React Workshop - Jumpstart

Authors:

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

Jumpstart Code Labs

Lab (1): Add TypeScript Types

Without types developers must infer the proper usages of the ContactsService.

The data returned - while assumed to be Contact[] - is ambiguous. Neither the IDE nor the compiler enforce that assumption.

Scenario

The current ContactsService class does not specify any types.

Let's add type information to all variables, function parameters, and function return values.

Tasks

1. Add type information to the JSON array of contacts
2. Add types to the `ContactsService` in `apps/contacts/src/app/services/contacts.service.ts` .

- Add `QueryParams` and `QueryOptions` interfaces for use in the `fetchContacts` method
- Add parameter and return types to all functions.
- Add types to `filterByName` and `filterByTitle`

Code Snippets

apps/contacts/src/app/services/data/contacts.ts

```
import { Contact } from '../contacts.model';

export const CONTACTS: Contact[] = [
    // ....
];
```

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

apps/contacts/src/app/services/contacts.service.ts

```
import uuid from 'react-uuid';
import { Contact } from './contacts.model';
import { CONTACTS } from './data/contacts';

let contacts: Contact[] = [];

export interface QueryParams  { userName: string;  title?: string; }
export interface QueryOptions { apiKey: string;  apiEndPoint: string; }

function fetchContacts({ userName }: QueryParams, options: QueryOptions): Promise<Contact[]> {
    // ...
}

export class ContactsService {
  private apiEndPoint = API_ENDPOINT;
  private apiKey = API_KEY;

  /**
   * Load a list from the REST service...
   */
  getContacts(useCache = false, params: QueryParams = null): Promise<Contact[]> {
    //...
  }

  getContactById(id: string): Promise<Contact | null> {
    //...
  }

  searchBy(userName: string, title: string = ''): Promise<Contact[]> {
    const filterByName = (who: Contact): boolean => (userName ? contains(who.name, userName) : true);
    const filterByTitle = (who: Contact): boolean => (title ? contains(who.position, title) : true);

    //....
  }

  private loadContacts(params: QueryParams): Promise<Contact[]> {
    const { apiEndPoint, apiKey } = this;
    return fetchContacts(params, { apiEndPoint, apiKey });
  }
}


function assignUIDs(list: Contact[]): Contact[] {
  return list.map(it => {
    it.id = uuid();
    return it;
  });
}


function contains(target: string, partial: string): boolean {
  return target.toLowerCase().indexOf(partial.toLowerCase()) > -1;
}
```

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

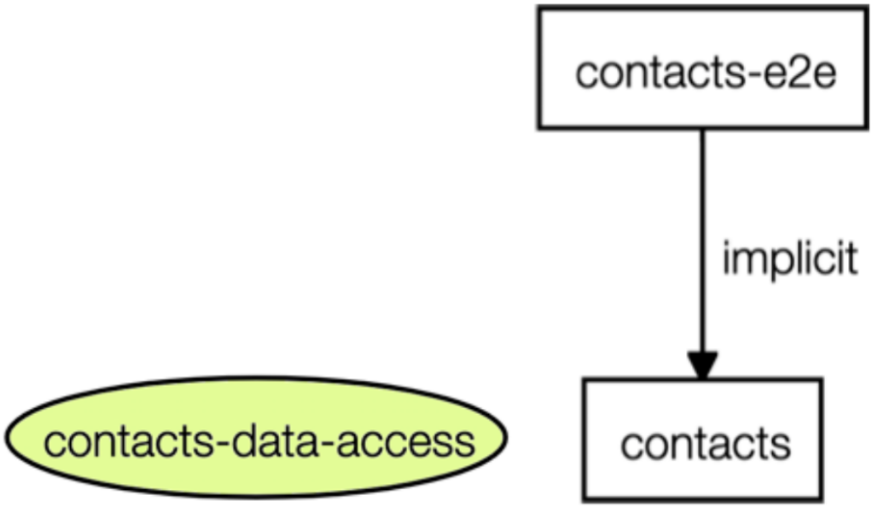
Lab (2a): Create Data-Access Library

The ContactsService and models are not encapsulated in a distinct, protected library. Such library can

- be reused and imported by 1...n UI libraries
- defines a public API that protects access to non-public artifacts.

