Unexpected
random photo sharing for authentic social media

Alexi Christakis
supervised by: Ruzica Piskac
December 11, 2019
Project Description

1.1 Abstract

One of the principle goals of social media is to bring humans closer together. Through direct messaging, status updates, and multi-media content sharing, technologies that have surged in the Web 2.0 have evolved to meet this objective. It's easier now than ever before to connect, yet social media remains alienating. Users experience a social pressure to cultivate profiles that put forward the "coolest" and highest versions of themselves that end up exacerbating the anxiety of others. Users are forced to discriminate between moments they'd like to share based on how others might perceive them, rather than feeling empowered to share authentic experiences from their lives.

Unexpected is an experiment on how we think about social media, and what it means to be genuine in traditional methods of content sharing. Taking inspiration from Emotional Sampling Method (a tool from psychology studies where participants record their mood at a random time), users receive a notification once a day that unlocks the app's camera for 10 minutes for them to take and share a photo. Posts can be shared only within that window, and the notification comes at a random time each day. Content is therefore forced to be temporal, authentic, and hopefully more intimate.

1.2 Introduction

Tech companies have a history of launching new features and products to increase the immediacy and intimacy of their platforms. Snapchat gained immense traction for its novel approach to messaging with ephemeral digital photos, which made the communication medium feel more immediate and trustworthy. Building on this success, they launched text-based chat (with the same ephemerality), video messages, video chat, filters, and stories. A story is a collection of multi-media posts shared by a user to all of their friends for a 24 hour period. Each post has its own timer, but can be watched multiple times. The idea of a “story” has been ported to nearly every other popular social media platform in the years following due to the immense success the feature brought Snapchat. The thought behind the story is to allow users to share elements of their day to a large audience effortlessly. Users can then watch snippets of their friends’ days, and live vicariously through their camera lens.

To ensure that stories and messages were “truthful,” Snapchat (unlike Instagram) required all content be recorded through the app, a policy that guaranteed anything shared occurred at the recorded timestamp. Snapchat seemed more intimate than competitors because its content wasn’t pre-recorded or fabricated. It felt more immediate because a photo posted to a story 30 minutes ago happened 30 minutes ago. The user’s agency of choice is diminished, unlike on a photo sharing service like Instagram (though in early versions photos also had to be taken inside of the app) where there’s no verification on a post’s timestamp or geotag. There, users carefully select experiences to share that appear the most interesting, look the most aesthetic, and have the greatest potential to make their followers jealous. Multiple photos from a single experience or trip can be disjointed, disproportionally representing their life. The resulting feedback loop where users are constantly comparing their lives to false representations of those they follow is a primary contributor to the social distress that has triggered backlash against social media in recent years.

Unexpected aims to counter the toxic trends of traditional social media by providing a system where users surrender control of the timing of the photos they share. This project is primarily a social experiment to test whether people are receptive to a drastic change in how they interact with social media, and technology as a means to share and communicate with others. I believe that relinquishing control over when it’s possible to share will democratize users’ photo streams, and ideally serve as an authentic complement to the behaviors observed on rival platforms.

The following pages document the codebase behind Unexpected, and why certain choices in its implementation were made.
Client

2.1 Description

A cross-platform React Native app written entirely in TypeScript. Powered by Redux (with various middlewares detailed below), React Navigation, CodePush, and Axios. Once a day, users receive a notification at a random time which unlocks the camera screen and allows them to share a photo.

![Unexpected icon design](image)

Figure 2.1: Unexpected icon design

2.2 Deployment

I opted to use Microsoft's tool CodePush to deliver seamless over-the-air updates to all devices. Multiple deployments can be managed at once, with different JavaScript binaries assigned to each one. Currently, Unexpected uses a staging and production environment. An integration with allow for continuous deployment whenever new code is merged and passes all tests.

Unexpected is live on TestFlight for beta distribution which runs the production CodePush deployment, with my own device running a staging release so that I can test new code on a real device (primarily for Redux state migrations) quickly before upgrading the bundle to more devices.

2.3 Application Logic

Unexpected uses Redux for state management and application logic. This abstraction removes all logic from the React components, leaving them entirely presentational. Any local state managed by a component is limited to form data or animation control. This also allows for easier testing, as by changing the provided properties to any component all behavior that would come from Redux can be mocked in a controlled manner.

