Angular Now

- Angular's version history
- Couple of new features show-cased
- Guesses on the future
- @angular/cli status

Angular History

Angular 2 (September 2016)

Angular 2.1 (October 2016)

- Preloading of lazy loaded routes
- `:enter` & `:leave` animation aliases

Angular 2.2 (November 2016)

- AOT with `@angular/upgrade`

Angular 2.3 (December 2016)

- `@angular/language-service`
- Improved Zone.js error messages

Angular 2.4 (December 2016)

- RxJS 5.0.0 stable

Angular 3

- Skipped for consistency

Angular 4 (March 2017)

- Smaller and faster
- First deprecations
- Animations -> `@angular/animations`
- Improved `*ngIf` and `*ngFor`
- Angular Universal adopted by the core team
- TS 2.1 and 2.2 compatibility
- Packaging updates (FESM, ES2015 builds, Closure)

Angular 4.1 (April 2017)

- `strictNullChecks` TS compiler option
- TS 2.3 support

Angular 4.2 (June 2017)

- Huge improvements on animations
- New angular.io

Angular 4.3 (July 2017)

- `HttpClient`
- New router lifecycle hooks (`GuardsCheckStart`, `GuardsCheckEnd`, `ResolveStart`, `ResolveEnd`)
- Conditionally disable animations (`[@.disabled]`)

Angular 5 (November 2017)

- First removed APIs (e.g. `OpaqueToken`)
- Improved support for PWAs
- Build optimizer
- Deprecate `@angular/http` in favor of `HttpClient` in `@angular/common/http` introduced in 4.3
- RxJS 5.5 lettable operators
- Angular Universal State Transfer API and DOM Support
- Angular compiler is now a TypeScript transform -> incremental rebuilds dramatically faster (`ng serve --aot`)
- Preserve Whitespace option
- Update on blur/submit:

```
<input name="firstName" ngModel [ngModelOptions]="{updateOn: 'blur'}">
```

Angular 5.1 (December 2017)

- TS 2.5 compatibility
- Angular Material & CDK stable release

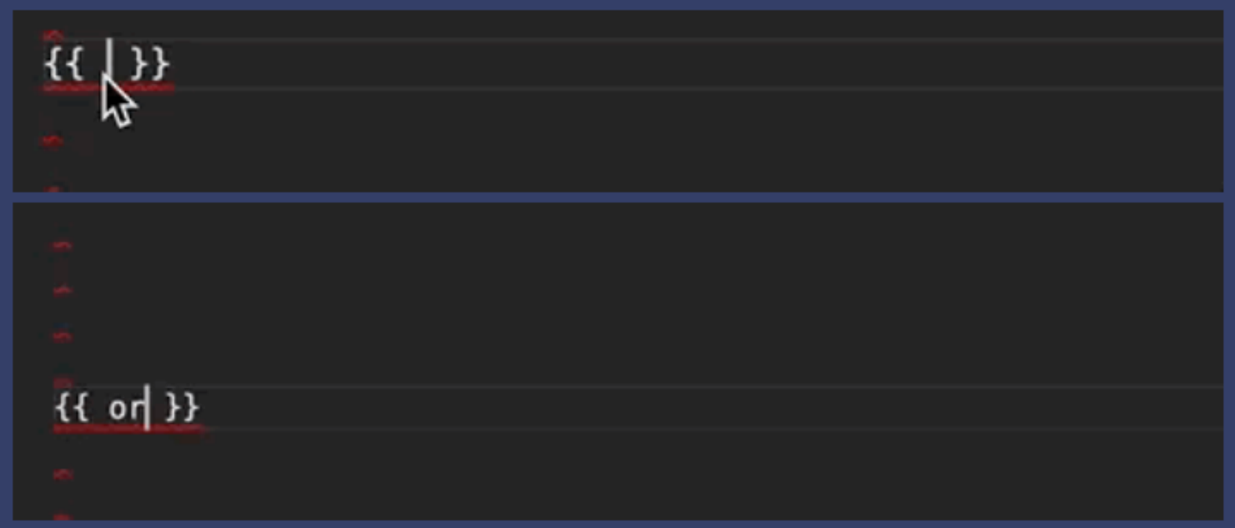
@angular/language-service

- Code completion, errors, hints etc. for templates
- Works at least in VS Code & JetBrains products (IntelliJ IDEA & WebStorm)
- Installable as simply as

```
npm install --save-dev @angular/language-service
```

or

```
devDependencies {
"@angular/language-service": "^5.0.0"
}
```



Build Optimizer

- Produces smaller bundles by for example skipping *vendor.bundle.js* generation and removing decorators
- On by default for production builds with CLI if using Angular 5
- Usage otherwise: `ng build --build-optimizer`
- Details

Guesses On the Future

Main concentration in near future (Angular 5.x):

- `@angular/service-worker` and other PWA updates
- CLI fixes
- Further watch mode speed improvements and fixes
- Documentation improvements
- TS 2.6 and stricter defaults in CLI

@angular/cli Status

- 1.6 just released with support for the latest Angular (5.1)
- Still huge number of problems but getting more stable
- Now supports PWAs better, yet not perfectly

Stuff Considered for @angular/cli V2

- AOT on everywhere by default (now only with `--prod`)
- NativeScript support
- Usability upgrades based on actual user feedback
- Configuration in JS/TS instead of JSON
- Webpack 4 -> further speed improvements

Advanced Observables

- Building an observable yourself
- Lettable operators
- Creating a custom operator
- Subjects
- Hot vs. cold observables
- Testing observables

Producer

- Every observable has some data source. In web these include:
  - DOM events (clicks, key presses, etc.): 0-N values
  - WebSockets: 0-N values
  - Intervals: 0-N values
  - AJAX: 1 value
  - Timeouts: 1 value
- This data source is called the producer as it produces the data. Data might be:
  - Coordinates of a click on screen
  - Pressed key
  - Number of times interval has triggered so far
  - HTTP response

RxJS Observable Creators

RxJS provides observable creators for most of the producers:

- `fromEvent` for DOM events
- `webSocket` for Web Sockets
- `of` and `from` for static values
- `range` for ranges
- `fromPromise` for current promise

Observable Autopsy

Observables are just functions binding the observer to the producer's events and provide the destructuring logic with uniform API:

```
function myObservable(observer) {
    const datasource = new DataSource();
    datasource.ondata = (e) => observer.next(e);
    datasource.onerror = (err) => observer.error(err);
    datasource.oncomplete = () => observer.complete();
    return () => {
        datasource.destroy();
    };
}
```

```
myObservable({
    next: console.log,
    error: console.error,
    complete: () => console.log('Completed'),
})
```

Exercise

1. Implement a function `IntervalObservable` which creates an observable that when subscribed emits next value (0, 1, 2 and so on) every N (argument) milliseconds. You can skip error or complete events. The following should work:

```
const subscription = intervalObservable(500).subscribe({ next: console.log });
setTimeout(subscription, 3000);
// Should log 0, 1, 2, 3, 4, 5
```

2. Make your observable more RxJS-ish by altering it to accept the `next` as first argument and returning an object with `unsubscribe` method:

```
const subscription = intervalObservable(500).subscribe(console.log);
setTimeout(subscription.unsubscribe, 3000);
// Should log 0, 1, 2, 3, 4, 5
```

3. Add second parameter for `intervalObservable` called `duration` which is used to determine when the observable should complete and then add second parameter for the complete function to be provided so that this works:

```
intervalObservable(500, 3000).subscribe(console.log, () => console.log('Comple
// Should log 0, 1, 2, 3, 4, 5, Completed!
```