Scenario

Let's create a Data-Access library in `/libs/contacts/data-access`.



Tasks

1. Use a terminal to run the command

```
nx g @nrwl/web:lib data-access \
      --directory=contacts \
     --tags="scope:contacts, type:data-access"  \
     --dryRun
```

2. make sure the `.eslintrc` has the `enforce-module-boundaries` settings shown below
3. move the contact services files

- from `apps/contacts/src/app/services/*/*`
- to `/libs/contacts/data-access/src/lib/*/*`

5. update the library public API to export the `Contact` interface and `ContactsService` service.
6. clean up the Contacts app by remove the references to `ContactsService`

> What else did the schematic do?
> Review the changes to `nx.json`, `tsconfig.json`, and `workspace.json`

Code Snippets

`.eslintrc` (partial)

Lab (2b): Create Shared-API Library

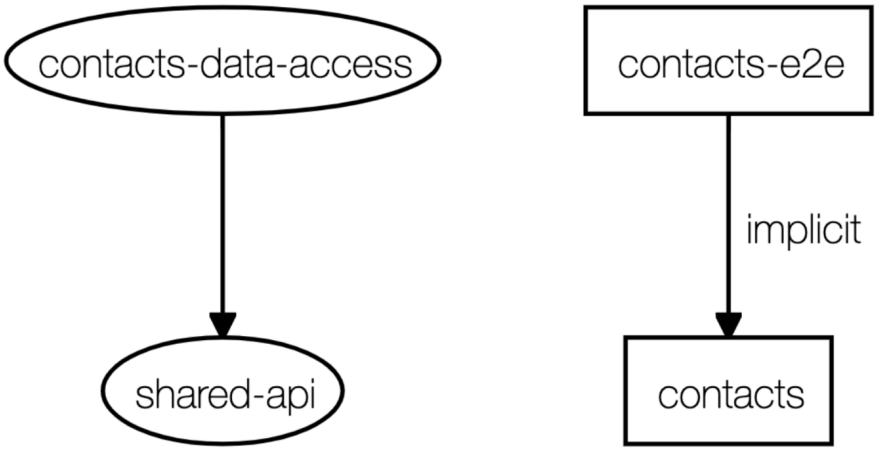
The `Contacts` model and other interfaces could be used by 1...n web apps and 1...n server apps...
Creating a distinct API typescript library allows the models/interfaces to be shared.

Scenario

Let's create a TypeScript library in `libs/shared/api` using the Nx tools.

Then we will:

- move the `Contact` interface into that library,
- define a public API (in the `index.ts`),
- enforce public API boundaries (`.eslintrc`)
- update the Data-Access library to import the `Contact` from the Shared-API library.



Tasks

1. Use a terminal to run the command

```
nx g @nrwl/web:lib api \
      --directory=shared \
     --tags="scope:contacts, type:data-access"  \
     --dryRun
```

3. move the **Contact** interface file
    - from `/libs/contacts/data-access/src/lib/contact.model.ts`
    - to `/libs/shared/api/src/lib/contacts/contact.model.ts`
4. update the ĎĐ vĞĶǫ! Ćl library public API to export the `Contact` interface.
5. update the Zl Ãǫ! ùùĞŸŸ library public API to re-export the `Contact` interface.
6. Update the imports of the **Contact** interface to use the new ĎĐ vĞĶǫ! Ćùlibrary
    - from `import { Contact } from './contacts.model';`
    - to `import { Contact } from '@workshop/shared/api';`

> Be prepared to talk about why `export * from '@workshop/shared/api'` was used.

