package com.twitter.tweetypie

package hydrator

import com.twitter.expandodo.thriftscala.Card

import com.twitter.expandodo.thriftscala.Card2

import com.twitter.servo.cache.Cached

import com.twitter.servo.cache.CachedValueStatus

import com.twitter.servo.cache.LockingCache

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.media.thriftscala.MediaRef

import com.twitter.tweetypie.repository.PastedMedia

import com.twitter.tweetypie.repository.TweetQuery

import com.twitter.tweetypie.repository.TweetRepoCachePicker

import com.twitter.tweetypie.repository.TweetResultRepository

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.tweetypie.util.Takedowns

import com.twitter.util.Return

import com.twitter.util.Throw

object TweetHydration {

/\*\*

\* Wires up a set of hydrators that include those whose results are cached on the tweet,

\* and some whose results are not cached but depend upon the results of the former.

\*/

def apply(

hydratorStats: StatsReceiver,

hydrateFeatureSwitchResults: TweetDataValueHydrator,

hydrateMentions: MentionEntitiesHydrator.Type,

hydrateLanguage: LanguageHydrator.Type,

hydrateUrls: UrlEntitiesHydrator.Type,

hydrateQuotedTweetRef: QuotedTweetRefHydrator.Type,

hydrateQuotedTweetRefUrls: QuotedTweetRefUrlsHydrator.Type,

hydrateMediaCacheable: MediaEntitiesHydrator.Cacheable.Type,

hydrateReplyScreenName: ReplyScreenNameHydrator.Type,

hydrateConvoId: ConversationIdHydrator.Type,

hydratePerspective: PerspectiveHydrator.Type,

hydrateEditPerspective: EditPerspectiveHydrator.Type,

hydrateConversationMuted: ConversationMutedHydrator.Type,

hydrateContributor: ContributorHydrator.Type,

hydrateTakedowns: TakedownHydrator.Type,

hydrateDirectedAt: DirectedAtHydrator.Type,

hydrateGeoScrub: GeoScrubHydrator.Type,

hydrateCacheableRepairs: TweetDataValueHydrator,

hydrateMediaUncacheable: MediaEntitiesHydrator.Uncacheable.Type,

hydratePostCacheRepairs: TweetDataValueHydrator,

hydrateTweetLegacyFormat: TweetDataValueHydrator,

hydrateQuoteTweetVisibility: QuoteTweetVisibilityHydrator.Type,

hydrateQuotedTweet: QuotedTweetHydrator.Type,

hydratePastedMedia: PastedMediaHydrator.Type,

hydrateMediaRefs: MediaRefsHydrator.Type,

hydrateMediaTags: MediaTagsHydrator.Type,

hydrateClassicCards: CardHydrator.Type,

hydrateCard2: Card2Hydrator.Type,

hydrateContributorVisibility: ContributorVisibilityFilter.Type,

hydrateHasMedia: HasMediaHydrator.Type,

hydrateTweetCounts: TweetCountsHydrator.Type,

hydratePreviousTweetCounts: PreviousTweetCountsHydrator.Type,

hydratePlace: PlaceHydrator.Type,

hydrateDeviceSource: DeviceSourceHydrator.Type,

hydrateProfileGeo: ProfileGeoHydrator.Type,

hydrateSourceTweet: SourceTweetHydrator.Type,

hydrateIM1837State: IM1837FilterHydrator.Type,

hydrateIM2884State: IM2884FilterHydrator.Type,

hydrateIM3433State: IM3433FilterHydrator.Type,

hydrateTweetAuthorVisibility: TweetAuthorVisibilityHydrator.Type,

hydrateReportedTweetVisibility: ReportedTweetFilter.Type,

scrubSuperfluousUrlEntities: TweetDataValueHydrator,

copyFromSourceTweet: TweetDataValueHydrator,

hydrateTweetVisibility: TweetVisibilityHydrator.Type,

hydrateEscherbirdAnnotations: EscherbirdAnnotationHydrator.Type,

hydrateScrubEngagements: ScrubEngagementHydrator.Type,

hydrateConversationControl: ConversationControlHydrator.Type,

hydrateEditControl: EditControlHydrator.Type,

hydrateUnmentionData: UnmentionDataHydrator.Type,

hydrateNoteTweetSuffix: TweetDataValueHydrator

): TweetDataValueHydrator = {

val scrubCachedTweet: TweetDataValueHydrator =

ValueHydrator

.fromMutation[Tweet, TweetQuery.Options](

ScrubUncacheable.tweetMutation.countMutations(hydratorStats.counter("scrub\_cached\_tweet"))

