain a `<section-title>` element. This is used to render some UI that acts as the header of the section. This header is "sticky", meaning it stays on screen while scrolling through items that belong to the corresponding section. Finally, `<item>` elements act the same as in the regular list, but can only appear within a `<section>`.

Images

Showing images in Hyperview is pretty similar to HTML, but there are a few differences.

```
<image source="/profiles/1.jpg" style="avatar" />
```

The `source` attribute specifies how to load the image. Like in HTML, the source can be an absolute or relative URL. Additionally, the source can be an encoded data URI, for example ``. However, the source can also be a "local" URL, refering to an image that is bundled as an asset in the mobile app. The local URL is prefixed with `./`:

```
<image source="./logo.png" style="logo" />
```

Using Local URLs is an optimization. Since the images are on the mobile device, they don't require a network request and will appear quickly. However, bundling the image with the mobile app binary increases the binary size. Using local images is a good tradeoff for images that are frequently accessed but rarely change. Good examples include the app logo, or common button icons.

The other thing to note is the presence of the `style` attribute on the `<image>` element. In HXML, images are required to have a style that has rules for the image's `width` and `height`. This is different from HTML, where `<img>` elements do not need to explicitly set a width and height. Web browsers will re-flow the content of a web page once the image is

fetched and the dimensions are known. While re-flowing content is a reasonable behavior for web-based documents, users do not expect mobile apps to re-flow as content loads. To maintain a static layout, HXML requires the dimensions to be known before the image loads.

3.3.3. Inputs

There's a lot to cover about inputs in Hyperview. Since this is meant to be an introduction and not an exhaustive resource, I'll highlight just a few types of inputs. Let's start with an example of the simplest type of input, a text field.

```
<text-field
  name="first_name" ❶
  style="input" ❷
  value="Adam" ❸
  placeholder="First name" ❹
/>
```

❶ The name used when serializing data from this input

❷ The style class applied to the UI element

❸ The currently value set in the field

❹ A placeholder to display when the value is empty

This element should feel familiar to anyone who's created a text field in HTML. One difference is that most inputs in HTML use the `<input>` element with a `type` attribute, eg `<input type="text">`. In Hyperview, each input has a unique name, in this case `<text-field>`. By using different names, we can use more expressive XML to represent the input.

For example, let's consider a case where we want to render a UI that lets the user select one among several options. In HTML, we would use a radio button input, something like `<input type="radio" name="choice" value="option1" />`. Each choice is represented as a unique input element. This never struck me as ideal. Most of the time, radio buttons are grouped together to affect the same name. The HTML approach leads to a lot of boilerplate (duplication of `type="radio"` and `name="choice"` for each choice). Also, unlike radio buttons on desktop, mobile OSes don't provide a strong standard UI for selecting one option. Most mobile apps use richer, custom UIs for these interactions. So in

HXML, we implement this UI using an element called `<select-single>`:

```
<select-single name="choice"> ❶
  <option value="option1"> ❷
    <text>Option 1</text> ❸
  </option>
  <option value="option2">
    <text>Option 2</text>
  </option>
</select-single>
```

❶ Element representing an input where a single choice is selected. The name of the selection is defined once here.

❷ Element representing one of the choices. The choice value is defined here.

❸ The UI of the selection. In this example, we use text, but we can use any UI elements.

The `<select-single>` element is the parent of the input for selecting one choice out of many. This element contains the `name` attribute used when serializing the selected choice. `<option>` elements within `<select-single>` represent the available choices. Note that each `<option>` element has a `value` attribute. When pressed, this will be the selected value of the input. The `<option>` element can contain any other UI elements within it. This means that we're not hampered by rendering the input as a list of radio buttons with labels. We can render the options as radios, tags, images, or anything else that would be intuitive for our interface. HXML styling supports modifiers for pressed and selected states, letting us customize the UI to highlight the selected option.

Like I mentioned, describing all features of inputs in HXML would take an entire chapter. Instead, I'll summarize a few other input elements and their features.