Code Snippets

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

```
export * from '@workshop/shared/api';
export * from './lib/contacts.service';
```

libs/shared/api/src/lib/shared-api.ts

```
export * from './contacts/contacts.model';
```

Lab (3a): Create UI Library

Let's use the `@ionic/react` component library and create custom feature library for Contacts.
This library contains the business UI features / workflow for contacts: List, ListItem, and Details.

Scenario

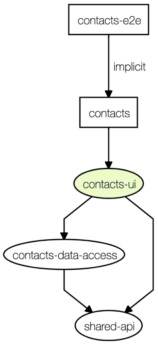
> Use Nx tools to create a React UI library at `libs/contacts/ui` .

This library will

- contain the React UI components ContactsList and ContactItem,
- import and use the `@workspace/contacts/data-access` library,
- will be constrained so only App can import this library.

apps/contacts/src/app/**app.tsx**

```
1 import { ContactsList } from '@workshop/contacts/ui';
```



Tasks

1. Use a terminal to run the command

```
nx g @nrwl/react:lib ui \
      --directory=contacts \
     --tags="scope:contacts, type:feature"  \
     --dryRun
```

3. Create a UI library style definitions in `libs/contacts/ui/src/lib/styles.ts`
4. Create a React Component **ContactsList**
5. Create a React Component **ContactItem**
6. update the Į ùlibrary public API to export the `ContactsList` view component.
7. Update the Contacts app to import and use the ContactsList
8. Remove the `body { background-color:...}` in `apps/contacts/src/app/app.scss`

> Do you know what a feature library is?
> Be prepared to talk about `@workshop/contacts/ui` ... where did this come from?

Code Snippets

libs/contacts/ui/src/lib/styles.ts

```ts
export const inlineItem = {
  '--min-height': '20px',
  '--border-radius': '5px',
  display: 'inline-block',
  paddingLeft: '10px'
} as React.CSSProperties;

export const stickyRight = {
  position: 'absolute',
  right: '10px',
  top: '12px'
} as React.CSSProperties;

export const iconOnLeft = {
  '--padding-start': '5px'
};

export const gridItem = {
  display: 'flex',
  alignItems: 'center',
  justifyContent: 'center'
} as React.CSSProperties;
```

(https://bit.ly/2Vk7yBA)

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

```tsx
import React, { Component } from 'react';
import * from '@ionic/react';
import { search } from 'ionicons/icons';

import { Contact } from '@workshop/shared/api';
import { ContactsService } from '@workshop/contacts/data-access';

import { inlineItem, iconOnLeft, stickyRight } from './styles';
import { ContactListItem } from './contact-item';

interface ContactsState { people: Contact[] };

export class ContactsList extends Component<{}, ContactsState> {
  private service = new ContactsService();

  constructor(props) {
    super(props);
    this.state = { people: [] };
  }

  componentDidMount() {
    this.service.getContacts().then(list => {
      this.setState({ people: list });
    });
  }

  render() {
    const doSearch = criteria => {
      this.service.searchBy(criteria).then(list => {
        this.setState({ people: list });
      });
    };

    return (
      <IonPage>
        <IonHeader>
          <IonToolbar>
            <IonItem style={inlineItem}>
              <IonIcon icon={search}></IonIcon>
              <IonInput
                clearInput
                autofocus
                style={iconOnLeft}
                onIonChange={e => doSearch((e.target as HTMLIonInputElement).value)}
                placeholder="Search by name..."
              ></IonInput>
            </IonItem>
            <IonTitle style={stickyRight}> Employees </IonTitle>
          </IonToolbar>
        </IonHeader>
        <IonContent>
          <IonList>
            {this.state.people.map((person, idx) => {
              return <ContactListItem key={idx} person={person} />;
            })}
          </IonList>
        </IonContent>
      </IonPage>
    );
  }
}
```

(https://gist.github.com/ThomasBurleson/92bd34c3317bf4c22b1b6d2d2b946ff3)

