React Workshop - RxJS

Authors:

- Thomas Burleson (mailto:thomas.burleson@ampf.com), Solutions Architect
- Harry Beckwith (mailto:harry.beckwith@ampf.com), Solutions Architect

RxJS Code Labs

Lab 1: Convert Promises to Observables

In the Jumpstart code, we used `Promise<T>` for our asynchronous solutions. While better than the raw EventListener API, Promises have limited features as we recall below:

Data Pull	Event Listeners	Promises	Observables
Single-Event	✔	✔	✔
Multi-Event	✔	—	✔
Chain	—	✔	✔
Object Inst	—	✔	✔
Sync	✔	—	✔
Async	—	✔	✔
Cancellable	✔	—	✔
Lazy	—	—	✔

Now that we have learned about Observables, let's convert our code to use Observables. We will also use RxJS operators to transform our data.

Tasks

1. Convert the `ContactsService` to use Observables.
2. Update the `ContactList` React component to use the new Observable API in `ContactsService`.
3. Update the `ContactDetail` React component to use the new Observable API in `ContactsService`.

Code Snippets

`libs/contacts/data-access/src/lib/contacts.service.ts`

```
export class ContactsService {
  constructor(private apiEndPoint: string, private apiKey: string) {}

  /**
   * Load a list from the REST service...
   */
  getContacts(useCache = false, params: QueryParams = null): Observable<Contact[]> {
    const request$ = new Observable<Contact[]>(subscriber => {
      let shouldNotify = true;
      params = params || ({} as QueryParams);
      this.loadContacts(params)
        .then(list => {
          if (shouldNotify) {
            subscriber.next((contacts = assignUIDs(list)));
            subscriber.complete();
          }
        })
        .catch(subscriber.error);
      return () => {
        shouldNotify = false;
      };
    });

    // Use cached list
    return contacts.length && useCache ? of(contacts) : request$;
  }

  getContactById(id: string): Observable<Contact | undefined> {
    const who = !!contacts
      ? contacts.reduce((result, it) => {
          return result || (it.id === id ? it : null);
        }, null)
      : null;
    return of(who);
  }

  /**
   * Case-insenstive, partial criteria matching...
   * @param userName
   * @param title
   */
  searchBy(userName: string, title: string = ''): Observable<Contact[]> {
    const filterByName = (who: Contact): boolean => (userName ? contains(who.name, userName) : true);
    const filterByTitle = (who: Contact): boolean => (title ? contains(who.position, title) : true);

    return this.getContacts(false, { userName, title }).pipe(
      map(list => list.filter(filterByName)),
      map(list => list.filter(filterByTitle))
    );
  }
}
```

(https://i.imgur.com/Ay6PP9x.png)

libs/contacts/ui/src/lib/contacts-list.tsx

```
export const ContactsList: React.FC = () => {
  const [service]           = useState<ContactsService>(injector.get(ContactsService));
  const [people, setPeople] = useState<Contact[]>([]);
  const doSearch = (e: Event) => {
    const criteria = (e.target as HTMLIonInputElement).value;
    const request$ = service.searchBy(criteria);
    request$.subscribe(setPeople);
  };

  useEffect(() => {
    const allContacts$ = service.getContacts();
    allContacts$.subscribe(setPeople);
  }, [service, setPeople]);

  return (...);
}
```

()

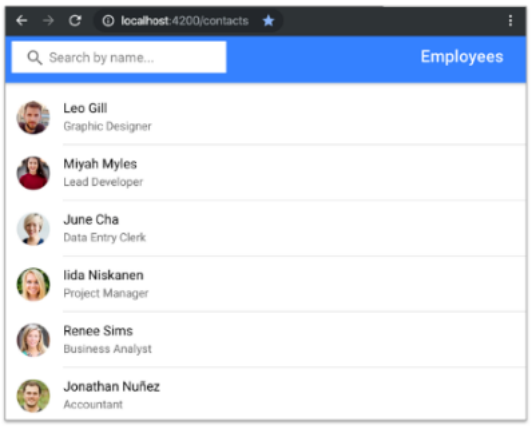
libs/contacts/ui/src/lib/contact-detail.tsx

```
export const ContactDetails: React.FC<ContactDetailProps> = () => {
  const { id }    = useParams();
  const history   = useHistory();
  const [service] = useState<ContactsService>(injector.get(ContactsService));
  const [contact, setContact] = useState<Contact>({} as Contact);

  // Use Router param `id` to lookup
  useEffect(() => {
    service.getContactById(id).subscribe(setContact);
  }, [id, service, setContact]);

  // 'Esc' keyboard shortcut to close the popup
  useEffect(() => {
    const key = 'esc';
    Mousetrap.bind([key], () => history.goBack());
    return () => Mousetrap.unbind([key]);
  }, [history]);

  return (...);
}
```

().