Every component also utilizes the new React syntax for functional components, hooks, and memoization to remove unnecessary re-renders. With carefully managed property equality functions, components only redraw when properties that necessitate an update change. This pattern solves the issue of where a higher-order-component updates and causes cascading re-renders down the component tree. With many components also connected to the Redux store new data can be propagated to many components in a single render cycle.
2.3.1 Redux

Redux is a global state-management tool that follows an Action & Reducer pattern. The application’s state is broken into subtrees that represent different data modules, and a set of functions deterministically update them. Each module has an associated reducer which defines how each dispatched action translates to an updated copy of the state. The main benefits of the Redux pattern include:

1. **Predictability**: any state mutation has a corresponding action and case in the module’s reducer. Behavior is guaranteed to be consistent regardless of where an action is dispatched from.

2. **Centralization**: all state mutations can be found in the same directory in the same pattern. Since they all abide by the rules of Redux, testing, logging, and persistence are made much easier.

3. **Debuggability**: another powerful benefit of the predictability and centralization is easier debugging. With some easy-to-configure middleware every dispatched action and state mutation can be logged.

4. **Flexibility**: because Redux centralizes all state mutations, they can be injected into any component. This abstraction from component-level state mutations makes updating presentational aspects of an application painless.

For Unexpected, I opted to use Redux because my previous experience with the library gave me comfort in implementing it in a new project. Where I had a new opportunity to learn though is in integrating it with TypeScript—all of the benefits I hoped to gain from TypeScript hinged upon a seamless and concise integration. To accomplish this, I wrote a series of helper functions that make Redux actions and reducers type-aware. When dispatching an action created from `createAction()` TypeScript guarantees that the payload passed is of the correct type, and that every reducer case handles that payload properly.

```typescript
interface Action<T extends string> {
    type: T;
}

interface ActionWithPayload<T extends string, P> extends Action<T> {
    payload: P;
}

export function createAction<T extends string>(type: T): Action<T>;
export function createAction<T extends string, P>(
    type: T,
    payload: P
): ActionWithPayload<T, P>;

export function createAction<T extends string, P>(type: T, payload?: P) {
    return payload === undefined ? { type } : { type, payload };
}

export type ActionsUnion<A extends ActionCreatorsMapObject> = ReturnType<A[ keyof A] >;

export type ExtractActionFromActionCreator<AC> = AC extends () => infer A
    ? A
    : AC extends (payload: any) => infer A
    ? A
    : AC extends (payload: any, error: any) => infer A
    ? A
    : never;
```
Below is an example from the user Redux module where the `createAction()` function is used to create an object of all the actions for the module. This object is then imported into components that depend upon the modules functionality.

```javascript
export enum ActionTypes {
    FETCH_USERS_POSTS = "post/FETCH_USERS_POSTS",
    FETCH_USERS_POSTS_SUCCESS = "post/FETCH_USERS_POSTS_SUCCESS",
    FETCH_FEED = "post/FETCH_FEED",
    FETCH_FEED_SUCCESS = "post/FETCH_FEED_SUCCESS",
    SEND_POST = "post/SEND_POST",
    SEND_POST_SUCCESS = "post/SEND_POST_SUCCESS",
    ON_ERROR = "post/ON_ERROR"
}

export const Actions = {
    fetchUsersPosts: (phoneNumber?: string) =>
    createAction(ActionTypes.FETCH_USERS_POSTS, { phoneNumber }),
    fetchUsersPostsSuccess: (phoneNumber: string, posts: PostType[]) =>
    createAction(ActionTypes.FETCH_USERS_POSTS_SUCCESS, { phoneNumber, posts }),
    fetchFeed: (fromDate?: Date) =>
    createAction(ActionTypes.FETCH_FEED, { fromDate }),
    fetchFeedSuccess: (posts: FeedReturnType) =>
    createAction(ActionTypes.FETCH_FEED_SUCCESS, { posts }),
    sendPost: (description: string) =>
    createAction(ActionTypes.SEND_POST, { description }),
    sendPostSuccess: (phoneNumber: string) =>
    createAction(ActionTypes.SEND_POST_SUCCESS, { phoneNumber }),
    onError: (error: string) =>
    createAction(ActionTypes.ON_ERROR, { error })
};
```