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

```tsx
import React from 'react';
import { IonAvatar, IonItem, IonLabel } from '@ionic/react';
import { Contact } from '@workshop/shared/api';

interface ListItemProps {
  person: Contact;
  isButton?: boolean;
}

export const ContactListItem: React.FC<ListItemProps> = ({ person }) => {
  return (
    <IonItem href="/">
      <IonAvatar slot="start">
        <img src={person.photo} />
      </IonAvatar>
      <IonLabel>
        <h2>{person.name}</h2>
        <p>{person.position}</p>
      </IonLabel>
    </IonItem>
  );
};
```

apps/contacts/src/app/app.tsx

```tsx
import React from 'react';
import { ContactsList } from '@workshop/contacts/ui';

import './app.scss';

export const App = () => {
  return <ContactsList />;
};
```

Lab (3b): Add UI Routing

For our application we want to navigate between the master (`ContactsList`) and detail (`ContactDetail`) views. This routing will also require us to use the Router param information to dynamically lookup the contact information.

Scenario

Let's use the `@ionic/react-router` library (easily replaced btw with React Router) to route between the `ContactsList` and the `ContactDetail` views.

In the Qŕ ÖÂ ùÄŽĠÄ ę use the Router param `id` to dynamically lookup the contact information and render the detail view.

- Add the `ContactDetail` view component to the `libs/contacts/ui` library,
- Add a `routerLink` option to the `ContactItem` (in the ContactsList),
- Update the public API for the `@workspace/contacts/ui`
- Add routing to the Contact app component in `apps/contacts/src/app.tsx`

Tasks

3. Create a React style sheet `contact-details.scss`
4. Create a React Class Component `ContactDetail`
5. Modify the ContactItem to use a `routerLink`
6. Update the UI library public API to export `ContactDetail`
7. Setup routing in the Contact App

> Notice how lifecycle methods used in the `ContactDetail` component.

Code Snippets

`libs/contacts/ui/src/lib/contact-detail.scss`

```
.contactDetails {
  ion-card-content {
    width: 300px;
    height: 240px;
    padding-left: 80px;
  }
  ion-avatar {
    width: 128px;
    height: 128px;
  }
  ion-label {
    padding-left: 10px;
    padding-top: 15px;
    display: block;
    h2 {
      font-weight: bold;
    }
  }
  ion-button {
    margin-bottom: 10px;
    margin-top: 10px;
    margin-left: 210px;
  }
}
```

(https://bit.ly/2VgnSDs)

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

```
import React, { Component } from 'react';
import { RouteComponentProps, Redirect } from 'react-router';
import { IonPage, IonFooter, IonButton, IonCard, IonCardContent, IonAvatar, IonLabel } from
'@ionic/react';

import './contact-detail.scss';
import { gridItem } from './styles';

import { Contact, ContactsService } from '@workshop/contacts/data-access';

export interface ContactDetailProps extends RouteComponentProps<{ id: string }> {}
export interface ContactDetailState {
  contact: Contact;
}

export class ContactDetails extends Component<ContactDetailProps, ContactDetailState> {
  constructor(props) {
    super(props);
    this.state = { contact: {} as Contact };
  }

  componentDidMount() {
    this.loadContact();
  }

  componentDidUpdate() {
    // Typical usage (don't forget to compare props):
    const newId = this.props.match.params.id;
    const oldId = this.state.contact ? this.state.contact.id : '';

    if (newId !== oldId) {
      this.loadContact(newId);
    }
  }

  private loadContact(id = '') {
    // Use Router param `id` to lookup
    const service = new ContactsService();
    const params = this.props.match.params;
    const request = service.getContactById(id || params.id);

    request.then(contact => this.setState({ contact }));
  }

  render() {
    const contact = this.state.contact;

    return contact ? (...) : ( <Redirect to="/contacts" /> );
  }
}
@ThomasBurleson
```

(https://gist.github.com/ThomasBurleson/87a92e6742cfa93b52fc728ffa1d0bab)