- `<select-multiple>` works like `<select-single>`, but it supports toggling multiple options on & off. This replaces checkbox inputs in HTML.
- The `<switch>` element renders a on/off switch that is common in mobile UIs
- The `<date-field>` element supports entering in specific dates, and comes with a wide range of customizations for formatting, settings ranges, etc.

Two more things to mention about inputs. First is the `<form>` element. The `<form>` element is used to group together inputs for serialization. When a user takes an action that

triggers a backend request, the Hyperview client will serialize all inputs in the surrounding `<form>` and include them in the request. This is true for both `GET` and `POST` requests. We will cover this in more detail when talking about behaviors later in this chatper. Also later in this chapter, I'll talk about support for custom elements in HXML. With custom elements, you can also create your own input elements. Custom input elements allow you to build incredible powerful interactions with simple XML syntax that integrates well with the rest of HXML.

3.3.4. Styling

So far, we haven't mentioned how to apply styling to all of the HXML elements. We've seen from the Hello World app that each `<screen>` can contain a `<styles>` element. Let's re-visit the Hello World app and fill out the `<styles>` element.

```
<doc xmlns="https://hyperview.org/hyperview">
  <screen>
    <styles> ❶
      <style class="body" flex="1" flexDirection="column" /> ❷
      <style class="header" borderBottomWidth="1" borderBottomColor="#ccc" />
      <style class="main" margin="24" />
      <style class="h1" fontSize="32" />
      <style class="info" color="blue" />
    </styles>

    <body style="body"> ❸
      <header style="header">
        <text style="info">My first app</text>
      </header>
      <view style="main">
        <text style="h1 info">Hello World!</text> ❹
      </view>
    </body>
  </screen>
</doc>
```

❶ Element encapsulating all of the styling for the screen

❷ Example of a definition of a style class for "body"

❸ Applying the "body" style class to a UI element

❹ Example of applying multiple style classes (h1 and info) to an element

You'll note that in HXML, styling is part of the XML format, rather than using a separate language like CSS. However, we can draw some parallels between CSS rules and the `<style>` element. A CSS rule consists of a selector and declarations. In the current version of HXML, the only available selector is a class name, indicated by the `class` attribute. The rest of the attributes on the `<style>` element are declarations, consisting of properties and property values.

UI elements within the `<screen>` can reference the `<style>` rules by adding the class names to their `<style>` property. Note the `<text>` element around "Hello World!" references two style classes: `h1` and `info`. The styles from the corresponding classes are merged together in the order they appear on the element. It's worth noting that styling properties are similar to those in CSS (color, margins/padding, borders, etc). Currently, the only available layout engine is based on flexbox.

Styles rules can get quite verbose. For the sake of brevity, I won't include the `<styles>` element in the rest of the examples in this chapter unless necessary.

3.3.5. Custom elements

The core UI elements that ship with Hyperview are quite basic. Most mobile apps require richer elements to deliver a great user experience. Luckily, HXML can easily accomodate custom elements in its syntax. This is because HXML is really just XML, aka "Extensible Markup Language". Extensibility is already built into the format! Developers are free to define new elements and attributes to represent custom elements.

Let's see this in action with a concrete example. Assume that we want to add a map element to our Hello World app. We want the map to display a defined area, and one or more markers at specific coordinates in that area. Let's translate these requirements into XML:

- An `<area>` element will represent the area displayed by the map. To specify the area, the element will include attributes for `latitude` and `longitude` for the center of the area, and a `latitude-delta` and `longitude-delta` indicating the +/- display area around the center.
- A `<marker>` element will represent a marker in the area. The coordinates of the marker will be defined by `latitude` and `longitude` attributes on the marker.

