Adopt Observable Data Services for Angular

Context and Problem Statement

Many of our components and services are tightly coupled towards the state of different domains. This results in tight coupling and difficulty in modifying the different areas. This has lead to `MessagingService` service being used to synchronize state updates by sending events. While event sourcing is a perfectly valid way of developing software, our current events are empty, which results in the components re-fetching their state manually.

Considered Options

- Observable/Reactive Data Services
- NGRX - Reactive State for Angular (Redux implementation)

Decision Outcome

Chosen option: **Observable data services**, because

- Allows us to quickly interate towards a more reactive data model.
  - Reactive data model lets us get rid of the event messages.
  - Components will always display the latest state.
- Does not require a significant upfront investment.
- The work towards a reactive data model will allow us to adopt patterns like NGRX in the future should it be needed.

Example

Organizations

The `OrganizationService` should take ownership of all Organization related data.

```
class OrganizationService {
  private _organizations: new BehaviorSubject<Organization[]>([]);
  organizations$: Observable<Organization[]> =
this._organizations$.asObservable();

  async save(organizations: { [adr: string]: OrganizationData }) {
    await this._organizations$.next(await this.decryptOrgs(this._activeAccount,
organizations));
  }
}

class Component implements OnDestroy {
  private destroy$: Subject<void> = new Subject<void>();

  ngInit() {
    this._organizationService.organizations$
```

```
      .pipe(takeUntil(this.destroy$))
      .subscribe((orgs) => {
        this.orgs = orgs;
      });
  }

  ngOnDestroy() {
    this.destroy$.next();
    this.destroy$.unsubscribe();
  }
}
```

In this example we use the `takeUntil` pattern which can be combined with an eslint rule to ensure each component clens up after themselves.

Pros and Cons of the Options

Observable Data Services

- Good: Lightweight
- Good: Can be refactored incrementally
- Good: Reactive, will notify on changes.
- Bad: No strict standard, more a set of guidelines.
- Bad: State is split into multiple tiny "stores"

NGRX

NGRX is the most popular Redux implementation for Angular. For more details, read about the motivation behind redux and view this diagram of NGRX architecture.

- Good: Most popular redux library for Angular
- Good: Decouples components
- Good: Single state which simplifies operations
- Bad: Requires a significant rewrite of the whole state layer
- Bad: Adds complexity and can make it difficult to understand the data flow.