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

```
1
2  interface ListItemProps { person: Contact; }
3
4  export const ContactListItem: React.FC<ListItemProps> = ({ person }) => {
5    const url = `/contacts/${person.id}`;
6    return (
7      <IonItem routerLink={url}>...</IonItem>
8    );
9  }
```

libs/contacts/ui/src/index.tsx

```
export * from './lib/contacts-list';
export * from './lib/contact-item';
export * from './lib/contact-detail';
```

apps/contacts/src/app/app.tsx

```
<IonApp>
  <IonReactRouter>
    <IonRouterOutlet>

      <Route path="/contacts"      exact component={ContactsList}   />
      <Route path="/contacts/:id" exact component={ContactDetails} />

      <Redirect exact from="/" to="/contacts" />

    </IonRouterOutlet>
  </IonReactRouter>
</IonApp>
```

Lab (4): Use Functional Components

Facebook recommends using Functional Components.

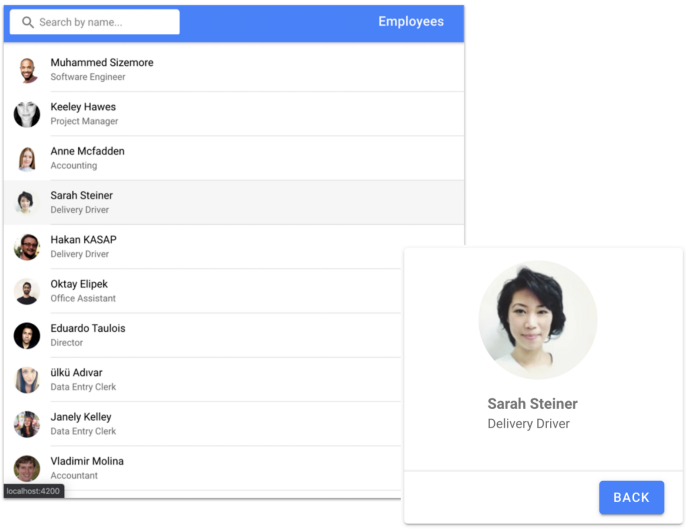
Let's convert our class-based components to functional components and specify type information to maximize productivity. With FC(s), we will also use React hooks to manage local state, centralize logic, and reduce the complexity of the view component code.

Finally we will also add keyboard support to auto-close the Detail view when the 'Escape' key is selected.

Scenario

Convert each of the following components to implementations as Functional Component; which also use React hooks.

- ContactsList: `libs/contacts/ui/src/lib/contacts-list.tsx`
- ContactDetail: `libs/contacts/ui/src/lib/contact-detail.tsx`



Tasks

1. Refactor ContactList to be a Functional Component
2. Refactor ContactDetail to be a Functional Component. Also add some code to listen for "Escape" keydowns and auto-navigate back to the list

> Notice how the components are much cleaner with the logic localized in the `render()` function?

> Be prepared to talk about your thoughts on FC(s) and hooks.

Code Snippets

`libs/contacts/ui/src/lib/contact-item.tsx`

```tsx
import React from 'react';
import { IonAvatar, IonItem, IonLabel } from '@ionic/react';
import { Contact } from '@workshop/shared/api';

interface ListItemProps {
  person: Contact;
  isButton?: boolean;
}

export const ContactListItem: React.FC<ListItemProps> = ({ person }) => {
  const url = `/contacts/${person.id}`;

  return (
    <IonItem routerLink={url}>
      <IonAvatar slot="start">
        <img src={person.photo} />
      </IonAvatar>
      <IonLabel>
        <h2>{person.name}</h2>
        <p>{person.position}</p>
      </IonLabel>
    </IonItem>
  );
};
```