Using these custom XML elements, an instance of the map in our app might look like this:

```
<doc xmlns="https://hyperview.org/hyperview">
  <screen>
    <body>
      <view>
        <text>Hello World!</text>
        <area latitude="37.8270" longitude="122.4230" latitude-delta="0.1"
longitude=delta="0.1"> ❶
          <marker latitude="37.8118" longitude="-122.4177" /> ❷
        </area>
      </view>
    </body>
  </screen>
</doc>
```

❶ Custom element representing the area rendered by the map

❷ Custom element representing a marker rendered at specific coordinates on the map

The syntax feels right at home among the core HXML elements. However, there's a potential problem. "area" and "marker" are pretty generic names. I could see `<area>` and `<marker>` elements being used by a customization to render charts & graphs. If our app renders both maps and charts, the HXML markup would be ambiguous. What should the client render when it sees `<area>` or `<marker>`?

This is where XML namespaces come in. XML namespaces eliminate ambiguity and collisions between elements and attributes used to represent different things. Remember that the `<doc>` element declares that `https://hyperview.org/hyperview` is the default namespace for the entire document. Since no other elements define namespaces, every element in the example above is part of the `https://hyperview.org/hyperview` namespace.

Let's define a new namespace for our map elements. Since this namespace will not be the default for the document, we also need to assign the namespace to a prefix we will add to our elements:

```
<doc xmlns="https://hyperview.org/hyperview"
xmlns:map="https://mycompany.com/hyperview-map">
```

This new attribute declares that the `map:` prefix is associated with the namespace

"https://mycompany.com/hyperview-map". This namespace could be anything, but remember the goal is to use something unique that won't have collisions. Using your company/app domain is a good way to guarantee uniqueness. Now that we have a namespace and prefix, we need to use it for our elements:

```
<doc xmlns="https://hyperview.org/hyperview"
xmlns:map="https://mycompany.com/hyperview-map"> ❶
  <screen>
    <body>
      <view>
        <text>Hello World!</text>
        <map:area latitude="37.8270" longitude="122.4230" latitude-delta="0.1"
longitude=delta="0.1"> ❷
          <map:marker latitude="37.8118" longitude="-122.4177" /> ❸
        </map:area> ❹
      </view>
    </body>
  </screen>
</doc>
```

❶ Definition of namespace aliased to "map"

❷ Adding the namespace to the "area" start tag

❸ Adding the namespace to the "marker" self-closing tag

❹ Adding the nemsapce to the "area" end tag

That's it! If we introduced a custom charting library with "area" and "marker" elements, we would create a unique namespace for those elements as well. Within the HXML doc, we could easily disambiguate `<map:area>` from `<chart:area>`.

At this point you might be wondering, "how does the Hyperview client know to render a map when my doc includes `<map:area>`"? It's true, so far we only defined the custom element format, but we haven't implemented the element as a feature in our app. We will get into the details of implementing custom elements later in the next chapter.

3.3.6. Behaviors

As discussed in earlier chapters, HTML supports two basic types of interactions:

- Clicking a hyperlink: the client will make a GET request and render the response as a

new web page.

- Submitting a form: the client will make a (typically) POST request with the serialized content of the form, and render the response as a new web page.

Clicking hyperlinks and submitting forms is enough to build simple web applications. But relying on just these two interactions limits our ability to build richer UIs. What if we want something to happen when the user mouses over a certain element, or perhaps when they scroll some content into the viewport? We can't do that with basic HTML. Additionally, both clicks and form submits result in loading a full new web page. What if we only want to update a small part of the current page? This is a very common scenario in rich web applications, where users expect to fetch and update content without navigating to a new page.

So with basic HTML, interactions (clicks and submits) are limited and tightly coupled to a single action (loading a new page). Of course, using JavaScript, we can extend HTML and add some new syntax to support our desired interactions. Htmx (and Intercooler before it) do exactly that with a new set of attributes:

- Interactions can be added to any element, not just links and forms.
- The interaction can be triggered via a click, submit, mouseover, or any other JavaScript event.
- The actions resulting from the trigger can modify the current page, not just request a new page.

By decoupling elements, triggers, and actions, htmx allows us to build rich Hypermedia-driven applications in a way that feels very compatible with HTML syntax and server-side web development.