Lab 2: Debounce and deduplicate terms

The goal of this exercise is to fine-tune the existing implementation of our instant search to leverage the power of observables. We will debounce and deduplicate the user supplied search terms in order to reduce the pressure on the server and increase scalability.



We need to stream the '**search by name**' input control changes through an Observable... so that we can use the power of the `debounce` and `distinctUntilChanged` operator to reduce the number of requests made to the server.

In ContactsList component, create a `Subject` instance that will be used from within the template. As the input control value changes, we will push the search term values into the Observable stream .

Tasks

1. Create an `emitter` Subject in `ContactsList` that can be used from the template to propagate the input changes

- import `Subject` from `rxjs`
- create an `emitter` state that is an instance of `new Subject()`
- Update the `doSearch()`` event handler to call emitter.next($event.target.value)`

> Make sure to import `debounceTime` and `distinctUntilChanged` RxJS operators.

2. Since `emitter` will be used to push the search criteria through a stream, we want to use the RxJS operators `debounceTime(400)` and `distinctUntilChanged()` to reduce the number of requests made to the server.
3. Inside the `useEffect()` , create stream references: `allContacts$` , `searchTerm$` . Listen for data emissions on both of these streams.

Code Snippets

`contacts-list.tsx`

```tsx
import { Subject } from 'rxjs';
import { debounceTime, distinctUntilChanged } from 'rxjs/operators';

import { ContactListItem } from './contact-item';
import { injector, ContactsService, Contact } from '@workshop/contacts/data-access';

export const ContactsList: React.FC = () => {
  const [service]           = useState<ContactsService>(injector.get(ContactsService));
  const [emitter]           = useState(new Subject<string>());
  const [people, setPeople] = useState<Contact[]>([]);



  return ( ... );
};
```

```tsx
import { Subject } from 'rxjs';
import { debounceTime, distinctUntilChanged } from 'rxjs/operators';

import { ContactListItem } from './contact-item';
import { injector, ContactsService, Contact } from '@workshop/contacts/data-access';

export const ContactsList: React.FC = () => {
  const [service]           = useState<ContactsService>(injector.get(ContactsService));
  const [emitter]           = useState(new Subject<string>());
  const [people, setPeople] = useState<Contact[]>([]);

  const doSearch = (e: Event) => {
    const criteria = (e.target as HTMLIonInputElement).value;
    emitter.next(criteria);
  };

  useEffect(() => {
    const allContacts$ = service.getContacts();
    const searchTerm$ = emitter.asObservable().pipe(
      debounceTime(250),
      distinctUntilChanged()
    );

    allContacts$.subscribe(list => setPeople(list));
    searchTerm$.subscribe(term => {
      const search$ = service.searchBy(term);
      search$.subscribe(setPeople);
    });
  }, [service, setPeople]);

  return (...);
};
```

Be prepared to talk about any issues with this approach.

Lab 3:   Out-of-Order Responses & Zombies

Asynchronous requests are tricky.

- How do we guarantee that the response is recieved for a specific request?
- How do we cancel a current ṿḠÖ̃ǨệÖ̃ḅ request; in order to issue a new request?
- How do we queue a batch set of requests?

Even more important is: “How do we avoid nested `subscribe()` calls?”

```
searchTerm$.subscribe(term => {
  const search$ = service.searchBy(term);

  search$.subscribe(list => {
      setPeople(list);
  });

});
```

> We must use the powerful RxJS operator `switchMap()` to both
> - avoid nested subscribes, and
> - cancel possible inflight requests.

Combining Streams

We actually have two (2) data streams: `allContacts$` and `searchTerm$` . We need to use the RxJS `merge()` to combine both streams into a single stream.

If the user starts searching BEFORE the allContacts$ loads, we need to cancel that request to load all contacts… so we see only the results for the searched contacts.