)

.lensed(TweetData.Lenses.tweet)

.onlyIf((td, opts) => opts.cause.reading(td.tweet.id))

// We perform independent hydrations of individual bits of

// data and pack the results into tuples instead of updating

// the tweet for each one in order to avoid making lots of

// copies of the tweet.

val hydratePrimaryCacheableFields: TweetDataValueHydrator =

ValueHydrator[TweetData, TweetQuery.Options] { (td, opts) =>

val ctx = TweetCtx.from(td, opts)

val tweet = td.tweet

val urlsMediaQuoteTweet: Stitch[

ValueState[(Seq[UrlEntity], Seq[MediaEntity], Option[QuotedTweet])]

] =

for {

urls <- hydrateUrls(getUrls(tweet), ctx)

(media, quotedTweet) <- Stitch.join(

hydrateMediaCacheable(

getMedia(tweet),

MediaEntityHydrator.Cacheable.Ctx(urls.value, ctx)

),

for {

qtRef <- hydrateQuotedTweetRef(

tweet.quotedTweet,

QuotedTweetRefHydrator.Ctx(urls.value, ctx)

)

qtRefWithUrls <- hydrateQuotedTweetRefUrls(qtRef.value, ctx)

} yield {

ValueState(qtRefWithUrls.value, qtRef.state ++ qtRefWithUrls.state)

}

)

} yield {

ValueState.join(urls, media, quotedTweet)

}

val conversationId: Stitch[ValueState[Option[ConversationId]]] =

hydrateConvoId(getConversationId(tweet), ctx)

val mentions: Stitch[ValueState[Seq[MentionEntity]]] =

hydrateMentions(getMentions(tweet), ctx)

val replyScreenName: Stitch[ValueState[Option[Reply]]] =

hydrateReplyScreenName(getReply(tweet), ctx)

val directedAt: Stitch[ValueState[Option[DirectedAtUser]]] =

hydrateDirectedAt(

getDirectedAtUser(tweet),

DirectedAtHydrator.Ctx(

mentions = getMentions(tweet),

metadata = tweet.directedAtUserMetadata,

underlyingTweetCtx = ctx

)

)

val language: Stitch[ValueState[Option[Language]]] =

hydrateLanguage(tweet.language, ctx)

val contributor: Stitch[ValueState[Option[Contributor]]] =

hydrateContributor(tweet.contributor, ctx)

val geoScrub: Stitch[ValueState[(Option[GeoCoordinates], Option[PlaceId])]] =

hydrateGeoScrub(

(TweetLenses.geoCoordinates(tweet), TweetLenses.placeId(tweet)),

ctx

)

Stitch

.joinMap(

urlsMediaQuoteTweet,

conversationId,

mentions,

replyScreenName,

directedAt,

language,

contributor,

geoScrub

)(ValueState.join(\_, \_, \_, \_, \_, \_, \_, \_))

.map { values =>

if (values.state.isEmpty) {

ValueState.unmodified(td)

} else {

values.map {

case (

(urls, media, quotedTweet),

conversationId,

mentions,

reply,

directedAt,

language,

contributor,

coreGeo

) =>

val (coordinates, placeId) = coreGeo

td.copy(

tweet = tweet.copy(

coreData = tweet.coreData.map(

\_.copy(

reply = reply,

conversationId = conversationId,

directedAtUser = directedAt,

coordinates = coordinates,

placeId = placeId

)

),

urls = Some(urls),

media = Some(media),

mentions = Some(mentions),

language = language,

quotedTweet = quotedTweet,

contributor = contributor

)

)

}

}

}

}

val assertNotScrubbed: TweetDataValueHydrator =

ValueHydrator.fromMutation[TweetData, TweetQuery.Options](

ScrubUncacheable

.assertNotScrubbed(

"output of the cacheable tweet hydrator should not require scrubbing"

)

.lensed(TweetData.Lenses.tweet)

)

val hydrateDependentUncacheableFields: TweetDataValueHydrator =

ValueHydrator[TweetData, TweetQuery.Options] { (td, opts) =>

val ctx = TweetCtx.from(td, opts)

val tweet = td.tweet

val quotedTweetResult: Stitch[ValueState[Option[QuotedTweetResult]]] =

for {

qtFilterState <- hydrateQuoteTweetVisibility(None, ctx)

quotedTweet <- hydrateQuotedTweet(

td.quotedTweetResult,

QuotedTweetHydrator.Ctx(qtFilterState.value, ctx)