HXML takes the idea of defining interactions via triggers & actions and builds them into the spec. We call these interactions "behaviors". We use a special `<behavior>` element to define then. Here's an example of a simple behavior that pushes a new mobile screen onto the navigation stack:

```
<text>
  <behavior ❶
    trigger="press" ❷
    action="push" ❸
    href="/next-screen" ❹
  />
  Press me!
</text>
```

❶ The element encapsulating an interaction on the parent `<text>` element.

❷ The trigger that will execute the interaction, in this case pressing the `<text>` element.

❸ The action that will execute when triggered, in this case pushing a new screen onto the current stack.

❹ The href to load on the new screen.

Let's break down what's happening in this example. First, we have a `<text>` element with the content "Press me!". We've shown `<text>` elements before in examples of HXML, so this is nothing new. But now, the `<text>` element contains a new child element, `<behavior>`. This `<behavior>` element defines an interaction on the parent `<text>` element. It contains two attributes that are required for any behavior:

- `trigger`: defines the user action that triggers the behavior
- `action`: defines what happens when triggered

In this example, the `trigger` is set to `press`, meaning this interaction will happen when the user presses the `<text>` element. The `action` attribute is set to `push`. `push` is an action that will push a new screen onto the navigation stack. Finally, Hyperview needs to know what content to load on the newly pushed screen. This is where the `href` attribute comes in. Notice we don't need to define the full URL. Much like in HTML, the `href` can be an absolute or relative URL.

So that's a first example of behaviors in HXML. You may be thinking this syntax seems quite verbose. Indeed, pressing elements to navigate to a new screen is one of the most common interactions in a mobile app. It would be nice to have a simpler syntax for the common case. Luckily, `trigger` and `action` attributes have default values of `press` and

`push`, respectively. Therefore, they can be ommitted to clean up the syntax:

```
<text>
  <behavior href="/next-screen" /> ❶
  Press me!
</text>
```

❶ When pressed, this behavior will open a new screen with the given URL.

This markup for the `<behavior>` will produce the same interaction as the earlier example. With the default attributes, the `<behavior>` element looks similar to an anchor `<a>` in HTML. But the full syntax achieves our goals of decoupling elements, triggers, and actions:

- Behaviors can be added to any element, they are not limited to links and forms.
- Behaviors can specify an explicit `trigger`, not just clicks or form submits.
- Behaviors can specify an explicit `action`, not just a request for a new page.
- Extra attributes like `href` provide more context for the action.

Additionally, using a dedicated `<behavior>` element means a single element can define multiple behaviors. This let us execute several actions from the same trigger. Or, we can execute different actions for different triggers on the same element. We will show examples of the power of multiple behaviors at the end of this chapter. First we need to show the variety of supported actions and triggers.

Actions

Behavior actions in Hyperview fall into 3 general categories:

- Navigation actions, which load new screens and move between them
- Update actions, which modify the HXML of the current screen
- System actions, which interact with with OS-level capabilities.

Navigation Actions

We've already seen the simplest type of action, `push`. We classify `push` as a "navigation action", since it's related to navigating screens in the mobile app. Pushing a screen onto the navigation stack is just one of several navigation actions supported in Hyperview. Users also need to be able to go back to previous screens, open and close modals, switch between

tabs, or jump to arbitrary screens. Each of these types of navigations is supported through a different value for the `action` attribute:

- `push`: Push a new screen into the current navigation stack. This looks like a screen sliding in from the right, on top of the current screen.
- `new`: Open a new navigation stack as a modal. This looks like a screen sliding in from the bottom, on top of the current screen.
- `back`: This is a complement to the `push` action. It pops the current screen off of the navigation stack (sliding it to the right).
- `close`: This is a complement to the `new` action. It closes the current navigation stack (sliding it down).
- `reload`: Similar to a browser's "refresh" button, this will re-request the content of the current screen.
- `navigate`: This action will attempt to find a screen with the given `href` already loaded in the app. If the screen exists, the app will jump to that screen. If it doesn't exist, it will act the same as `push`.