Below is the full data tree for Unexpected shown as a TypeScript interface. The application is divided into the following modules: app, which tracks high-level state like network connectivity, whether the app is backgrounded, and if the camera is enabled; auth, which stores the JWT and loading states for the sign-up page; user, which contains user entities and the current user’s phone number; permissions, for granted permissions; image, for image caching; and post, for post data.
Unexpected Redux Data-Store Model

```javascript
root {
  app: {
    appStatus: AppStatusType; // backgrounded | inactive | active
    networkStatus: NetInfoState; // internet access
    camera: { enabled: boolean; timeOfExpiry?: Moment; }
  };
  auth: {
    loading: boolean;
    phoneNumber: string;
    isAwaitingCode: boolean;
    authError: string;
    jwt: string | null;
  };
  user: {
    phoneNumber: string;
    users: { [phoneNumber: string]: UserType }; // cache of all downloaded user entities
    loading: boolean;
    error: any;
  };
  permissions: { // what permissions have been granted by the user
    loading: boolean;
    notifications: NotificationsResponse;
    camera: PermissionStatus;
    location: PermissionStatus;
    contacts: PermissionStatus;
    error: string;
  };
  image: {
    currentImage: TakePictureResponse | null;
    uploading: boolean;
    uploadError: any;
    cache: { // image caching system
      profile: { [phoneNumber: string]: {
        ts: number;
        uri: string; // file path to downloaded image on device
        fallback: string;
      };
    };
    feed: { [phoneNumber: string]: {
      [id: string]: {
        ts: number;
        uri: string; // file path to downloaded image on device
        fallback: string;
      };
    };
  };
  post: {
    users: { [phoneNumber: string]: {
      posts: PostType[];
      lastFetched: Date;
      stale: boolean;
    };
  };
  feed: {
    posts: FeedPostType[];
    lastFetched: Date;
    stale: boolean;
  };
  loading: boolean;
  error: string;
};
```
2.3.2 JavaScript Generators

The main drawback of using a state management pattern like Redux is that all state mutations must be run synchronously. Of course, in most applications a lot of state updates rely on asynchronous tasks like data-fetching. There are a number of available middlewares for Redux that allow for asynchronous code to dispatch actions to the data store. Of these I opted to use a library called Redux-Saga, which is built on top of the JavaScript ES6 Generator syntax.

Generators are essentially functions which produce a sequence of results rather than a single one. They return an object with functions to advance the generator to the next yield call. In an integration with Redux, generators are run as side-effects to action dispatches. This allows for asynchronous code to be executed as the result of a normal Redux action. Here’s an example from the user Redux module of Unexpected, where the generator `userSagas` subscribes individual generators to different action types. The generator `onUpdateUser` is subscribed to the `UPDATE_USER` action type, and is then passed the action as an argument. `onUpdateUser` can dispatch any number of actions from its body after the asynchronous code is complete. In this example, a PATCH request is made to the backend to update the corresponding user record, and when the promise resolves an action is dispatched to update the local state to reflect the change.

```javascript
export function* userSagas() {
    yield all([
        yield takeLatestActionTypes.FETCH_USER, onFetchUser),
        yield takeLatestActionTypes.FETCH_USERS, onFetchUsers),
        yield takeLatestActionTypes.CREATE_NEW_USER, onCreateUser),
        yield takeLatestActionTypes.UPDATE_USER, onUpdateUser),
    ]);}

function* onUpdateUser(action: ExtractActionFromActionCreator<typeof Actions.updateUser> ) {
    const jwt = yield select(selectors.jwt);
    const phoneNumber = yield select(selectors.phoneNumber);

    const { user } = action.payload;

    try {
        const res: AxiosResponse<UserType> = yield client.patch('/user/${phoneNumber}',
            { user: { ...user } },
            { headers: getHeaders({ jwt })
        });

        const { data } = res;

        yield put(Actions.loadUsers([data]));
    } catch (err) {
        yield put(Actions.onError(err));
    }
}
```
The user module reducer simply switches to a loading state when `UPDATE_USER` is dispatched so that presentational components display a loading indicator, and the `loadUsers` action turns it off.

### 2.3.3 Persistence to Local Storage

Another helpful piece of middleware I implemented in Unexpected is Redux-Persist. State persistence is crucial to a seamless and uninterrupted user experience, as the application shouldn’t prompt a user to sign in upon every launch. Redux-Persist allows for the entire Redux store to be serialized and cached into the device’s local storage. The root component is wrapped in a `<Persistor />` component which delays the first render until the store is persisted. Upon persistence, the initial route is determined based on the user’s authentication status.

### 2.4 Navigation

Navigation is powered by the open source library React Navigation, specifically the new experimental dynamic API. Routes are declared using JSX tags, with navigator and screen components exposed by the library.

For Unexpected, the root navigator is based on tab navigation, with a stack within each tab to navigate between the root screen for that tab, post screens, and profile screens. This navigator is then encompassed in a switch navigator which alternates between the authenticated and unauthenticated roots.