)

} yield {

ValueState.join(qtFilterState, quotedTweet).map(\_.\_2)

}

val pastedMedia: Stitch[ValueState[PastedMedia]] =

hydratePastedMedia(

PastedMediaHydrator.getPastedMedia(tweet),

PastedMediaHydrator.Ctx(getUrls(tweet), ctx)

)

val mediaTags: Stitch[ValueState[Option[TweetMediaTags]]] =

hydrateMediaTags(tweet.mediaTags, ctx)

val classicCards: Stitch[ValueState[Option[Seq[Card]]]] =

hydrateClassicCards(

tweet.cards,

CardHydrator.Ctx(getUrls(tweet), getMedia(tweet), ctx)

)

val card2: Stitch[ValueState[Option[Card2]]] =

hydrateCard2(

tweet.card2,

Card2Hydrator.Ctx(

getUrls(tweet),

getMedia(tweet),

getCardReference(tweet),

ctx,

td.featureSwitchResults

)

)

val contributorVisibility: Stitch[ValueState[Option[Contributor]]] =

hydrateContributorVisibility(tweet.contributor, ctx)

val takedowns: Stitch[ValueState[Option[Takedowns]]] =

hydrateTakedowns(

None, // None because uncacheable hydrator doesn't depend on previous value

TakedownHydrator.Ctx(Takedowns.fromTweet(tweet), ctx)

)

val conversationControl: Stitch[ValueState[Option[ConversationControl]]] =

hydrateConversationControl(

tweet.conversationControl,

ConversationControlHydrator.Ctx(getConversationId(tweet), ctx)

)

// PreviousTweetCounts and Perspective hydration depends on tweet.editControl.edit\_control\_initial

// having been hydrated in EditControlHydrator; thus we are chaining them together.

val editControlWithDependencies: Stitch[

ValueState[

(

Option[EditControl],

Option[StatusPerspective],

Option[StatusCounts],

Option[TweetPerspective]

)

]

] =

for {

(edit, perspective) <- Stitch.join(

hydrateEditControl(tweet.editControl, ctx),

hydratePerspective(

tweet.perspective,

PerspectiveHydrator.Ctx(td.featureSwitchResults, ctx))

)

(counts, editPerspective) <- Stitch.join(

hydratePreviousTweetCounts(

tweet.previousCounts,

PreviousTweetCountsHydrator.Ctx(edit.value, td.featureSwitchResults, ctx)),

hydrateEditPerspective(

tweet.editPerspective,

EditPerspectiveHydrator

.Ctx(perspective.value, edit.value, td.featureSwitchResults, ctx))

)

} yield {

ValueState.join(edit, perspective, counts, editPerspective)

}

Stitch

.joinMap(

quotedTweetResult,

pastedMedia,

mediaTags,

classicCards,

card2,

contributorVisibility,

takedowns,

conversationControl,

editControlWithDependencies

)(ValueState.join(\_, \_, \_, \_, \_, \_, \_, \_, \_))

.map { values =>

if (values.state.isEmpty) {

ValueState.unmodified(td)

} else {

values.map {

case (

quotedTweetResult,

pastedMedia,

ownedMediaTags,

cards,

card2,

contributor,

takedowns,

conversationControl,

(editControl, perspective, previousCounts, editPerspective)

) =>

td.copy(

tweet = tweet.copy(

media = Some(pastedMedia.mediaEntities),

mediaTags = pastedMedia.mergeTweetMediaTags(ownedMediaTags),

cards = cards,

card2 = card2,

contributor = contributor,

takedownCountryCodes = takedowns.map(\_.countryCodes.toSeq),

takedownReasons = takedowns.map(\_.reasons.toSeq),

conversationControl = conversationControl,

editControl = editControl,

previousCounts = previousCounts,

perspective = perspective,

editPerspective = editPerspective,

),

quotedTweetResult = quotedTweetResult

)

}

}

}

}

val hydrateIndependentUncacheableFields: TweetDataEditHydrator =

EditHydrator[TweetData, TweetQuery.Options] { (td, opts) =>

val ctx = TweetCtx.from(td, opts)

val tweet = td.tweet

// Group together the results of hydrators that don't perform

// filtering, because we don't care about the precedence of

// exceptions from these hydrators, because the exceptions all

// indicate failures, and picking any failure will be

// fine. (All of the other hydrators might throw filtering

// exceptions, so we need to make sure that we give precedence

// to their failures.)