`push`, `new`, and `navigate` all load a new screen. Thus, they require an `href` attribute so that Hyperview knows what content to request for the new screen. `back`, `close` do not load new screens, and thus do not require the `href` attribute. `reload` is an interesting case. By default, it will use the URL of the screen when re-requesting the content for the screen. However, if you want to replace the screen with a different one, you can provide an `href` attribute with `reload` on the behavior element.

Let's look at an example that uses several navigation actions on one screen:

```
<screen>
  <body>
    <header>
      <text>
        <behavior action="back" /> ❶
        Back
      </text>

      <text>
        <behavior action="new" href="/widgets/new" /> ❷
        New
      </text>
    </header>
    <text>
      <behavior action="reload" /> ❸
      Check for new widgets
    </text>
    <list>
      <item key="widget1">
        <behavior action="push" href="/widgets/1" /> ❹
      </item>
    </list>
  </body>
</screen>
```

❶ Takes the user to the previous screen

❷ Opens a new modal to add a widget

❸ Reloads the content of the screen, showing new widgets from the backend

❹ Pushes a new screen with details for a specific widget

Update Actions

Behavior actions are not just limited to navigating between screens. They can also be used to change the content on the current screen. We call these "update actions". Much like navigation actions, update actions make a request to the backend. However, the response is not an entire HXML document, but a fragment of HXML. This fragment is added to the HXML of the current screen, resulting in an update to the UI. The `action` attribute of the `<behavior>` determines how the fragment gets incorporated into the HXML. We also need to introduce a new `target` attribute on `<behavior>` to define where the fragment gets incorporated in the existing doc. The `target` attribute is an ID reference to an existing

element on the screen.

Hyperview supports the current update actions, representing different ways to incorporate the fragment into the screen:

- `replace`: replaces the entire target element with the fragment
- `replace-inner`: replaces the children of the target element with the fragment
- `append`: adds the fragment after the last child of the target element
- `prepend`: adds the fragment before the first child of the target element.

Let's look at some examples to make this more concrete. For these examples, let's assume our backend accepts `GET` requests to `/fragment`, and the response is a fragment of HXML that looks like `<text>My fragment</text>`.

```
<screen>
  <body>
    <text>
      <behavior action="replace" href="/fragment" target="area1" /> ❶
      Replace
    </text>
    <view id="area1">
      <text>Existing content</text>
    </view>

    <text>
      <behavior action="replace-inner" href="/fragment" target="area2" /> ❷
      Replace-inner
    </text>
    <view id="area2">
      <text>Existing content</text>
    </view>

    <text>
      <behavior action="append" href="/fragment" target="area3" /> ❸
      Append
    </text>
    <view id="area3">
      <text>Existing content</text>
    </view>

    <text>
      <behavior action="prepend" href="/fragment" target="area4" /> ❹
      Prepend
    </text>
    <view id="area4">
      <text>Existing content</text>
    </view>

  </body>
</screen>
```

❶ Replaces the area1 element with fetched fragment

❷ Replaces the child elements of area2 with fetched fragment

❸ Appends the fetched fragment to area3

❹ Prepends the fetched fragment to area4

In this example, we have a screen with four buttons corresponding to the four update actions: `replace`, `replace-inner`, `append`, `prepend`. Below each button, there's a corresponding `<view>` containing some text. Note that the `id` of each view matches the `target` on the behaviors of the corresponding button.

When the user presses the first button, the Hyperview makes a request for `/fragment`. Next, it looks for the target, ie the element with id "area1". Finally, it replaces the `<view id="area1">` element with the fetched fragment, `<text>My fragment</text>`. The existing view and text contained in that view will be replaced. To the user, it will look like "Existing content" was changed to "My fragment". In the HXML, the element `<view id="area1">` will also be gone.

The second button behaves in a similar way to the first one. However, the `replace-inner` action does not remove the target element from the screen, it only replaces the children. This means the resulting markup will look like `<view id="area2"><text>My fragment</text></view>`.