An example of how this dynamic API works. Nested React components determine how different routes are grouped and layered. This is the top level navigation router for Unexpected. The `isAuthorized` variable is a derived property from the Redux store which checks for a valid JWT and user entity.

```jsx
<NavigationNativeContainer ref={Navigation.setTopLevelNavigator}>
  <Stack.Navigator
    screenOptions={{ animation: "fade" }}
    initialRouteName={isAuthorized ? "AUTHENTICATED" : "UNAUTHENTICATED"}>
    <Stack.Screen
      name="AUTHENTICATED"
      options={{ headerShown: false }}
      component={AuthenticatedRoot}
    />
    <Stack.Screen
      name="UNAUTHENTICATED"
      options={{ headerShown: false }}
      component={UnathenticatedRoot}
    />
  </Stack.Navigator>
</NavigationNativeContainer>
```
2.4.1 Authentication: Stack Navigator

Once a phone number is entered, a new text field appears for the verification code to be entered. All textfields are wrapped in keyboard-aware views so that touch-active zones are not blocked by the keyboard launching.

2.4.2 Home: Tab Navigator

Each of the home screen tabs contains its own stack of profile and post screens on top of their root screens of the Feed, User Profile, and Search. Tapping a post or user in any screen pushes the corresponding screen to that tab’s stack.

2.4.3 Capture: Stack Navigator

These two screens are only accessible when the day’s random notification unlocks them. The navigation stack is pushed as a modal on top of the home screen tab navigator, and is activated by a button which appears above the tab bar. The stack can be dismissed by swiping down. It can be relaunched until the 10-minute timer has counted down completely. Once a post is successfully shared, the stack is dismissed automatically.

2.4.4 Other Screens

The other screens available to the Unexpected user are settings, profile modification (bio and profile picture), permissions, and friends (accessible by tapping the “friends” from a user’s profile).
Figure 2.4: Feed

Figure 2.5: Profile

Figure 2.6: Search
2.5 Testing

2.5.1 Unit
Unit tests which perform both snapshot comparisons and basic behavioral validation are written for each component. Jest—a JavaScript testing framework written by Facebook—runs the tests and provides syntax for separating tests into blocks. Enzyme—a library written by AirBnb—provides helpful tools for testing React components like searching for child elements, updating state, changing props, etc.

2.5.2 Integration
Detox—a library written by Wix—provides powerful tools for end-to-end testing of native mobile applications. Test cases are written much like user stories, with Detox providing functions to find and simulate the press of buttons, text entry fields, and even gestures.
3.1 Description

A NodeJS Express app connected to a MongoDB atlas instance for user data persistence and a Google Cloud Platform storage instance for photo storage. This application is currently hosted by Heroku. Written entirely in TypeScript, model types are exposed to the client app for consistency and type-safety. The application is also written with a library called TypeScript Express Decorators, which handles service and endpoint injection, routing, and makes middleware creation easy.

3.2 Services

The application is comprised of a number of services that handle data mutations and interfacing with outside services (like data storage, notifications, and text messages). The main services are described here.

3.2.1 Scheduler

The most important part of the experience of Unexpected is the random notification which unlocks the client’s camera. To accomplish this goal, Unexpected’s backend uses a scheduler service which runs a cron job once a day to generate a random time for each user to receive their notification. The time generation is timezone aware, and randomly picks a time between 10am and 10pm. A job is then scheduled for each of these jobs to send a notification to the corresponding user. Extensions to this service include exposing more granular control to the user for them to specify when they’d like to be notified.

3.2.2 Notification

The notification service interfaces with two push notification instances: Apple’s Push Notification System, and Google Cloud Messaging. When a push notification is requested, the user’s device OS is checked in order to use the correct service.

3.2.3 Twilio

The Twilio service exposes functions to send text messages. Used by the authentication service and endpoint to send verification text messages to users requesting to authenticate their client instance.

3.2.4 Mongo Models

Each Mongo document type is supported by its own service which handles all data mutations. They all inherit an abstract CRUD (Create, Read, Update, Delete) class which provides specific type-safe handlers for basic operations. Each class instance then implements more complex requests that may also depend upon other document types. Abstracting the connection to each document collection yields a number of benefits including: safer types, less redundant code, simplified endpoint methods, easier testing.