RxJS Building Blocks

- Observables
- Operators
- Subscriptions/subscribers
- Subjects
- Schedulers

Lettable Operators

- New way to apply operators released in RxJS 5.5

```
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';
Observable.of(1, 2, 3).map(x => x * 2).subscribe(console.log);
```

becomes

```
import { of } from 'rxjs/observable/of';
import { map } from 'rxjs/operators/map';
of(1, 2, 3).pipe(map(x => x * 2)).subscribe(console.log);
```

- No more `Observable.prototype` patching
- 4 operators renamed:
  - `do` -> `tap`
  - `catch` -> `catchError`
  - `switch` -> `switchAll`
  - `finally` -> `finalize`

Lettable Operators

- Pros:
  - Tree-shaking possible
  - Static analysis possible
  - Functional composition -> custom operators easier to implement
- Works best with TS >2.4
- Import from `rxjs/operators/<operator name>` instead of `rxjs/operators` to minimize bundles

Exercise

Convert the following to lettable operators syntax

```
Observable.of(1, 2, 3).map(x => x * 2).subscribe(console.log);
Observable.fromEvent(document, 'click')
    .filter(event => event.clientX < 100)
    .bufferTime(1000)
    .subscribe(console.log);
```

Custom Operators

With RxJS 5.5. lettable operators as simple as implementing function that returns a function with signature:

```
(source: Observable<T>): Observable<R>
```

```
export const toPower =
  (n: number) =>
    (source: Observable<number>): Observable<number> =>
      source.pipe(map((x: number) => Math.pow(x, n)));
```

which could then be composed like

```
import { range } from 'rxjs/observable/range';
import { toPower } from './to-power.operator';

range(0, 10).pipe(toPower(3)).subscribe(console.log)
```

Exercise

Implement custom operator called `flatten` that operates on an array of arrays (like `[[1, 2, 3], [4, 5, 6]]`) and flattens them to single-dimension array ([1, 2, 3, 4, 5, 6])

Your operator should log `[1, 2, 3, 4, 5, 6]` when used like this:

```
import { of } from 'rxjs/observable/of';
of([[1, 2, 3], [4, 5, 6]]).pipe(flatten()).subscribe(console.log)
```

Tip: Your operator function accepts no parameters at all

Bonus: Make types sound so that the array that is returned has to contain items of same type than the original (generic typing)

Hot vs. Cold Observables

Ben Lesh (RxJS 5 author):

- "An observable is "cold" if its underlying producer is created and activated during subscription."
- "An observable is "hot" if its underlying producer is either created or activated outside of subscription."
- "Warm": Subscriptions shared but producer is only initiated once there is a single subscription

In practice:

- RxJS Observables are cold
- Promises are always hot

Cold Observables

```
const observable = this.httpClient.get('example.com/foo.json'); // 1
observable.subscribe(console.log); // 2
```

On line 1 the producer (HTTP request) isn't created/activated yet On line 2 the subscription causes producer to be activated

```
observable.filter(x => x % 2 === 0)
  .subscribe(x => console.log('even', x));
observable.filter(x => x % 2 === 1)
  .subscribe(x => console.log('odd', x));
```

Subject

Subject is a combination of observable and observer

- Has the `Observable`'s methods like `subscribe` and `pipe`
- Has the `next` method to publish new values

```
const subject = new Subject<number>();
subject.subscribe(console.log);
subject.next(100); // 100 is logged to console
```

Subscription is shared under the hood -> can have multiple subscribers (unlike observable):

```
const subject = new Subject();
const obs = interval(1000);
obs.subscribe(subject);
subject.subscribe(console.log);
setTimeout(() => {
  subject.subscribe(console.log);
}, 5000);
```

Prints 0, 1, 2, 3, 4, 4, 5, 5, 6, 6

Making Observables Hot

```
function makeHot(cold) {
  const subject = new Subject();
  cold.subscribe(subject);
  return new Observable((observer) => subject.subscribe(observer));
}
```

```
const httpRequest = this.httpClient.get('example.com/foo.json');
const hotObservable = makeHot(httpRequest); // Request is sent
hotObservable.subscribe(console.log); // Logs the response once available
hotObservable.subscribe(console.log); // Second subscribe does not generate second
```

Preloading Data

```
@Injectable()
export class PreloadedNewsService {
    url = 'https://hacker-news.firebaseio.com/v0/topstories.json';
    request: Observable<any>;

    constructor(private httpClient: HttpClient) {
        this.request = makeHot(this.httpClient.get(this.url));
    }

    getNews() {
        return this.request;
    }
}
```

```
this.preloadedNewsService.getNews().subscribe();
```

.publish()

- Returns a `ConnectableObservable` that shares the subscription to the original observable but only subscribes once `.connect()` is called

```
const obs = this.httpClient.get(this.url).publish(); // Returns ConnectableObs
obs.subscribe(console.log);
obs.subscribe(console.log);
obs.connect();
```

- Calling `.connect()` multiple times subscribes to the source multiple times
- Subscriptions are made for the original observable, so they won't work:

```
const obs = this.httpClient.get(this.url).publish(); // Returns ConnectableObs
obs.subscribe(console.log); // This is logged normally
setTimeout(() => {
obs.connect();
obs.subscribe(console.log); // Does nothing as the original observable (HTTP r
}, 5000);
obs.connect();
```

.share()

- Returns a new observable that always shares a single subscription to the original observable
- Same as `.publish().refCount()`
- `.refCount` is only available on `ConnectableObservable`. It returns a new observable that connects and stays connected to the source as long as there is at least one subscription to itself

```
const obs = httpClient.get(this.url).share();
obs.subscribe(console.log);
obs.subscribe(console.log); // Only one request made since HTTP call takes lor
```

- Assuming the HTTP response is retrieved in 10 seconds, the following will be make the request twice

```
const obs = httpClient.get(this.url).share();
obs.subscribe(console.log);
setTimeout(() => obs.subscribe(console.log), 10000);
```

Too Hot Observable

The following won't log anything assuming HTTP response arrives in 5 seconds

```
const obs = this.httpClient.get(this.url).publish(); // Returns ConnectableObserva
obs.connect();
setTimeout(() => {
  obs.subscribe(console.log);
}, 5000);
```

Subject does not store values

Subject Variations

- `BehaviorSubject`: Gives the last (or initial) value instantly when subscribing

```
const subject = new BehaviorSubject<number>(10);
subject.subscribe(value => console.log('A: ' + value));
subject.next(20);
subject.subscribe(value => console.log('B: ' + value));
// A: 10
// A: 20
// B: 20
```

- `ReplaySubject`: Like `BehaviorSubject` but no initial value and can record multiple previous values
- `AsyncSubject`: Gives the last value emitted for subscribers but only when it completes:

```
const subject = new AsyncSubject<number>();
subject.subscribe(value => console.log('A: ' + value));
subject.next(10);
subject.subscribe(value => console.log('B: ' + value));
subject.next(20);
subject.complete();
// A: 20
// B: 20
```

Exercise

Implement function `instantCache` that starts the interval instantly when created (is hot) and provides the same value for each subscriber

```
const observable = instantCache(interval(1000).pipe(take(4)));
setTimeout(() => observable.subscribe(console.log), 5000);
// Should print "3" in 5 seconds
```

Tip: take completes after emitting 4 values Tip: there are many subjects capable for this

Testing Observables

- Subscribe manually and then check that the values are correct
- Mocking problematic if no dependency injection is used (thank god Angular)
- Example:

```
it('should make a call to Hacker News', () => {
observable.take(1).subscribe(result => {
  expect(result).toEqual([1, 2, 3]);
});
});
```

Angular 4's async as Syntax

Angular 4 introduced two updates to templates

```
<div *ngIf="userList | async as users; else loading">
  <div *ngFor="let user of users; count as count; index as i">
User {{i}} of {{count}}
  </div>
</div>
<ng-template #loading>Loading...</ng-template>
```

Decision Tree at reactivex.io/rxjs/ - Demo

Decision tree

State Management & Unidirectional Data Flow

- What is state and its management?
- Service-based state management and its problems
- Unidirectional data flow
- Redux
- @ngrx for state management

What Is State?