The third and fourth buttons do not remove any content from the screen. Instead, the fragment will be added either after (in the case of `append`) or before (`prepend`) the children of the target element.

For completeness, let's look at the state of the screen after a user presses all four buttons:

```
<screen>
  <body>
    <text>
      <behavior action="replace" href="/fragment" target="area1" />
      Replace
    </text>
    <text>My fragment</text> ❶

    <text>
      <behavior action="replace-inner" href="/fragment" target="area2" />
      Replace-inner
    </text>
    <view id="area2">
      <text>My fragment</text> ❷
    </view>

    <text>
      <behavior action="append" href="/fragment" target="area3" />
      Append
    </text>
    <view id="area3">
      <text>Existing content</text>
      <text>My fragment</text> ❸
    </view>

    <text>
      <behavior action="prepend" href="/fragment" target="area4" />
      Prepend
    </text>
    <view id="area4">
      <text>My fragment</text> ❹
      <text>Existing content</text>
    </view>

  </body>
</screen>
```

❶ Fragment completely replaced the target using `replace` action

❷ Fragment replaced the children of the target using `replace-inner` action

❸ Fragment added as last child of the target using `append` action

❹ fragment added as the first child of the target using `prepend` action

The examples above show actions making `GET` requests to the backend. But these actions can also make `POST` requests by setting `verb="post"` on the `<behavior>` element. For both `GET` and `POST` requests, the data from the parent `<form>` element will be serialized and included in the request. For `GET` requests, the content will be URL-encoded and added as query params. For `POST` requests, the content will be form-URL encoded and set on the request body. Since they support `POST` and form data, update actions are often used to send data to the backend.

So far, our example of update actions require getting new content from the backend and adding it to the screen. But sometimes we just want to change the state of existing elements. The most common state to change for an element is its visibility. Hyperview has `hide`, `show`, and `toggle` actions that do just that. Like the other update actions, `hide`, `show`, and `toggle` use the `target` attribute to apply the action to an element on the current screen.

```
<screen>
  <body>
    <text>
      <behavior action="hide" target="area" /> ❶
      Hide
    </text>

    <text>
      <behavior action="show" target="area" /> ❷
      Show
    </text>

    <text>
      <behavior action="toggle" target="area" /> ❸
      Toggle
    </text>

    <view id="area"> ❹
      <text>My fragment</text>
    </view>
  </body>
</screen>
```

❶ Hides the element with id "area".

❷ Shows the element with id "area".

❸ Toggles the visibility of the element with id "area".

❹ The element targeted by the actions.

In this example, the three buttons labeled "Hide", "Show", and "Toggle" will modify the display state of the `<view>` with ID "area". Pressing "Hide" multiple times will have no affect once the view is hidden. Likewise, pressing "Show" multiple times will have no affect once the view is showing. Pressing "Toggle" will keep flipping the visibility status of the element between showing and hidden.

Hyperview comes with other actions that modify the existing HXML. We won't cover them in detail, but I'll mention them breifly here:

- `set-value`: this action can set the value of an input element such as `<text-field>`, `<switch>`, `<select-single>`, etc.
- `select-all` and `unselect-all` work with the `<select-multiple>` element to select/deselect all options.

System Actions

Some standard Hyperview actions don't interact with the HXML at all. Instead, they expose functionality provided by the mobile OS. For example, both Android and iOS support a system-level "Share" UI. This UI allows sharing URLs and messages from one app to another app. Hyperview has a `share` action to support this interaction. It involves a custom namespace, and share-specific attributes.

```
<behavior
  xmlns:share="https://instawork.com/hyperview-share" ❶
  trigger="press"
  action="share" ❷
  share:url="https://www.instawork.com" ❸
  share:message="Check out this website!" ❹
/>
```

❶ Defines the namespace for the share action.

❷ The action of this behavior will bring up the share sheet.