val hydratorsWithoutFiltering =

Stitch.joinMap(

hydrateTweetCounts(tweet.counts, TweetCountsHydrator.Ctx(td.featureSwitchResults, ctx)),

// Note: Place is cached in memcache, it is just not cached on the Tweet.

hydratePlace(tweet.place, ctx),

hydrateDeviceSource(tweet.deviceSource, ctx),

hydrateProfileGeo(tweet.profileGeoEnrichment, ctx)

)(ValueState.join(\_, \_, \_, \_))

/\*\*

\* Multiple hydrators throw visibility filtering exceptions so specify an order to achieve

\* a deterministic hydration result while ensuring that any retweet has a source tweet:

\* 1. hydrateSourceTweet throws SourceTweetNotFound, this is a detached-retweet so treat

\* the retweet hydration as if it were not found

\* 2. hydrateTweetAuthorVisibility

\* 3. hydrateSourceTweet (other than SourceTweetNotFound already handled above)

\* 4. hydrateIM1837State

\* 5. hydrateIM2884State

\* 6. hydrateIM3433State

\* 7. hydratorsWithoutFiltering miscellaneous exceptions (any visibility filtering

\* exceptions should win over failure of a hydrator)

\*/

val sourceTweetAndTweetAuthorResult =

Stitch

.joinMap(

hydrateSourceTweet(td.sourceTweetResult, ctx).liftToTry,

hydrateTweetAuthorVisibility((), ctx).liftToTry,

hydrateIM1837State((), ctx).liftToTry,

hydrateIM2884State((), ctx).liftToTry,

hydrateIM3433State((), ctx).liftToTry

) {

case (Throw(t @ FilteredState.Unavailable.SourceTweetNotFound(\_)), \_, \_, \_, \_) =>

Throw(t)

case (\_, Throw(t), \_, \_, \_) => Throw(t) // TweetAuthorVisibility

case (Throw(t), \_, \_, \_, \_) => Throw(t) // SourceTweet

case (\_, \_, Throw(t), \_, \_) => Throw(t) // IM1837State

case (\_, \_, \_, Throw(t), \_) => Throw(t) // IM2884State

case (\_, \_, \_, \_, Throw(t)) => Throw(t) // IM3433State

case (

Return(sourceTweetResultValue),

Return(authorVisibilityValue),

Return(im1837Value),

Return(im2884Value),

Return(im3433Value)

) =>

Return(

ValueState

.join(

sourceTweetResultValue,

authorVisibilityValue,

im1837Value,

im2884Value,

im3433Value

)

)

}.lowerFromTry

StitchExceptionPrecedence(sourceTweetAndTweetAuthorResult)

.joinWith(hydratorsWithoutFiltering)(ValueState.join(\_, \_))

.toStitch

.map { values =>

if (values.state.isEmpty) {

EditState.unit[TweetData]

} else {

EditState[TweetData] { tweetData =>

val tweet = tweetData.tweet

values.map {

case (

(sourceTweetResult, \_, \_, \_, \_),

(counts, place, deviceSource, profileGeo)

) =>

tweetData.copy(

tweet = tweet.copy(

counts = counts,

place = place,

deviceSource = deviceSource,

profileGeoEnrichment = profileGeo

),

sourceTweetResult = sourceTweetResult

)

}

}

}

}

}

val hydrateUnmentionDataToTweetData: TweetDataValueHydrator =

TweetHydration.setOnTweetData(

TweetData.Lenses.tweet.andThen(TweetLenses.unmentionData),

(td: TweetData, opts: TweetQuery.Options) =>

UnmentionDataHydrator

.Ctx(getConversationId(td.tweet), getMentions(td.tweet), TweetCtx.from(td, opts)),

hydrateUnmentionData

)

val hydrateCacheableFields: TweetDataValueHydrator =

ValueHydrator.inSequence(

scrubCachedTweet,

hydratePrimaryCacheableFields,

// Relies on mentions being hydrated in hydratePrimaryCacheableFields

hydrateUnmentionDataToTweetData,

assertNotScrubbed,

hydrateCacheableRepairs

)

// The conversation muted hydrator needs the conversation id,

// which comes from the primary cacheable fields, and the media hydrator

// needs the cacheable media entities.