- Collection of application's data
- Snapshot of everything stored within the app at any given time

Different Kinds of State

- Client vs server state (current form filled by user vs. database-backed)
- UI vs application state (is button enabled vs. list of clients)
- Temporary vs permanent state (is collapsible section open or not vs. what is in database)

Where Is State Stored?

- Database in server
- Database in client
- Functionality-scoped service (`CustomersService`)
- Functionality's component (`CustomerListComponent`)
- In URL (`my-app.com/customers/10`)

State Synchronizing

- State is rarely in single location only
- Often state is moved from one place to another = state synchronization
- E.g.:
  - From server to client
  - From Angular service to component and vice versa
  - Via URL parameters to new service

State Management

- Managing all of the mentioned gets cumbersome in large applications
- Purpose of state management provide consistent methodology for managing state that is:
  - Testable
  - Debuggable
  - Easy to reason about
  - Manageable
  - Scalable
  - Visualizable

Using Services

- Service by backend resource (`CustomerService`, `ReportService`)
- Service has method for loading, saving, updating etc. some resource

```
@Injectable()
export class HackerNewsService {

  constructor(private httpClient: HttpClient) {
  }

  findAll() {
    return this.httpClient.get('https://hacker-news.firebaseio.com/v0/topstories.j
  }

  create(data: any) {
    // Simulate creating a new news
    const id = Math.floor(Math.random() * 100000);
    return Observable.of(id).pipe(delay(1000));
  }
}
```

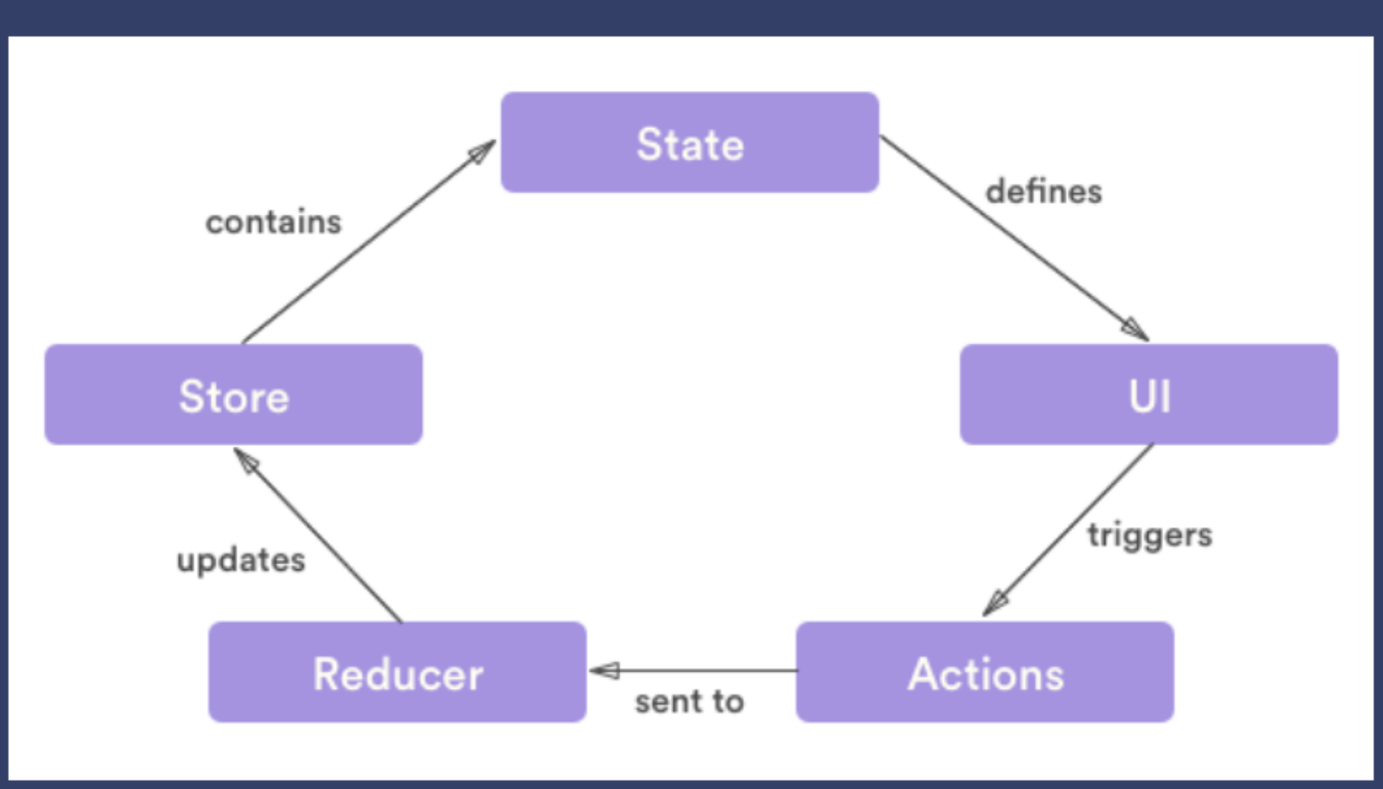
Problem with Services

- Reinventing the wheel every time with custom logic for asynchronous
- Unnecessary API calls easily done
- State management & synchronization split over all of the services

Unidirectional Data Flow

Basic concepts:

- There is only a single application-wide state called *store*
- *Store* (and thus state) can only be changed by *a reducer*
- *Reducer* is triggered by *an action*
- *Action* is a pair of name and payload such as `('ADD', 10)` triggered by UI



Source: Stack Overflow

Example (Empty) Store

- Store is usually a object:

```
const store = {
    users: [],
    reports: [],
    currentUser: {}
};
```

- Though it can as well be anything like `number`:

```
const store = 0;
```

Actions

Composes of

- *Type*
  - Action's name as string
  - Should be unique within the app
  - Often consists of target reducer with the modification like `'[Counter] Add'`
- *Payload*
  - Any kind of data that is passed along for reducers
  - For example the increment size or newly created object

Reducers

- Pure & synchronous function:
  - Result (output) depends only on arguments (inputs) passed
  - Inputs aren't modified
  - Does not cause side effects such as HTTP calls (revisited later)
  - Returns instantly instead of waiting for something (HTTP request, timeout) to be done
- Simplest possible reducer returns the very same state:

```
function reducer(currentState, action) {
    return currentState;
}
```

- The result of reducer will always be the new state of application
- Type of an action might look like:

```
interface Action {
    name: string;
    payload: any;
}
```

More advanced reducer

Assuming the application's state is number, reducer could look like:

```
function reducer(currentState: number, action: Action) {
    switch(action.name) {
        case '[Counter] Add': {
            return currentState + action.payload;
        }
        case '[Counter] Decrease': {
            return currentState - action.payload;
        }
        default: {
            return currentState;
        }
    }
}
```