❸ URL to be shared

❹ Message to be shared

We've seen XML namespaces when talking about custom elements. Here, we are using a namespace for the `url` and `message` attributes on the `<behavior>`. These attribute names are generic and likely used by other components and behaviors, so the namespace ensures there will be no ambiguity. When pressed, the "share" action will trigger. The values of the `url` and `message` attributes will be passed to the system Share UI. From there, the user will be able to share the URL & message via SMS, email, or other communication apps.

The `share` action shows how a behavior action can use custom attributes to pass along extra data needed for the interactions. But some actions require even more structured data. This can be provided via child elements on the `<behavior>`. Hyperview uses this to implement the `alert` action. The `alert` action shows a customized system-level dialog box. This dialog needs configuration for a title and message, but also for customized buttons. Each button needs to then trigger another behavior when pressed. This level of configuration cannot be done with just attributes, so we use custom child elements to represent the behavior of each button.

```
<behavior
  xmlns:alert="https://hyperview.org/hyperview-alert" ❶
  trigger="press"
  action="alert" ❷
  alert:title="Continue to next screen?" ❸
  alert:message="Are you sure you want to navigate to the next screen?" ❹
>
  <alert:option alert:label="Continue"> ❺
    <behavior action="push" href="/next" /> ❻
  </alert:option>
  <alert:option alert:label="Cancel" /> ❼
</behavior>
```

❶ Defines the namespace for the alert action.

❷ The action of this behavior will bring up a system dialog box.

❸ Title of the dialog box.

❹ A "continue" option in the dialog box

❺ When "continue" is pressed, push a new screen onto the navigation stack.

❻ A "cancel" option that dismisses the dialog box.

Like the `share` behavior, `alert` uses a namespace to define some attributes and elements. The `<behavior>` element itself contains the `title` and `message` attributes for the dialog box. The button options for the dialog are defined using a new `<option>` element nested in the `<behavior>`. Notice that each `<option>` element has a label, and then optionally contains a `<behavior>` itself! This structure of the HXML allows the system dialog to trigger any interaction that can be defined as a `<behavior>`. In the example above, pressing the "Continue" button will open a new screen. But we could just as easily trigger an update action to change the current screen. We could even open a share sheet, or a second dialog box. But please don't do that in a real app! With great power comes great responsibility.

Custom Actions

You can build a lot of mobile UIs with Hyperview's standard navigation, update, and system actions. But the standard set may not cover all interactions you will need for your mobile app. Luckily, the action system is extensible. In the same way you can add custom elements to Hyperview, you can also add custom behavior actions. Custom actions have a similar syntax to the `share` and `alert` actions, using namespaces for attributes that pass along extra data. Custom actions also have full access to the HXML of the current screen, so they can modify the state or add/remove elements from the current screen. In the next chapter, we will create a custom behavior action to enhance our mobile contacts app.

Triggers

We've already seen the simplest type of trigger, a `press` on an element. Hyperview supports many other common triggers used in mobile apps.

`longPress`

Closely related to a press is a long-press. A behavior with `trigger="longPress"` will trigger when the user presses and holds on the element. "Long-press" interactions are often used for shortcuts and power features. Sometimes, elements will support different actions for both a `press` and `longPress`. This is done using multiple `<behavior>` elements on the same UI element.

```
<text>
  <behavior trigger="press" action="push" href="/next-screen" /> ❶
  <behavior trigger="longPress" action="push" href="/secret-screen" /> ❷
  Press (or long-press) me!
</text>
```

❶ Normal press will open the next screen

❷ Long press will open a different screen

In this example, a normal press will open a new screen and request content from `/next-screen`. However, a long press will open a new screen with content from `/secret-screen`. This is a contrived example for the sake of brevity. A better UX would be for the long-press to bring up a contextual menu of shortcuts and advanced options. This could be achieved by using `action="alert"` and opening a system dialog box with the shortcuts.