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

```tsx
import { ContactsService } from '@workshop/contacts/data-access';
import { ContactListItem } from './contact-item';

export const ContactsList: React.FC = () => {
  const [service] = useState(() => new ContactsService());
  const [people, setPeople] = useState([]);
  const doSearch = (e: Event) => {
    const criteria = (e.target as HTMLIonInputElement).value;
    service.searchBy(criteria).then(setPeople);
  };

  useEffect(() => {
    service.getContacts().then(setPeople);
  }, [service, setPeople]);

  return (
    <IonPage>
      <IonHeader>
        <IonToolbar>
          <IonItem style={inlineItem}>
            <IonIcon icon={search}></IonIcon>
            <IonInput
              clearInput
              autofocus
              style={iconOnLeft}
              onIonChange={doSearch}
              placeholder="Search by name..."
            ></IonInput>
          </IonItem>
          <IonTitle style={stickyRight}> Employees </IonTitle>
        </IonToolbar>
      </IonHeader>
      <IonContent>
        <IonList>
          {people.map((person, idx) => {
            return <ContactListItem key={idx} person={person} />;
          })}
        </IonList>
      </IonContent>
    </IonPage>
  );
};
```

Lab (5): Create Custom Hooks

Our view components now use React hooks and manifest significantly easier code. Yet the views still have too much logic.
Let's refactor our code to use custom hooks that hide all state logic and update triggers.

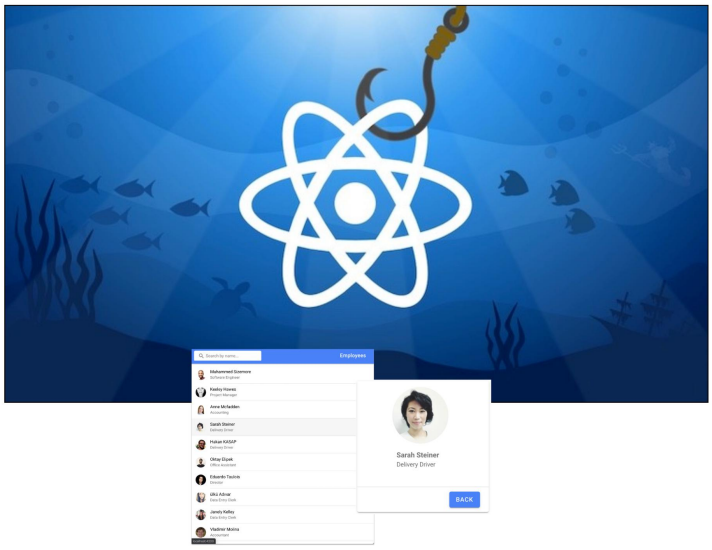
With these changes, the custom hooks will also give the 'latest' data to the view for rendering. And - in the case of the `ContactsList` - the actual search functionality is also hidden in the custom hook.

> Our views now become presentational components!

Scenario

To use Custom hooks, let's

- Create a `contacts.hooks.ts` module for our custom hooks
- Refactor the React Hooks from the View components to the `contacts.hooks.tsx`
- Define the tuples that the hooks return to the views
- Update the View components to use the new custom hooks



> Be prepared to talk about your thoughts regarding custom hooks.

Tasks

1. Create `libs/contacts/data-access/src/lib/contacts.hooks.ts`
   - Refactor the React Hooks code from **ContactsList**
   - Refactor the React Hooks code from **ContactDetails**
   - Define tuples that will be returned to the functional view components
2. Update the Data-Access library Public API to expose the custom hooks
3. Update **ContactDetails** to use the custom hook
4. Update **ContactList** to use the custom hook

> Notice how the the function components are now essentially presentational components.

> Be prepared to talk about your smart components vs presentational.

Code Snippets

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

```
import { useState, useEffect } from 'react';
import { useHistory } from 'react-router-dom';
import { useParams } from 'react-router';

import * as H from 'history';
import { Contact } from '@workshop/shared/api';
import { ContactsService } from './contacts.service';

/**
 * Define tuple types
 */
export type ContactHookResults = [Contact[], (criteria: string) => void];
export type ContactDetailsResult = [Contact, H.History<H.LocationState>];

/**
 * Custom React Hook useful to load Contact details
 */
export function useContactDetailHook(): ContactDetailsResult {
  const { id } = useParams();
  const history = useHistory();
  const [service] = useState(() => new ContactsService());
  const [contact, setContact] = useState<Contact>({} as Contact);

  useEffect(() => {
    service.getContactById(id).then(setContact);
  }, [id]);

  return [contact, history];
}

/**
 * Custom React Hook useful to search and load Contacts
 */
export function useContactsHook(): ContactHookResults {
  const [service] = useState(() => new ContactsService());
  const [criteria, setCriteria] = useState<string>('');
  const [people, setPeople] = useState<Contact[]>([]);

  useEffect(() => {
    service.searchBy(criteria).then(setPeople);
  }, [criteria, service, setPeople]);

  return [people, setCriteria];
}
```