So the reducer would work like

```
const initialState = 100;
const newState = reducer(initialState, { name: '[Counter] Add', payload: 10}); //
```

Redux

- Library implementing unidirectional data flow kind of state management
- By Facebook
- Based on the Flux
- De facto state management library when for React
- Three principles:
  - Single source of truth (store)
  - State is read-only (actions)
  - Changes are made with pure functions (reducers)

Redux - Demo

Immutability in JS

- In JavaScript nothing is actually immutable
- `const` is only constant reference to object:

```
const obj = { foo: 'bar' };
obj.foo = 'asd'; // OK
obj = {}; // TypeError: Assignment to constant variable.
```

- Libraries available such as Immutable.js (by Facebook)
- Favor spreads that produce new object:

```
const state = [1, 2, 3];
const payload = 4;
const newState = [...state, payload];
```

Side Effects

- Everything has been synchronous so far - what about asynchronous actions?
- Wikipedia: "a function is said to have a side effect if it modifies some state outside its scope"
- Problem: Programs behavior may be affected by order of evaluation
- Numerous solutions in Redux world, e.g. redux-thunk, redux-saga and redux-observable

@ngrx

- Reactive Extensions for Angular (RxJS = Reactive Extensions for JS)
- Full state management solution for Angular applications
- Basically Redux + RxJS
- Set of npm modules:
  - @ngrx/store - Redux clone
  - @ngrx/effects - redux-observable like side effect management
  - @ngrx/router-store - Connects Angular router to store
  - @ngrx/devtools - Tooling for
  - @ngrx/entity - Entity State adapter for managing record collections
- Angular's current router is based on @ngrx router
- Victor Savkin and Rob Wormald are part of the authors

@ngrx/store

- Redux clone
- Observables (RxJS) used for store accessing
- Connected with Angular's dependency injection

@ngrx/store - Setup

```
npm install @ngrx/store
```

```
import { NgModule } from '@angular/core'
import { StoreModule } from '@ngrx/store';
import { counterReducer } from './counter';

export interface AppState {
    counter: number;
}

@NgModule({
  imports: [
    BrowserModule,
    `StoreModule.forRoot({ counter: counterReducer })`
  ]
})
export class AppModule {}
```

@ngrx/store - Reducer

```
// counter.ts
import { Action } from '@ngrx/store';

export const INCREMENT = '[Counter] Increment';
export const DECREMENT = '[Counter] Decrement';
export const RESET = '[Counter] Reset';

export function counterReducer(state: number = 0, action: Action) {
    switch (action.type) {
        case INCREMENT:
            return state + 1;

        case DECREMENT:
            return state - 1;

        case RESET:
            return 0;

        default:
            return state;
    }
}
```

@ngrx/store - Accessing State

```
export class MyAppComponent {
    counter: Observable<number>;

    constructor(private store: Store<AppState>) {
        this.counter = store.select('counter');
    }
}
```

```
<div>Current Count: {{ counter | async }}</div>
```

@ngrx/store - Dispatching Actions

```
export class MyAppComponent {
    constructor(private store: Store<AppState>) {
    }

    increment(){
        this.store.dispatch({ type: INCREMENT });
    }

    decrement(){
        this.store.dispatch({ type: DECREMENT });
    }

    reset(){
        this.store.dispatch({ type: RESET });
    }
}
```

Typed Actions

Typed actions make it easy to wrap the action type and payload for dispatching

```
export const INCREMENT: '[Counter] Increment';
export const RESET: '[Counter] Reset';

export class IncrementAction implements Action {
  readonly type = INCREMENT;

  constructor(public value: number) {
  }
}

export class ResetAction implements Action {
  readonly type = RESET;
}


export type CounterAction = IncrementAction | ResetAction;
```

Typed Action Reducer

```
import { CounterAction, INCREMENT } from '../actions/counter.actions';

export function counterReducer(state = 10, action: CounterAction) {
  switch (action.type) {
    case INCREMENT:
      return state + action.value;

    default:
      return state;
  }
}
```

Typed Action Dispatching

```
this.store.dispatch(new IncrementAction(value));
this.store.dispatch(new ResetAction());
```

Exercise

Install @ngrx/store to the project and implement the store with single reducer called `HackerNewsReducer` which reacts to action called `SET` that sets the array of ids. Dispatch this kind of action when application is opened with some dummy ids. The payload should be typed as `number[]`.

1. Install @ngrx/store with npm
2. Instantiate the `StoreModule` and set up `AppState` in `app/app.module.ts`
3. Declare the typed actions in `app/actions/hacker-news.actions.ts`
4. Implement the reducer in `app/reducers/hacker-news.reducer.ts`
5. Create a component `HackerNewsComponent` that reads from the store and dispatches the initial action to set some data
6. Use the created component in `AppComponent`

Testing Reducers

- Reducers are pure and synchronous
- Testability is one of the main benefits of reducers
- Example:

```
import { IncrementAction } from '../actions/counter.actions';
import { counterReducer } from './counter.reducer';

describe('counter reducer', () => {
  describe('increment', () => {
    it('should increase the counter with value', () => {
      const state = counterReducer(
        10,
        new IncrementAction(50)
      );
      expect(state).toEqual(60);
    });
  });
});
```

Exercise

Test your reducer for setting the array.

@ngrx/store-devtools

- See all actions happened ever
- Import/export state
- Cancel actions
- Reorder actions
- Time travelling
- Generate tests

@ngrx/store-devtools - Setup

```
npm install @ngrx/store-devtools
```

```
@NgModule({
  declarations: [
    AppComponent,
  ],
  imports: [
    BrowserModule,
    StoreModule.forRoot({ counter: counterReducer }),
    `StoreDevtoolsModule.instrument()`
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

Install Chrome extension

@ngrx/store-devtools - Demo

Exercise

Install the Chrome extension and instrument the dev tools and verify you see them in the developer tools of Chrome.

@ngrx/effects

- Side effect model for @ngrx
- Models side effects as Observables
- Each `effect` is a property with annotation `@Effect()` of the Effects class
- Each `effect` checks with `ofType` whether an action is for it and if so, returns a new action at some point
- Simplest effect that just changes the action type

```
export class CounterEffects {
  constructor(private actions$: Actions) {
  }

  @Effect() loadCounterIncrement$: Observable<Action> = this.actions$
    .ofType('[Counter] Fetch increment')
    .pipe(
      map(() => new IncrementAction(10))
    );
}
```

@ngrx/effects - Setup

```
npm install @ngrx/effects
```

```
import { EffectsModule } from '@ngrx/effects';
import { CounterEffects } from './effects/counter-news';

@NgModule({
  imports: [
    `EffectsModule.forRoot([CounterEffects])`
  ]
})
export class AppModule {}
```

@ngrx/effects - Effect

```
import { LOAD_COUNTER_INCREMENT, LoadFailed, Set } from '../actions/counter.action

@Injectable()
export class CounterEffects {
  constructor(private httpClient: HttpClient,
              private actions$: Actions) {
  }