val hydrateUncacheableMedia: TweetDataValueHydrator =

ValueHydrator[TweetData, TweetQuery.Options] { (td, opts) =>

val ctx = TweetCtx.from(td, opts)

val tweet = td.tweet

val mediaCtx =

MediaEntityHydrator.Uncacheable.Ctx(td.tweet.mediaKeys, ctx)

val media: Stitch[ValueState[Option[Seq[MediaEntity]]]] =

hydrateMediaUncacheable.liftOption.apply(td.tweet.media, mediaCtx)

val conversationMuted: Stitch[ValueState[Option[Boolean]]] =

hydrateConversationMuted(

tweet.conversationMuted,

ConversationMutedHydrator.Ctx(getConversationId(tweet), ctx)

)

// MediaRefs need to be hydrated at this phase because they rely on the media field

// on the Tweet, which can get unset by later hydrators.

val mediaRefs: Stitch[ValueState[Option[Seq[MediaRef]]]] =

hydrateMediaRefs(

tweet.mediaRefs,

MediaRefsHydrator.Ctx(getMedia(tweet), getMediaKeys(tweet), getUrls(tweet), ctx)

)

Stitch

.joinMap(

media,

conversationMuted,

mediaRefs

)(ValueState.join(\_, \_, \_))

.map { values =>

if (values.state.isEmpty) {

ValueState.unmodified(td)

} else {

val tweet = td.tweet

values.map {

case (media, conversationMuted, mediaRefs) =>

td.copy(

tweet = tweet.copy(

media = media,

conversationMuted = conversationMuted,

mediaRefs = mediaRefs

)

)

}

}

}

}

val hydrateHasMediaToTweetData: TweetDataValueHydrator =

TweetHydration.setOnTweetData(

TweetData.Lenses.tweet.andThen(TweetLenses.hasMedia),

(td: TweetData, opts: TweetQuery.Options) => td.tweet,

hydrateHasMedia

)

val hydrateReportedTweetVisibilityToTweetData: TweetDataValueHydrator = {

// Create a TweetDataValueHydrator that calls hydrateReportedTweetVisibility, which

// either throws a FilteredState.Unavailable or returns Unit.

ValueHydrator[TweetData, TweetQuery.Options] { (td, opts) =>

val ctx = ReportedTweetFilter.Ctx(td.tweet.perspective, TweetCtx.from(td, opts))

hydrateReportedTweetVisibility((), ctx).map { \_ =>

ValueState.unmodified(td)

}

}

}

val hydrateTweetVisibilityToTweetData: TweetDataValueHydrator =

TweetHydration.setOnTweetData(

TweetData.Lenses.suppress,

(td: TweetData, opts: TweetQuery.Options) =>

TweetVisibilityHydrator.Ctx(td.tweet, TweetCtx.from(td, opts)),

hydrateTweetVisibility

)

val hydrateEscherbirdAnnotationsToTweetAndCachedTweet: TweetDataValueHydrator =

TweetHydration.setOnTweetAndCachedTweet(

TweetLenses.escherbirdEntityAnnotations,

(td: TweetData, \_: TweetQuery.Options) => td.tweet,

hydrateEscherbirdAnnotations

)

val scrubEngagements: TweetDataValueHydrator =

TweetHydration.setOnTweetData(

TweetData.Lenses.tweetCounts,

(td: TweetData, \_: TweetQuery.Options) => ScrubEngagementHydrator.Ctx(td.suppress),

hydrateScrubEngagements

)

/\*\*

\* This is where we wire up all the separate hydrators into a single [[TweetDataValueHydrator]].

\*

\* Each hydrator here is either a [[TweetDataValueHydrator]] or a [[TweetDataEditHydrator]].

\* We use [[EditHydrator]]s for anything that needs to run in parallel ([[ValueHydrator]]s can

\* only be run in sequence).

\*/

ValueHydrator.inSequence(

// Hydrate FeatureSwitchResults first, so they can be used by other hydrators if needed

hydrateFeatureSwitchResults,

EditHydrator

.inParallel(

ValueHydrator

.inSequence(

// The result of running these hydrators is saved as `cacheableTweetResult` and

// written back to cache via `cacheChangesEffect` in `hydrateRepo`

TweetHydration.captureCacheableTweetResult(

hydrateCacheableFields

),

// Uncacheable hydrators that depend only on the cacheable fields

hydrateUncacheableMedia,

// clean-up partially hydrated entities before any of the hydrators that look at

// url and media entities run, so that they never see bad entities.

hydratePostCacheRepairs,

// These hydrators are all dependent on each other and/or the previous hydrators

hydrateDependentUncacheableFields,

// Sets `hasMedia`. Comes after PastedMediaHydrator in order to include pasted

// pics as well as other media & urls.

hydrateHasMediaToTweetData

)