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

```
export * from '@workshop/shared/api';
export * from './lib/contacts.service';
export * from './lib/contacts.hooks';
```

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

```
import { useContactDetailHook } from '@workshop/contacts/data-access';

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

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

  return contact ? (
    <IonPage style={gridItem} className="contactDetails">
      <IonCard>
        <IonCardContent>
          <IonAvatar slot="start">
            <img src={contact.photo} />
          </IonAvatar>
          <IonLabel>
            <h2>{contact.name}</h2>
            <p>{contact.position}</p>
          </IonLabel>
        </IonCardContent>
        <IonFooter>
          <IonButton routerLink="/contacts">Back</IonButton>
        </IonFooter>
      </IonCard>
    </IonPage>
  ) : (
    <Redirect to="/contacts" />
  );
};
```

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

```
import { useContactsHook } from '@workshop/contacts/data-access';
import { ContactListItem } from './contact-item';

export const ContactsList: React.FC = () => {
  const [people, doSearch] = useContactsHook();

  return (
    <IonPage>
      <IonHeader>
        <IonToolbar>
          <IonItem style={inlineItem}>
            <IonIcon icon={search}></IonIcon>
            <IonInput
              clearInput
              autofocus
              style={iconOnLeft}
              onIonChange={e => doSearch((e.target as HTMLIonInputElement).value)}
              placeholder="Search by name..."
            ></IonInput>
          </IonItem>
          <IonTitle style={stickyRight}> Employees </IonTitle>
        </IonToolbar>
      </IonHeader>
      <IonContent>
        <IonList>
          {people.map((person, idx) => {
            return <ContactListItem key={idx} person={person} />;
          })}
        </IonList>
      </IonContent>
    </IonPage>
  );
};
```

Lab (6a): Using Context

Both the `ContactsList` and the `ContactDetail` hooks create new instances of the `ContactsService`.

> The service instance is not shared!

We can share the instance using React Context(s). This approach allows use to provide a lookup mechanism in the view hierarchy... that is accessible from our custom hooks.

Scenario

To use the Context features and sharing the service instance:

- Define a ContactsContext object in `libs/contacts/data-access/src/lib/contacts.injector.tsx`.
  - Update the Data-Access API to expose the `ContactsContext`
- Let's refactor our routing to **ContactsDashboard** in `libs/contacts/ui/src/lib/contacts.dashboard.tsx`.
  - Modify the App to use only the **ContactsDashboard**
  - Update the UI library API to expose the **ContactsDashboard**
  - Use the `ContactsContext` wrapper with `value={service}` to provide the service instance.
- Refactor the React Hooks to employ `useContext()` to get access to the service instance.

> Be prepared to talk about your thoughts regarding Contexts. What are the downsides?

Tasks

Code Snippets

Lab (6b): Using Dependency Injection

Context provides a mechanism for sharing instances & data. It does not, however, help the developers partition service construction for use, nor caching, nor universal lookups.

> We need a Dependency Injection mechanism to solve these issues.

Leveraging DI give us more options... and allows us to even deprecate the use of Context (if appropriate).

Scenario

To use Dependendy Injection:

- Define a custom DependencyInjector
- Update the ContactsService to support DI
- Update the Contact Hooks to use the custom injector (instead of `useContext()`)
- Remove the ContactsContext

> How would you compare DI to Context? Be prepared to talk about your thoughts regarding DI.

Tasks

Code Snippets