Tasks

1. Update the `ContactsList` component to `merge` both streams, use `switchMap` to cancel inflight requests, and use `takeUntil` to cancel the default load (if needed).

```tsx
export const ContactsList: React.FC = () => {
  const [service]           = useState<ContactsService>(injector.get(ContactsService));
  const [emitter]           = useState<Subject<string>>(new Subject<string>());
  const [people, setPeople] = useState<Contact[]>([]);

  const doSearch = (e: Event) => {
    const criteria = (e.target as HTMLIonInputElement).value;
    emitter.next(criteria);
  };

  useEffect(() => {
    const term$        = emitter.asObservable();
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));
    const searchTerm$  = term$.pipe(
      debounceTime(250),
      distinctUntilChanged(),
      switchMap(criteria => service.searchBy(criteria))
    );
    const watch = merge(allContacts$, searchTerm$).subscribe(setPeople);

    return () => watch.unsubscribe();
  }, [service, setPeople]);

  return (...);
};
```

(https://i.imgur.com/xu51FQe.png)

Hint: Use three (3) separate variables for the (3) streams.

Lab 4: Refactor Stream Logic into Search API

Currently our view component `ContactsList` has code to `debounce`, `distinctUntilChanged`, and `switchMap`.

> This is logic that should be refactored to the business layer.

Let's update our `ContactsService` API to handle **input** streams.

Tasks

1. Update `ContactsService` to add a new method `autoSearch()`
2. Update `ContactsList` to use the new `ContactsService::autoSearch()` API.

Code Snippets

libs/contacts/data-access/src/lib/contacts.service.ts

```
/**
 * Watch input stream, apply biz rules and then
 * search for Contacts with partial name terms
 * @param term$
 * @param debounceMs
 */
autoSearch(
  term$: Observable<string>,
  debounceMs = 250
): Observable<Contact[]> {

  return term$.pipe(
    debounceTime(debounceMs),
    distinctUntilChanged(),
    switchMap(term => this.searchBy(term))
  );

}
```

libs/contacts/ui/src/lib/contacts.list.tsx

```
export const ContactsList: React.FC = () => {

  ...

  useEffect(() => {
    const term$ = emitter.asObservable();
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));
    const searchTerm$ = service.autoSearch(term$);
    const watch = merge(allContacts$, searchTerm$).subscribe(setPeople);

    return () => watch.unsubscribe();
  }, [service, setPeople]);

  return (...);
};
```

Lab 5: Using `useObservable()`

Subscriptions are used to **extract** data that is emitted through an Observable stream. Yet, one the more challenging parts of Observables is that of managing subscriptions.

> The goal is to avoid explicit `subscribe()` calls as much as possible.

@mindspace-io/react publishes a custom hook `useObservable(stream)` that:

- auto subscribes to `stream`
- publishes (in a tuple) both the emitted data AND setter function to watch a new observable (if needed)
- auto-unsubscribes from an existing stream before subscribing to the new stream
- auto-unsubscribes when the host view component unmounts

> Let's use this new custom hook `useObservable()` to simplify our code.

Tasks

1. Import `useObservable` from the `@mindspace-io/react` package/library.
2. Use the hook to create/manage the `[people, setPeople$]` tuple.
3. Use the hook to manage the `[criteria, setCriteria$]` tuple.
4. Update `useEffect()` to use both setter functions (which set Observables... not raw data).

Code Snippets

```
import { useObservable } from '@mindspace-io/react';

export const ContactsList: React.FC = () => {
  const [emitter] = useState<Subject<string>>(new Subject<string>());
  const [service] = useState<ContactsService>(injector.get(ContactsService));

  const doSearch = (e: Event) => emitter.next((e.target as HTMLIonInputElement).value);

  const [people,   setPeople$]   = useObservable<Contact[]>(null, []);
  const [criteria, setCriteria$] = useObservable<string>(null, '');

  useEffect(() => {
    const term$        = emitter.asObservable();
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));
    const searchTerm$  = service.autoSearch(term$);

    setCriteria$(term$);
    setPeople$(merge(allContacts$, searchTerm$));

  }, [service, setPeople$, setCriteria$] );

  return (...);
};
```

Lab 6: Facades as View Models

Facades allow developers to

- hide complexity
- publish simple API + data models to views
- support 1-way data flows to-from the UI layers

Let's create a `ContactsFacade` that supports the following API:

```
export class ContactsFacade {
  readonly contacts$: Observable<Contact[]>;
  readonly criteria$: Observable<string>;

  constructor(private service: ContactsService) { }

  /**
   * Search changes emits from `criteria$` which then triggers
   * the ContactsService to requery the API based on partial user name
   * matches.
   */
  searchFor(partial: string): Observable<Contact[]> { }

  /**
   * Select contact by ID
   */
  selectById(id: string): Observable<Contact | undefined> { }
}
```

> Notice that the `ContactsFacade` has an instance of the ContactsService injected into the constructor via DI

