PEC1. Desarrollo de una web

Introducción

Para el desarrollo de la práctica se ha utilizado el lenguaje de programación ReasonML y Reason React como librería de componentes de interfaz. Por otro lado, se ha utilizado Parceljs como *module bundler*.

ReasonML es un lenguaje, o más bien, una sintaxis diferente para el lenguaje de programación OCaml, esta nueva sintaxis está creada para que sea más atractiva para los desarrolladores con experiencia en JavaScript, y entre otras cosas incluye soporte para JSX. ReasonML se apoya en BuckleScript para poder generar código JavaScript a partir de ReasonML/OCaml.

Configuración del entorno

Para iniciar nuestro proyecto, debemos crear un fichero `package.json`, podemos hacerlo a mano o bien utilizar `npm init` (alternativamente `yarn init`) y seguir el asistente de configuración. Ahora podemos instalar las dependencias, para nuestro proyecto vamos a necesitar `bs-platform` que viene con la librería estandard de `BuckleScript` y otras herramientas como `bsb -init` que permite crear un proyecto `ReasonML` a partir de una plantilla.

Posteriormente instalamos `parcel` con `yarn global add parcel-bundler` o `npm install -g parcel-bundler`, puesto que va a ser la herramienta que vamos a utilizar para empaquetar nuestra aplicación. En la documentación oficial vemos que ofrecen soporte para `ReasonML` por lo que siguiendo los pasos de https://parceljs.org/reasonML.html instalamos `reason-react`, `react-dom` y `react`. Por otro lado, hay que hacer unas pequeñas modificaciones en el fichero `bsconfig.json` puesto que acaba de salir una nueva versión de `Reason React` y la documentación de Parcel aún no está actualizada. Simplemente hay que camabiar la versión de `react-jsx` de la 2 a la 3. Además, también deberemos mantener `bsconfig.json` con las dependencias de librerías ReasonML/BuckleScript que necesitemos, en nuestro caso deberemos añadir `reason-react` y `bs-css` a `bs-dependencies`.

Creamos el archivo `index.html` y `src/Greeting.re` tal y como indica en la documentación de parcel, aunque hay que actualizar `src/Greeting.re` con los cambios de la versión `0.7.0` de `Reason React` que quedaría así:

```
/* src/Greeting.re */

[@react.component]
let make = (~name) =>
  <div> (React.string("Hello! " ++ name)) </div>;
```

y comprobamos que funcione. Para ello, ejecutamos `parcel index.html` y en `locahost:1234` deberíamos de "Hello Parcel".

Una vez que hemos comprobado actualizamos nuestros scripts de `package.json` para que con el comando `npm start` ejecute `parcel index.html` y podamos desarrollar cómodamente. Después de hacer varias pruebas he comprobado que Parcel no realiza *hot reload* de los cambios que realizo en mis ficheros ReasonML, y en el github de Parcel está éste issue donde más gente tiene el mismo problema. Al final pude solucionar el problema con estos scripts en el package.json:

```
    "start": "concurrently -c --kill-others 'npm:watch-html' 'npm:watch-reason'",
    "watch-html": "parcel index.html",
    "watch-reason": "bsb -make-world -w"
  },
```

`concurrently` es otra dependendencia de desarrollo que he tenido que instalar para que todo pudiese funcionar. Para añadir esta dependencia, hay que añadir el flag que indica que es sólo para desarrollo (`npm i --save-dev concurrently` o `yarn add --dev concurrently`).

Por último, para crear la versión de producción se ha añadido el siguiente script a package.json:

```
  "scripts": {
     ...
     "build": "parcel build index.html --public-url /",
     ...
   },
```

Desarrollo

Este es el primer desarrollo frontend que hago con ReasonML, por lo que muchas de las decisiones que he tomado a la hora de estructurar el código, posiblemente no sean correctas. Por otro lado he intentado seguir los patrones que he adquirido de anteriores desarrollos de React y Elm, puesto que mi primer impresión es que Reason React parece una mezcla de estos.

La jerarquía de carpetas ha resultado de la siguiente forma:

```
src
├── Router.re
├── Theme.re
├── Utils.re
├── components # Componentes
|   ├── Component_Article.re
|   ├── Component_ArticleDetailPage.re
|   ├── Component_ArticleListPage.re
|   ├── Component_Breadcrumbs.re
|   ├── Component_Button.re
|   ├── Component_Footer.re
|   ├── Component_Link.re
|   ├── Component_Navbar.re
|   ├── Component_Page.re
|   └── Components.re
├── entities # Entidades
|   ├── Entities.re
|   ├── Entities_Article.re
|   └── Entities_Breadcrumb.re
├── imgs # Imágenes
|   ├── flowtype.svg
|   ├── ...svg
|   └── webpack.svg
├── index.re
├── layout # Páginas
|   ├── Bundlers.re
|   ├── Languages.re
|   ├── Page404.re
|   └── Root.re
└── style.css
```