3.2.5 Image

The Image service handles all image uploads and downloads. It hosts a connection to the Google Cloud Storage instance, and handles the opening and closing of read/write streams to individual files. Methods are also available for resizing or compressing images.
3.3 Endpoints

The following REST endpoints are made available to the Unexpected client to empower user stories such as updating their information, friending other users, sending posts, and generating their feed. Nearly every endpoint is secured using the web standard JSON Web Token (JWT). Each user is granted a unique token hashed from a private key held by the cloud instance, and their phone number. All REST requests are routed through the base url: https://api.expect.photos/

3.3.1 verify

/:phoneNumber

POST: Sends a text message to the given phone number with a single-use verification code which expires after 10 minutes. An encrypted copy of the generated code is stored as a "verification-message" type document in Mongo for retrieval later when a subsequent request is made to verify a code.

/:phoneNumber/:code

POST: Retrieves the verification-message document from Mongo and compares the encrypted versions of both codes for equality. If they are not equal the request ends by sending a response to the client of the shape: 

`{verified:false}`. If they are equal, Mongo is queried to see if there exists a user entity under the requested phone number so it can be returned. The final return is of the shape: 

`{verified:true, user?:userEntity, token:JWT}`, where the token is a JWT hashed from a private key and the user's phone number.

3.3.2 user

PUT: Creates a user from the supplied data in body. Authentication middleware checks to ensure that the user that's requested to be created matches who the requester is authenticated as (by checking the phone number encrypted in the JWT).

/user/search/:query

GET: Search the user document collection for all matches to :query. Uses regular expressions to parse and search first and last names in a case-insensitive manner.

/:phoneNumber

GET: Returns the requested (:phoneNumber) user entity.

PATCH: Updates the requested (:phoneNumber) user entity with data supplied in body. Authentication middleware checks to ensure that the user requesting the mutation matches.

/user/:phoneNumber/friend/:to

PATCH: Send a friend request from :phoneNumber to :to. Also sends a notification to :to alerting them to their new friend request.

/user/:phoneNumber/accept/:to

PATCH: Accept a friend request from :phoneNumber to :to. Also sends a notification to :phoneNumber alerting them that their request was accepted.

/user/:phoneNumber/cancel/:to

PATCH: Cancel a friend request from :phoneNumber to :to.

/user/:phoneNumber/deny/:to

PATCH: Deny a friend request from :phoneNumber to :to.
PATCH: Delete the friendship between :phoneNumber and :to.

3.3.3 post

/:phoneNumber

PUT: Send a post by :phoneNumber. Retrieves post data from :body.
GET: Get all posts sent by :phoneNumber. Used in the profile page in the client.

/:phoneNumber/feed

GET: Queries post document collection for all recent posts made by users followed by the user :phoneNumber.

3.3.4 image

For all image uploads, the image is loaded into application memory as a multi-part file, converted into a read-stream, and piped to a write-stream connected to the Google Cloud Storage instance. For all image downloads, a buffer is downloaded from GCP Storage, and returned to the client as a binary file.

/:phoneNumber

PUT: Updates the profile picture :phoneNumber with the multi-part file read into application memory.
GET: Downloads :phoneNumber's profile picture.

/:phoneNumber/:id

PUT: Uploads a new post image with the id :id for the user :phoneNumber.
GET: Downloads the post image with the id :id for the user :phoneNumber.

3.4 Testing

3.4.1 Unit

Unit tests are performed at the level of the service. My backend implements a number of abstracted services which interface with both the database and GCP storage bucket

3.4.2 Integration

Integration tests are built using the library Supertest—a high-level abstraction for testing HTTP. Test cases using Supertest make assertions against what endpoints will return. With a mocked Mongo environment, endpoints can be tested to ensure the data they return is correct. These tests are run at the level of the controllers—the endpoint routes.
Conclusion

Through the development of Unexpected I had the opportunity to work with backend development, TypeScript, and end-to-end architecture design. I learned how to build an Express app for a REST API, use JWT for authentication, host images in a storage bucket, set up endpoints to upload and download files, use MongoDB, and set up a job scheduler.

In development of the front-end, I learned how to utilize the new features of React 16.8, including hooks and the new functional component syntax. I thought critically about what patterns to implement in the control of application logic, and decided to learn how to use the JavaScript Generator.

One of the most satisfying parts of this project (from a software engineering perspective) was how TypeScript boosted my velocity by eliminating all type errors during development. I opted for a mono-repo structure so that all API requests from the React Native app could be strictly-typed by inferring endpoint return-types. Overall, I’m very satisfied with the robustness of the software I have developed, and look forward to conducting a more formal launch next semester to gather feedback of this non-traditional method of photo sharing.