The API maintains the circular, 1-way data flow:

- Data flows outputs (from the facade) are ONLY via streams.
- Inputs are ONLY methods into the Facade

Tasks

1. Implement the `ContactsFacade` at `libs/contacts/data-access/src/lib/contacts.facade.ts`
2. Ensure you export the Facade from the library public API (see `libs/contacts/data-acess/src/index.ts`)
3. Register the `ContactsFacade` class with the dependency injection (DI) engine (see `libs/contacts/data-access/src/lib/contacts.injector.ts`)
4. Update the `ContactsList` view component to use the `ContactsFacade` instead of the `ContactsService`
5. Update the `ContactDetail` view component to use the `ContactsFacade` instead of the `ContactsService`

Code Snippets

`libs/contacts/data-access/src/lib/contacts.facade.ts`

```ts
import { Observable, merge, Subject } from 'rxjs';

import { Contact } from '@workshop/shared/api';

export class ContactsFacade {
  private emitter: Subject<string>;

  readonly contacts$: Observable<Contact[]>;
  readonly criteria$: Observable<string>;

  constructor(private service: ContactsService) {
    const emitter = new Subject<string>();
    const term$ = emitter.asObservable();
    const searchByCriteria$ = service.autoSearch(term$);
    const allContacts$ = service.getContacts().pipe(takeUntil(term$));

    this.emitter = emitter;
    this.criteria$ = term$;
    this.contacts$ = merge(searchByCriteria$, allContacts$);
  }

  /**
   * Search changes emits from `criteria$` which then triggers
   * the ContactsService to requery the API based on partial user name
   * matches.
   *
   * It also kills a full-load if that request is still pending.
   */
  searchFor(partial: string): Observable<Contact[]> {
    this.emitter.next(partial);
    return this.contacts$;
  }

  selectById(id: string): Observable<Contact | undefined> {
    return this.service.getContactById(id);
  }
}
```

libs/contacts/data-access/src/lib/contacts.injector.ts

```ts
import { makeInjector, DependencyInjector } from '@mindspace-io/react';
import { ContactsService, API_ENDPOINT, API_KEY } from './contacts.service';
import { ContactsFacade } from './contacts.facade';

/**
 * Create a DependencyInjector for Contacts features: service + facade
 */
export const injector: DependencyInjector = makeInjector([
  { provide: API_KEY,        useValue: '873771d7760b846d51d025ac5804ab' },
  { provide: API_ENDPOINT,    useValue: 'https://uifaces.co/api?limit=25' },
  { provide: ContactsService, useClass: ContactsService, deps: [API_ENDPOINT, API_KEY] },
  { provide: ContactsFacade,  useClass: ContactsFacade,  deps: [ContactsService] }
]);
```

libs/contacts/ui/src/lib/contacts-list.tsx

```tsx
import { useObservable } from '@mindspace-io/react';
import { injector, Contact, ContactsFacade } from '@workshop/contacts/data-access';
import { ContactListItem } from './contact-item';

export const ContactsList: React.FC = () => {
  const facade: ContactsFacade = injector.get(ContactsFacade);

  const [criteria] = useObservable<string>(facade.criteria$, '');
  const [people]   = useObservable<Contact[]>(facade.contacts$, []);

  const doSearch = (e: Event) => {
    const criteria = (e.target as HTMLIonInputElement).value;
    facade.searchFor(criteria);
  };

  return (...);
}
```

libs/contacts/ui/src/lib/contact-detail.tsx

```tsx
import { useObservable } from '@mindspace-io/react';
import { injector, Contact, ContactsFacade } from '@workshop/contacts/data-access';
import { ContactListItem } from './contact-item';


export const ContactDetails: React.FC<ContactDetailProps> = () => {
  const history                = useHistory();
  const { id }                 = useParams();
  const [contact, setContact$] = useObservable<Contact>(null, {} as Contact);

  const facade: ContactsFacade = injector.get(ContactsFacade);

  useEffect(() => {
    setContact$(facade.selectById(id));
  }, [facade, id, setContact$]);

  return (...);
}
```

Lab 7: Custom Hooks with Facades

In Jumpstart, we learned how custom hooks can also hide complexity.
Let's use custom hooks to hide our usages of the `ContactsFacade` and the `useObservable()` hook itself.

Since each custom hook is used only for a specific view component, we can define the API for each hook:

- `useContacts(): [string, Contact[], ContactsFacade]`
- `useContactDetails(): [Contact, { goBack: ()=>void }]`

> We design the hook tuples to only supply the data and functions needed by the view component!
> Notice the `useContactsTuple` returns 3 elements.