Routing

`Router.re` es el módulo que controla que componente se renderiza en función de la url y su código es el siguiente:

```
 1 open Components;
 2
 3 [@react.component]
 4 let make = () => {
 5   open Entities;
 6   let url = ReasonReactRouter.useUrl();
 7
 8   let articles = Article.articles;
 9
10   switch (url.path) {
11   | ["module-bundlers"] => <Bundlers />
12   | ["frontend-languages"] => <Languages />
13   | [slug] when Article.existsBySlug(slug, articles) =>
14     let {title, image, content}: Article.t =
15       Article.findBySlug(slug, articles);
16
17     let breadcrumbs: list(Breadcrumb.t) =
18       switch (slug) {
19       | x when Article.existsBySlug(slug, Article.bundlers) => [
20           {path: "module-bundlers", text: "Bundlers"},
21         ]
22       | x when Article.existsBySlug(slug, Article.languages) => [
23           {path: "frontend-languages", text: "Languages"},
24         ]
25       | _ => []
26       };
27
28     <ArticleDetailPage title image content breadcrumbs />;
29   | [] => <Root />
30   | _ => <Page404 />
31   };
32 };
```

Lo que hace es escuchar los cambios de la url (línea 6) y devuelve el componente que se debe renderizar en función de `url.path`. Como se puede observar en la línea 10, se utiliza una expresión `switch` que a diferencia de el switch de JavaScript, aquí se utiliza *pattern matching*.

En este caso, devolverá el componente `<Bundlers />` sólo si `url.path` es una lista de un sólo elemento donde ese elemento es `"module-bundlers"`. Algo similar ocurre con el componente `<Languages />` pero en este caso cuando ese elemento es `"frontend-languages"`.

El siguiente caso es algo más complejo puesto que utiliza una guarda `when`. Lo que ocurre aquí es que captura el único elemento de la lista en la variable `slug` y comprueba si existe algún artículo cuyo slug sea el capturado por la variable, de ser así devuelve `<ArticleDetailPage .../>`.

Si `url.path` es una lista vacía, devolverá el componente `<Root />`.

Por último, si `url.path` es cualquier otra cosa, devolverá `<Page404 />`.

Componentes

En la carpeta `components` se han colocado los componentes de Reason React de UI. Sólo reciben propiedades y renderizan JSX. Un ejemplo sencillo de éstos componentes es `Component_Button.re`. Su código se muestra a continuación:

```
 1 module Styles = {
 2   open Css;
 3
 4   let button =
 5     style([
 6       backgroundColor(hsl(240, 100, 95)),
 7       borderRadius(px(4)),
 8       boxShadow(~y=px(2), ~blur=px(5), hsla(240, 100, 10, 0.1)),
 9       color(Theme.primaryColor),
10       cursor(`pointer),
11       fontFamily(Theme.primaryFontFamily),
12       display(inlineBlock),
13       maxWidth(px(150)),
14       padding2(em(0.5), em(1.0)),
15     ]);
16 };
17
18 let s = React.string;
19
20 [@react.component]
21 let make = (~label, ~className="", ~onClick) =>
22   <div
23     className={Js.Array.joinWith(" ", [|Styles.button, className|])} onClick>
24     {s(label)}
25   </div>;
```

En la línea 20 está la directiva/decorador que indica que esa función es un componente React. Por convención a las funciones constructoras se les llama `make`. En esta función tenemos 3 argumentos con nombre, que son las props de React.

La primera prop es `~label` que es un string y será la etiqueta del botón. La segunda es `~className` y en este caso está inicializada a una cadena vacía, lo que hace que esta prop sea opcional y sirve por si queremos personalizar el botón adicional mediante una clase adicional. Por último, `~onClick` será la función o controlador que se ejecute cuando se pulse el botón; ésta prop también es obligatoria, puesto que no tiene sentido un botón que no hace nada.

Por encima de la línea 20, tenemos un módulo llamado `Styles`. Ese módulo contiene los estilos de nuestro botón. Para ello hemos usado `bs-css` una librería para escribir CSS con ReasonML. Para los

estilos se ha seguido un enfoque de *mobile first* pero apenas se han tenido que utilizar *media queries* puesto que el diseño realizado es muy simple y se adapta fácilmente tanto a móvil como a navegador de escritorio.

El resto de componentes de la aplicación son similares, por lo que no merece la pena entrar en detalle en otros componentes.

Entidades

En la carpeta `entities` hemos guardado los tipos de datos y las funciones para manipular esos datos. El principal tipo de dato de esta práctica es el `Article` que tiene un título, un *slug*, imagen y contenido:

```
 1 type t = {
 2   slug: string,
 3   title: string,
 4   content: string,
 5   image: string,
 6 };
 7
 8 let existsBySlug = (slug: string) =>
 9   List.exists((article: t) => slug == article.slug);
10
11 let findBySlug = (slug: string) =>
12   List.find((article: t) => slug == article.slug);
```

Las operaciones que se han creado han sido `existsBySlug` y `findBySlug`. El primero comprueba si existe un artículo dado un *slug*, y el segundo busca un artículo dado un *slug*.

Por último, en ese fichero se han creado dos listas de artículos, artículos sobre lenguajes de programación (`languages`), y artículos sobre *bundlers* (`bundlers`). Para un futuro ha quedado procesar los artículos desde archivos de Markdown con un *loader* de parcel como https://github.com/The-Politico/parcel-plugin-react-markdown.

Ejecución del código y website de prueba

Para ejecutar el proyecto, necesitamos instalar las dependencias con `npm install` y posteriormente ejecutar `npm start` .

Por último, el resultado de esta práctica se puede ver en la url uoc-reason-frontend.now.sh.