  @Effect() loadCounterIncrement$: Observable<Action> = this.actions$
    .ofType(LOAD_COUNTER_INCREMENT)
    .pipe(
      mergeMap(
        () => this.httpClient
          .get('counter.example.com/increment')
          .pipe(
            // If successful, dispatch set action with result list
            map((data: any[]) => new IncrementAction(data)),
            // If request fails, dispatch failed action
            catchError(() => of(new LoadFailed()))
          )
      )
    );
}
```

Exercise

Implement news fetching from URL `https://hacker-news.firebaseio.com/v0/topstories.json` as an effect and populate the current news state with result. Skip the error handling until next exercise.

Exercise

Implement error/success messages component that

- Is rendered on top of the screen if there are messages
- Is controlled by its own reducer with its own actions and the reducer is tested
- Is shown when loading of top stories succeeds and/or fails

Tip: Start by thinking what should be handled by reducers and what by effects?

Non-dispatching Effects

- Event does not need to dispatch a new action in the end
- If so, pass `{ dispatch: false }` as a argument for the `@Effect()`
- Example:

```
@Effect({ dispatch: false }) logActions$ = this.actions$
    .pipe(tap(action => console.log(action)));
```

Testing Effects

- Easiest with Angular TestBed to utilize dependency injection
- `providerMockActions` will deliver a new Observable to subscribe to for each test

```
describe('counter effects', () => {
  let effects: CounterEffects;
  let actions: Subject<any>;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [
        HttpClientTestingModule,
      ],
      providers: [
        CounterEffects,
        provideMockActions(() => actions),
      ],
    });

    effects = TestBed.get(CounterEffects);
    httpMock = TestBed.get(HttpTestingController);
  });
```

Testing Effects

```
it('should make the HTTP request for counter increment', () => {
    actions = new ReplaySubject(1);
    actions.next(new LoadCounterIncrement());

    effects.loadCounterIncrement$.subscribe(result => {
      expect(result).toEqual(new IncrementAction(10));
    });

    const request = httpMock.expectOne('counter.example.com/increment');
    request.flush(10);
    httpMock.verify();
});
```

Exercise

Implement the testing for your Hacker news effects.

@ngrx/router-store

- Connects Angular router to @ngrx/store
- `ROUTER_NAVIGATION` action dispatched on navigation before any guards or resolvers run
- Reducer throwing an error for `ROUTER_NAVIGATION` cancels navigation
- `ROUTER_CANCEL` (guard prevented navigation) and `ROUTER_ERROR` (navigation error)

@ngrx/router-store - Setup

```
npm install @ngrx/router-store
```

```
import { StoreRouterConnectingModule, routerReducer } from '@ngrx/router-store';
import { App } from './app.component';

@NgModule({
  imports: [
    BrowserModule,
    StoreModule.forRoot({ router: routerReducer }),
    RouterModule.forRoot([
      // routes
    ]),
    StoreRouterConnectingModule
  ],
  bootstrap: [App]
})
export class AppModule { }
```

Exercise

1. Add routing to your application by adding

```
RouterModule.forRoot([
  { path: '', component: HackerNewsComponent }
]),
```

to the `imports` list in *app.module.ts* and replacing *app.component.html* content with `<router-outler></router-outlet>`.

1. Add reducer to prevent accessing single page in the system and instead navigate to home page and generate an error message.

Feature modules & Lazy Loading

- Both `@ngrx/store` and `@ngrx/effects` support feature modules
- Also lazy loading is supported

@ngrx/entity

- Latest addition to @ngrx
- API to manipulate and query entity collection
- Not covered in more detail

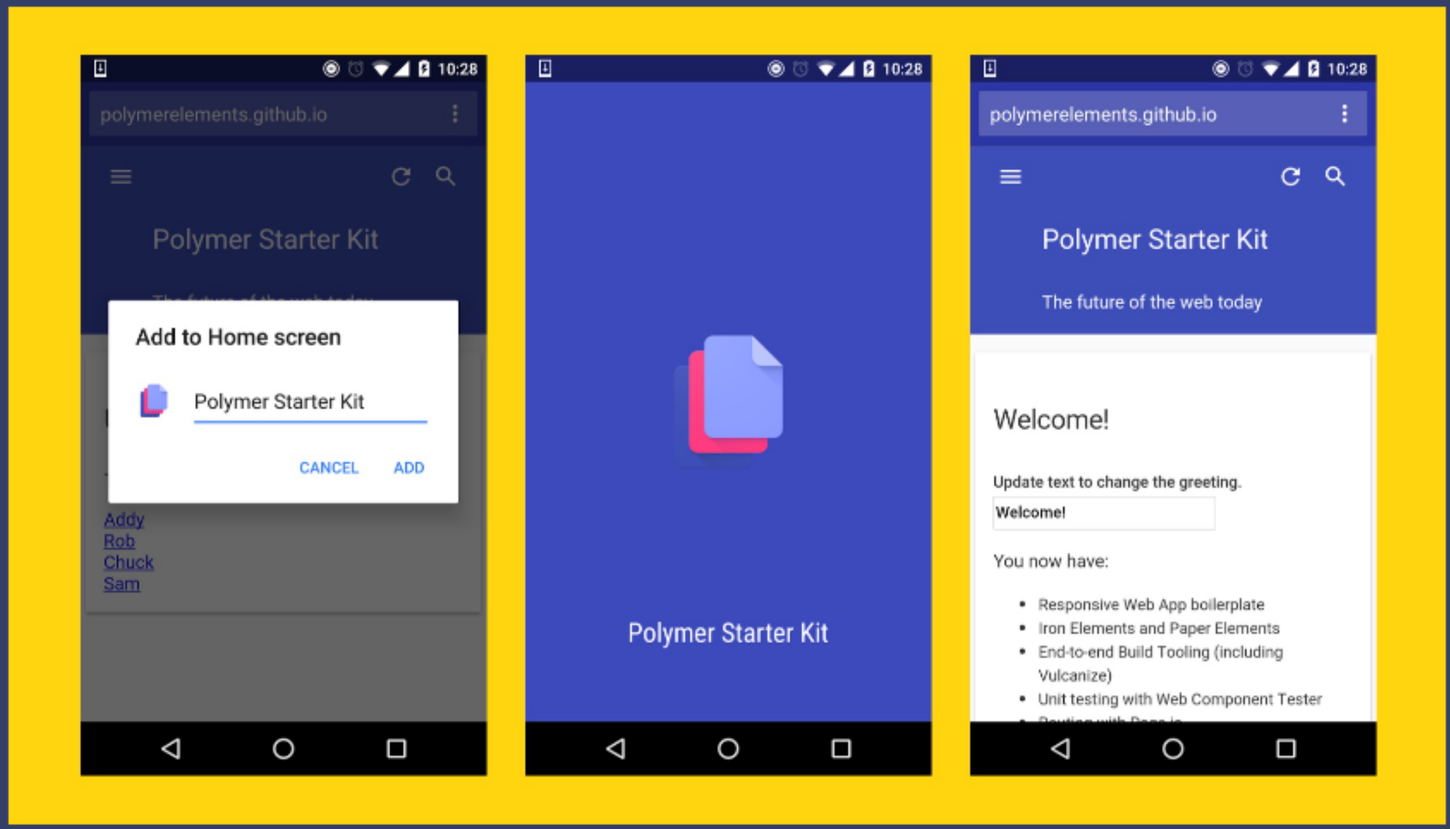
Progressive Web Applications (PWA)

- Definition
- Characteristics
- Technologies one by one
- Platform support
- Lighthouse

Definition

"Progressive Web Apps (PWAs) use modern web capabilities to deliver fast, engaging, and reliable mobile web experiences that are great for users and businesses."

PWA


10:28
polymerelements.github.io
Polymer Starter Kit
Add to Home screen
Polymer Starter Kit
CANCEL
ADD
Addy
Rob
Chuck
Sam
10:28
Polymer Starter Kit
10:28
polymerelements.github.io
Polymer Starter Kit
The future of the web today
Welcome!
Update text to change the greeting.
Welcome!
You now have:
Responsive Web App boilerplate
Iron Elements and Paper Elements
End-to-end Build Tooling (including Vulcanize)
Unit testing with Web Component Tester

Characteristics

- *Progressive* - Work for every user, regardless of browser choice because they're built with progressive enhancement as a core tenet.
- *Responsive* - Fit any form factor: desktop, mobile, tablet, or forms yet to emerge.
- *Connectivity* independent - Service workers allow work offline, or on low quality networks.
- *App-like* - Feel like an app to the user with app-style interactions and navigation.
- *Fresh* - Always up-to-date thanks to the service worker update process.
- *Safe* - Served via HTTPS to prevent snooping and ensure content hasn't been tampered with.
- *Discoverable* - Are identifiable as "applications" thanks to W3C manifests and service worker registration scope allowing search engines to find them.
- *Re-engageable* - Make re-engagement easy through features like push notifications.
- *Installable* - Allow users to "keep" apps they find most useful on their home screen without the hassle of an app store.
- *Linkable* - Easily shared via a URL and do not require complex installation.

Who Uses PWAs?

- Twitter Lite
- Ali Express
- Trivago
- Washington Post
- Forbes
- Flipboard
- Telegram

Technologies

- Service Worker
- Web App Manifest
- Server-side rendering
- Local databases

Universal Rendering

- JS app that can be rendered either in browser or in the server
- Once the app (JS) is loaded in the browser the HTML representation is replaced with the JS one
- Supported by *@angular/cli@1.6.0*
- Server-side rendering (SSR) often used term
  - Takes application and route as input and produces HTML representation
- CLI Tutorial
- Benefits:
  - Fast first load
  - Caching
  - Search-engine optimization

Exercise

Build Universal app (source):

1. Run:

```
npm install --save @angular/platform-server @nguniversal/module-map-ngfactory-
ng g universal universal
```

2. Copy this to be the `server.ts` in your project's root
3. Copy this to be the `webpack.server.config.js` in your project's root
4. Add these to the `scripts` section of `package.json`:

```
"ssr": "npm run build:ssr && npm run serve:ssr",
"build:ssr": "npm run build:client-and-server-bundles && npm run webpack:serve
"serve:ssr": "node dist/server.js",
"build:client-and-server-bundles": "ng build --prod && ng build --prod --app 1
"webpack:server": "webpack --config webpack.server.config.js --progress --colo
```

Continue to next slide ->

89 / 125

Exercise

1. Alter `.angular-cli.json` to contain

```
"outDir": "dist/server/",
```

for server and

```
"outDir": "dist/browser/",
```

for browser
2. Run

```
npm run ssr
```

and go to `localhost:4000` to see what initial page load's HTML now looks

State Retrieval

- Problem: Both, browser and backend, ask for the same data

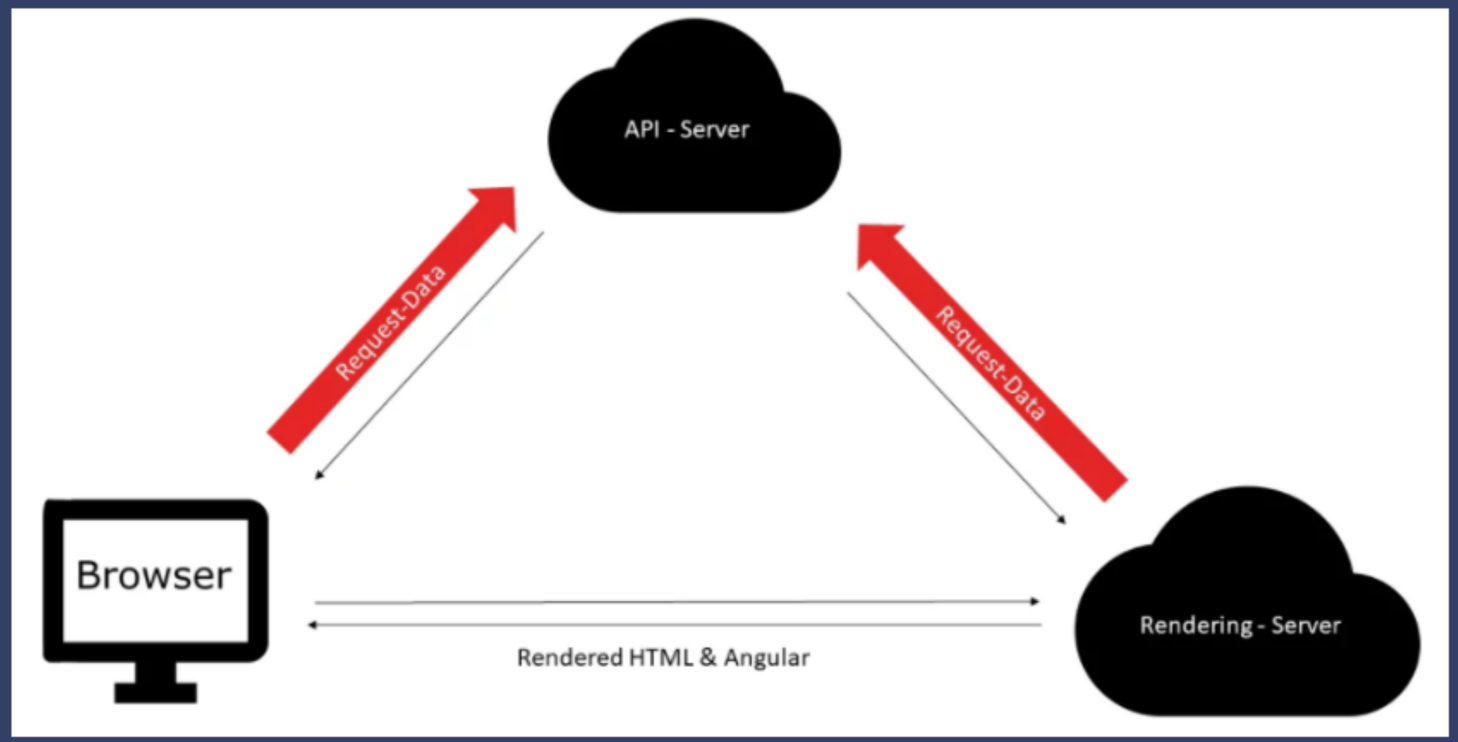


Image source: Malcoded

Solutions

- Solution 1: Data retrieved on both platforms
- Solution 2: Include state with JS

Data retrieved on both platforms

- E.g. Browser makes HTTP call to backend while backend fetches data from database
- Works only if backend has direct access to the data source

State Transfer

- Browser receives the data from the backend while bootstrapped
- Angular Transfer State API

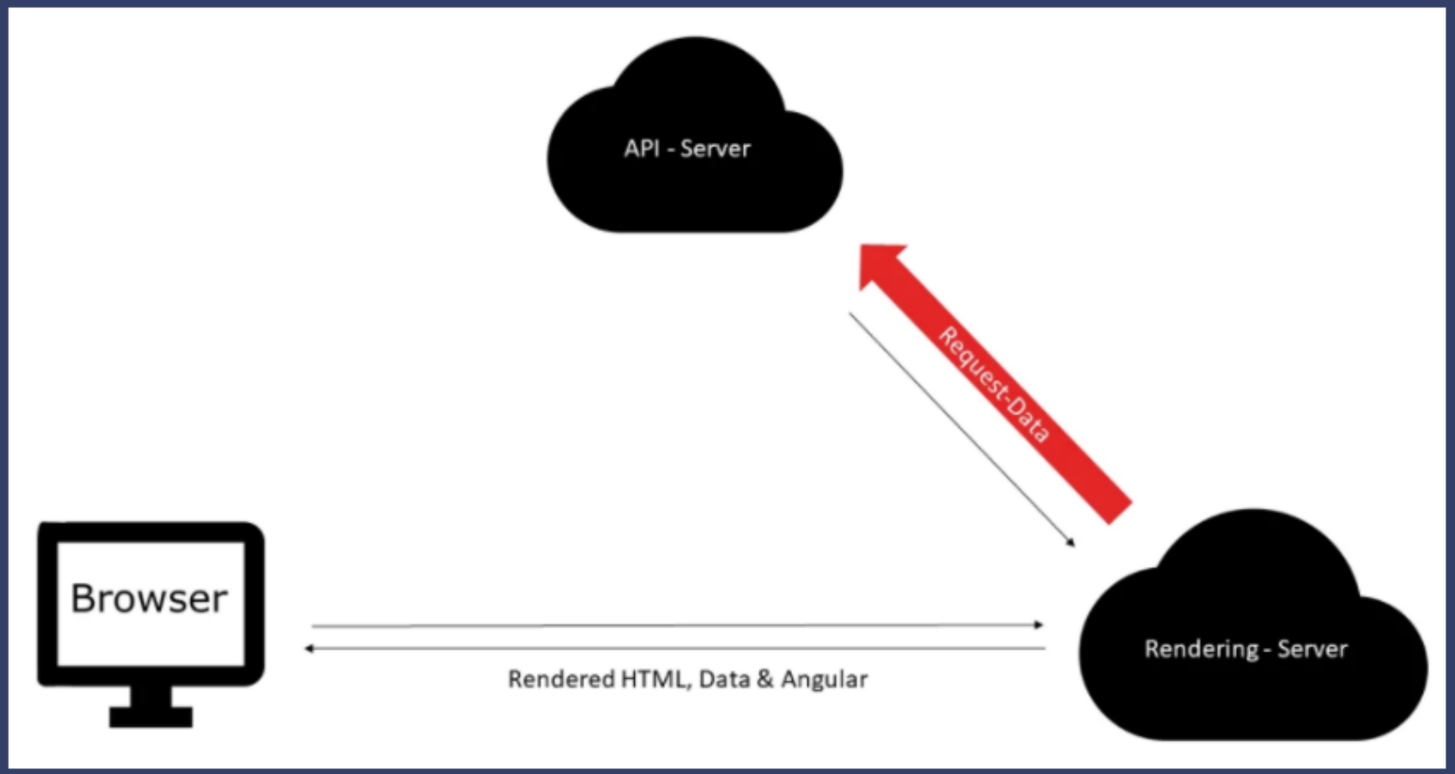


Image source: Malcoded

Angular State Transfer API

- Composed of a module for each platform:
  - `BrowserTransferStateModule` found from `@angular/platform-browser`
  - `ServerTransferStateModule` found from `@angular/platform-server`
- Provides easy interface to move data from backend to the browser when control is changed

Service Workers

- "A service worker is an event-driven worker registered against an origin and a path."
- Separate thread -> Has no DOM access!
- Only available over HTTPS
- AppCache replacement

Service Worker Features

- Access to Background sync & Push APIs
- Network request interception
- Access to file system
- Communication with main thread

Exercise

Enable offline usage via service workers

1. Run