Tasks

1. Create the custom hooks in `libs/contacts/data-access/src/lib/contacts.hook.ts`
2. Update the public API in `libs/contacts/data-access/src/index.ts`
3. Refactor `ContactsList` to use the `useContacts()` hook.
4. Refactor `ContactDetail` to use the `useContactDetails` hook.

> Be sure to clean up your imports!

Code Snippets

`libs/contacts/data-access/src/lib/contacts.hook.ts`

```
import { useHistory } from 'react-router-dom';
import { useParams } from 'react-router';
import { useState, useEffect } from 'react';
import { useObservable } from '@mindspace-io/react';

import { Contact } from '@workshop/shared/api';
import { injector } from './contacts.injector';
import { ContactsFacade } from './contacts.facade';

export type ContactsTuple = [string, Contact[], ContactsFacade];
export type ContactDetailsTuple = [Contact, { goBack: () => void }];

export function useContacts(): ContactsTuple {
  const [facade] = useState<ContactsFacade>(() => injector.get(ContactsFacade));
  const [criteria] = useObservable(facade.criteria$, '');
  const [contacts] = useObservable(facade.contacts$, []);

  return [criteria, contacts, facade];
}

export function useContactDetails(): ContactDetailsTuple {
  const { id } = useParams();
  const [facade] = useState<ContactsFacade>(() => injector.get(ContactsFacade));
  const [contact, setContact$] = useObservable<Contact>(null, {} as Contact);
  const history = useHistory();

  useEffect(() => {
    setContact$(facade.selectById(id));
  }, [id, facade]);

  return [contact, history];
}
```

libs/contacts/ui/src/lib/contacts-list.tsx

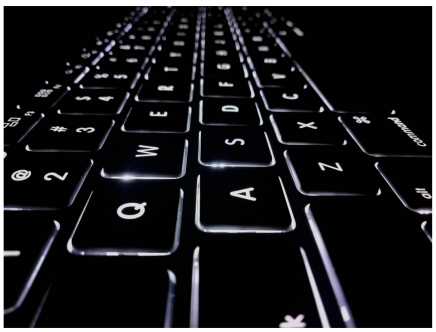
```
export const ContactsList: React.FC = () => {
  const [criteria, people, facade] = useContacts();
  const doSearch = (e) => {
    facade.searchFor((e.target as HTMLIonInputElement).value);
  }

    return (
    <IonPage>
      <IonHeader>
        <IonToolbar>
          <IonItem style={inlineItem}>
            <IonIcon icon={search}></IonIcon>
            <IonInput
              clearInput
              autofocus
              value={criteria}
              onIonChange={doSearch}>
            </IonInput>
}
```

libs/contacts/ui/src/lib/contact-detail.tsx

```
export interface ContactDetailProps extends RouteComponentProps<{ id: string }> {}

export const ContactDetails: React.FC<ContactDetailProps> = () => {
  const [contact, history] = useContactDetails();

  useEffect(() => {
    const key = 'esc';
    // 'Esc' keyboard shortcut to close the popup
    Mousetrap.bind([key], () => history.goBack());

    return () => Mousetrap.unbind([key]);
  }, [history]);
```

Lab 8: MouseTrap Custom Hook

Let's implement another custom hook that enables versatile re-use of MouseTrap shortcut key handler.



Tasks

1. Implement the custom hook in `useMouseTrap()`
2. Update the `ContactDetails` view to use this hook.

Code Snippets

libs/contacts/data-access/src/lib/contacts.hook.ts

```
/**
 * Support global keyboard shortcuts...
 * auto-cleanup when view unmounts
 */
export function useMouseTrap(
     shortcut: string[] | string,
     handler : () => void,
     deps    : DependencyList = [])
{
  const memoFn = useCallback(handler, deps);

  useEffect(() => {
    Mousetrap.bind(shortcut, memoFn);
    return () => Mousetrap.unbind(shortcut);
  }, [memoFn]);
}
```

> Be sure prepared to talk about the best location to publish this hook.

libs/contacts/ui/src/lib/contact-detail.tsx

```
export const ContactDetails: React.FC<ContactDetailProps> = () => {
  const [contact, navigate] = useContactDetails();
  const history = useHistory();

  // 'Esc' keyboard shortcut to close the popup
  useMouseTrap('esc', () => navigate.goBack(), [history.location]);

  return (...);
}
```

Thanks to Jimmy Guzman for this excellent enhancement and contribution!