`load`

Sometimes we want an action to trigger as soon as the screen loads. `trigger="load"` does exactly this. One use case is to quickly load a shell of the screen, and then fill in the main content on the screen with a second update action.

```
<body>
  <view>
    <text>My app</text>
    <view id="container"> ❶
      <behavior trigger="load" action="replace" href="/content" target="container">
❷
      <text>Loading...</text> ❸
    </view>
  </view>
</body>
```

❶ Container element without the actual content

❷ Behavior that immediately fires off a request for /content to replace the container

❸ Loading UI that appears until the content is fetched and replaced.

In this example, We load a screen with a heading ("My app") but no content. Instead, we show a `<view>` with ID "container" and some "Loading…" text. As soon as this screen

loads, the behavior with `trigger="load"` fires off the `replace` action. It requests content from the `/content` path and replaces the container view with the response.

`visible`

Unlike `load`, the `visible` trigger will only execute the behavior when the element with the behavior is scrolled into the viewport on the mobile device. This allows us to The `visible` action is commonly used to implement an infinite-scroll interaction on a `<list>` of `<item>` elements. The last item in the list includes a behavior with `trigger="visible"`. The `append` action will fetch the next page of items and append them to the list.

`refresh`

This trigger captures a "pull to refresh" action on `<list>` and `<view>` items. This interaction is associated with fetching up-to-date content from the backend. Thus, it's typically paired with an update or reload action to show the latest data on the screen.

```
<body>
  <view scroll="true">
    <behavior trigger="refresh" action="reload" /> ❶
    <text>No items yet</text>
  </view>
</body>
```

❶ When the view is pulled down to refresh, reload the screen

Note that adding a behavior with `trigger="refresh"` to a `<view>` or `<list>` will add the pull-to-refresh interaction to the element, including showing a spinner as the element is pulled down.

`focus`, `blur`, *and* `change`

These triggers are related to interactions with input elements. Thus, they will only trigger behaviors attached to elements like `<text-field>`. `focus` and `blur` will trigger when the user focuses and blurs the input element, respectively. `change` will trigger when the value of the input element changes, like when the user types a letter in a text field. These triggers are often used with behaviors that need to perform some server-side validation on the form fields. For example, when the user types in a username and then blurs the field, a behavior could trigger on `blur` to make a request to the backend and check for uniqueness

of the username. If the entered username is not unique, the response could include an error message letting the user know they need to pick a different username.

Using Multiple Behaviors

Most of the example shown above attach a single `<behavior>` to an element. But there's no such limitation in Hyperview; elements can define multiple behaviors. We already saw an example where a single element had different actions triggered on `press` and `longPress`. But we can also trigger multiple actions on the same trigger.

```
<screen>
  <body>
    <text id="area1">Area 1</text>

    <text>
      <behavior action="hide" target="area1" /> ❶
      <behavior action="hide" target="area2" /> ❷
      Hide
    </text>

    <text id="area2">Area 2</text>
  </body>
</screen>
```

❶ Hide element with ID "area1" when pressed

❷ Hide element with ID "area2" when pressed

In this admittedly contrived example, we want to hide 2 elements on the screen when pressing the "Hide" button. The two elements are far apart in the HXML, and cannot be hidden by hiding a common parent element. But, we can trigger two behaviors at the same time, each one executing a "hide" action but targeting different elements.

3.4. Summary

At the beginning of this chapter, I made a case for developing mobile apps using a Hypermedia architecture. I also explained why using HTML & web views is not the best approach to deliver a native-feeling mobile experience. Hyperview as a framework allow us to develop native mobile apps without giving up the Hypermedia architecture. Unlike HTML, HXML is a mobile-first format with first-class support for the interactions and patterns users expect on mobile. HXML has built-in rich Hypermedia functionality inspired

by htmx and IntercoolerJS. However, HXML takes things a step further with the concept of behaviors and custom elements/actions. By controlling the Hyperview client, developers are free to extend HXML to suit their needs. All this can be done without writing and shipping scripts embedded in HXML. In fact, the concept of scripting doesn't exist in the format! This allows HXML to remain pure and declarative, while the client evolves new capabilites over time.

In the next chapter, we will put the ideas of Hyperview into action, by porting the Contacts App from a web app to a native mobile application.