```
npm install @angular/service-worker
npm install -g http-server
ng set apps.0.serviceWorker=true
```

2. Add to `imports` in `app.module.ts`:

```
import { ServiceWorkerModule } from '@angular/service-worker';
import { environment } from '../environments/environment';
imports: [
...
ServiceWorkerModule.register('/ngsw-worker.js', {enabled: environment.producti
]
```

3. Copy this as `src/ngsw-config.json`
4. Run

```
ng build --prod
cd dist/browser/
http-server -p 8080
```

Web App Manifest

- JSON file that contains metadata about the application
- Makes the app installable to home screen
- Usually named `manifest.webmanifest`
- Contains:
  - General info such as name, description etc.
  - Look'n'feel (home screen icons, color theme, screen size and orientation)
  - Related applications

Exercise

Add Web App Manifest:

1. Save this as `src/manifest.webmanifest`
2. Add "manifest.webmanifest" to the both `assets` arrays in `.angular-cli.json`
3. Add

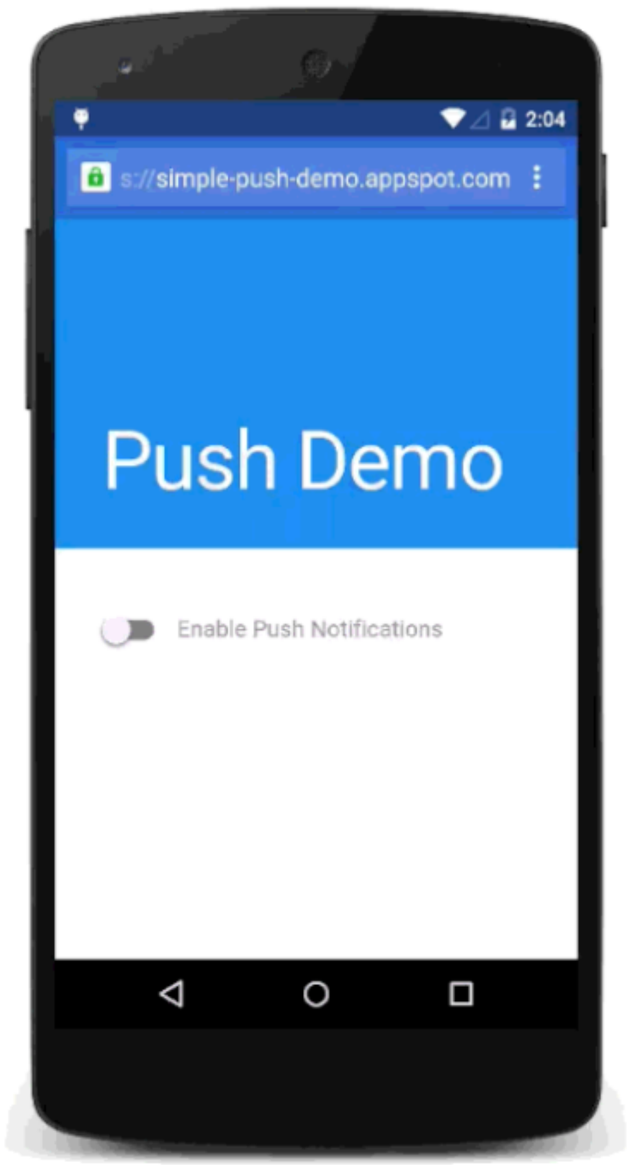
```
<link rel="manifest" href="manifest.webmanifest"/>
```

to the `index.html` in `head` section

IndexedDB

- Local noSQL database
- Good for large data blobs such as files
- Supports transactions

Push Messages



Support

- caniuse.com:
  - Service Worker
  - Push API
  - Web App Manifest
  - IndexedDB
- Is Service Worker Ready?

WebKit (Apple) Support

Software

Apple signals it's willing to let next-gen web apps compete with iOS apps

Service Workers land in WebKit, clearing way for better in-browser applications

By Thomas Claburn in San Francisco 4 Aug 2017 at 20:26 23 SHARE ▼

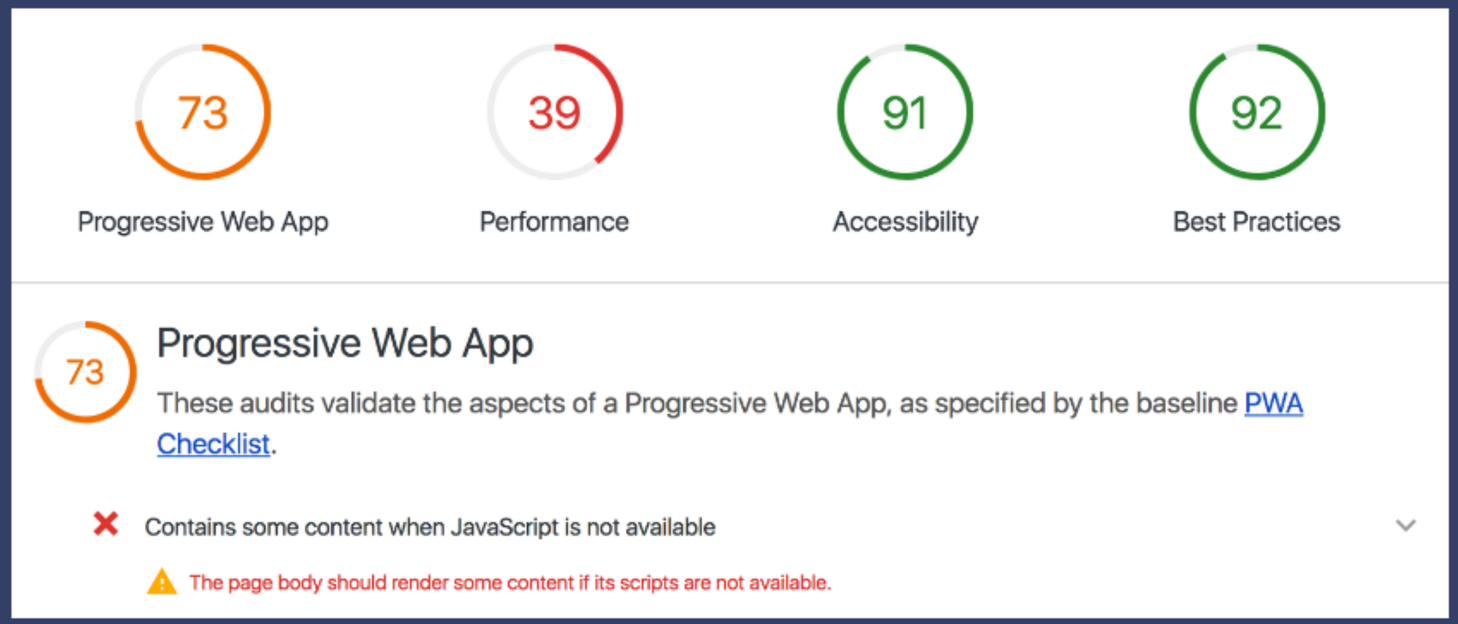
Greg Blass Follow

Software Engineer and Web/App Developer. DreamIt Philly Alum. Lover of Ruby on Rails, React/React Native, Music, Craft Beer and Jessi

Jul 24 · 8 min read

Apple's refusal to support Progressive Web Apps is a detriment to future of the web

Lighthouse - PWA assessment tool



Exercise

Run Lighthouse for your application, you should get score of 64 for PWA section

E2E Testing

- What is it?
- Technologies
- Page objects
- Promise manager
- Chrome Headless
- Different philosophies

What Is It?

- End to End testing
- Tests the system as whole by simulating user interaction via the browser
- Can verify that the DOM looks like it should
- Usually just the "happy paths"

Technologies used

- Protractor
- Any browser supported by Protractor (Chrome by default)
- Jasmine for test case declaration and expectations

Protractor

- E2E test framework originally made for Angular.js
- Runs tests in supported browsers
- Automatically waits for all pending tasks (HTTP requests, timers) on Angular
- Based on WebDriver and thus Selenium

Protractor - Example

```
describe('angularjs homepage todo list', function() {
  it('should add a todo', function() {
    browser.get('https://angularjs.org');

    element(by.model('todoList.todoText')).sendKeys('write first protractor test')
    element(by.css('[value="add"]')).click();

    var todoList = element.all(by.repeater('todo in todoList.todos'));
    expect(todoList.count()).toEqual(3);
    expect(todoList.get(2).getText()).toEqual('write first protractor test');

    // You wrote your first test, cross it off the list
    todoList.get(2).element(by.css('input')).click();
    var completedAmount = element.all(by.css('.done-true'));
    expect(completedAmount.count()).toEqual(2);
  });
});
```

Page Objects

Page object encapsulates the page access

```
import { browser, by, element } from 'protractor';

export class AppPage {
  navigateTo() {
    return browser.get('/');
  }

  getParagraphText() {
    return element(by.css('app-root h1')).getText();
  }
}
```

Corresponding Test

```
import { AppPage } from './app.po';

describe('koulutus App', () => {
  let page: AppPage;

  beforeEach(() => {
    page = new AppPage();
  });

  it('should display welcome message', () => {
    page.navigateTo();
    expect(page.getParagraphText()).toEqual('Welcome to app!');
  });
});
```

Debugging

- Pausing for debugging
- Sleep
- Interactive mode
- Screenshots

Promise Manager

- The promises are queued by default
- Leads to odd behavior:

```
it('should find an element by text input model', function() {
browser.get('app/index.html#/form');

var username = element(by.model('username'));
username.clear();
username.sendKeys('Jane Doe');

var name = element(by.binding('username'));

expect(name.getText()).toEqual('Jane Doe');

// Point A
});
```

- Disabled in upcoming Selenium 4.0

Async/await Without Promise Manager

```
it('should find an element by text input model', async function() {
    await browser.get('app/index.html#/form');

    var username = await element(by.model('username'));
    await username.clear();
    await username.sendKeys('Jane Doe');

    var name = await element(by.binding('username'));

    await expect(name.getText()).toEqual('Jane Doe');

    // Point A
});
```

Turning Promise Manager Off

protractor.conf.js:

```
exports.config = {
  ...,
  SELENIUM_PROMISE_MANAGER: false
};
```

Different Approaches

Test philosophies:

- Just the UI
- Full system
- Something in the middle

Just the UI

Pros:

- Fast
- No random failures

Cons:

- Only assesses the UI
- Requires a lot of mocking

Full System

Pros:

- Proves that actual system works as intended

Cons:

- Random failures
- Take lots of time
- How to handle external services such as email sending?

Learnt in Practice

- Use Chrome Headless for running (easy installation and repetition)
- Use Chrome or screenshots for debugging
- Use Docker (constant environment)
- Only fail after N failures
- Run every N minutes or on each commit (depending on E2E test duration etc.)
- Incremental tests instead of reset between each test file

Chrome Headless

- Chrome without window -> can be ran from command line
- Supports DOM manipulation, JS execution, screenshots, PDF printing etc.
- Support in Chrome 59 (Linux & Mac OS X) and 60 (Windows)
- Modern day PhantomJS

Chrome Headless & Protractor

protractor.conf.js:

```
exports.config = {
    capabilities: {
        'browserName': 'chrome',
        'chromeOptions': {
          'args': [
            '--disable-gpu',
            '--headless',
            '--window-size=1280,1696'
          ]
        }
    },
    directConnect: true
};
```

Pitfalls

- Intervals in Angular code
- Managing login