.toEditHydrator,

// These hydrators do not rely on any other hydrators and so can be run in parallel

// with the above hydrators (and with each other)

hydrateIndependentUncacheableFields

)

.toValueHydrator,

// Depends on reported perspectival having been hydrated in PerspectiveHydrator

hydrateReportedTweetVisibilityToTweetData,

// Remove superfluous urls entities when there is a corresponding MediaEntity for the same url

scrubSuperfluousUrlEntities,

// The copyFromSourceTweet hydrator needs to be located after the hydrators that produce the

// fields to copy. It must be located after PartialEntityCleaner (part of postCacheRepairs),

// which removes failed MediaEntities. It also depends on takedownCountryCodes having been

// hydrated in TakedownHydrator.

copyFromSourceTweet,

// depends on AdditionalFieldsHydrator and CopyFromSourceTweet to copy safety labels

hydrateTweetVisibilityToTweetData,

// for IPI'd tweets, we want to disable tweet engagement counts from being returned

// StatusCounts for replyCount, retweetCount.

// scrubEngagements hydrator must come after tweet visibility hydrator.

// tweet visibility hydrator emits the suppressed FilteredState needed for scrubbing.

scrubEngagements,

// this hydrator runs when writing the current tweet

// Escherbird comes last in order to consume a tweet that's as close as possible

// to the tweet written to tweet\_events

hydrateEscherbirdAnnotationsToTweetAndCachedTweet

.onlyIf((td, opts) => opts.cause.writing(td.tweet.id)),

// Add an ellipsis to the end of the text for a Tweet that has a NoteTweet associated.

// This is so that the Tweet is displayed on the home timeline with an ellipsis, letting

// the User know that there's more to see.

hydrateNoteTweetSuffix,

/\*\*

\* Post-cache repair of QT text and entities to support rendering on all clients

\* Moving this to end of the pipeline to avoid/minimize chance of following hydrators

\* depending on modified tweet text or entities.

\* When we start persisting shortUrl in MH - permalink won't be empty. therefore,

\* we won't run QuotedTweetRefHydrator and just hydrate expanded and display

\* using QuotedTweetRefUrlsHydrator. We will use hydrated permalink to repair

\* QT text and entities for non-upgraded clients in this step.

\* \*/

hydrateTweetLegacyFormat

)

}

/\*\*

\* Returns a new hydrator that takes the produced result, and captures the result value

\* in the `cacheableTweetResult` field of the enclosed `TweetData`.

\*/

def captureCacheableTweetResult(h: TweetDataValueHydrator): TweetDataValueHydrator =

ValueHydrator[TweetData, TweetQuery.Options] { (td, opts) =>

h(td, opts).map { v =>

// In addition to saving off a copy of ValueState, make sure that the TweetData inside

// the ValueState has its "completedHydrations" set to the ValueState.HydrationStates's

// completedHydrations. This is used when converting to a CachedTweet.

v.map { td =>

td.copy(

cacheableTweetResult = Some(v.map(\_.addHydrated(v.state.completedHydrations)))

)

}

}

}

/\*\*

\* Takes a ValueHydrator and a Lens and returns a `TweetDataValueHydrator` that does three things:

\*

\* 1. Runs the ValueHydrator on the lensed value

\* 2. Saves the result back to the main tweet using the lens

\* 3. Saves the result back to the tweet in cacheableTweetResult using the lens

\*/

def setOnTweetAndCachedTweet[A, C](

l: Lens[Tweet, A],

mkCtx: (TweetData, TweetQuery.Options) => C,

h: ValueHydrator[A, C]

): TweetDataValueHydrator = {

// A lens that goes from TweetData -> tweet -> l

val tweetDataLens = TweetData.Lenses.tweet.andThen(l)

// A lens that goes from TweetData -> cacheableTweetResult -> tweet -> l

val cachedTweetLens =

TweetLenses

.requireSome(TweetData.Lenses.cacheableTweetResult)

.andThen(TweetResult.Lenses.tweet)

.andThen(l)

ValueHydrator[TweetData, TweetQuery.Options] { (td, opts) =>

h.run(tweetDataLens.get(td), mkCtx(td, opts)).map { r =>

if (r.state.isEmpty) {

ValueState.unmodified(td)

} else {

r.map { v => Lens.setAll(td, tweetDataLens -> v, cachedTweetLens -> v) }

}

}

}

}

/\*\*

\* Creates a `TweetDataValueHydrator` that hydrates a lensed value, overwriting

\* the existing value.

\*/

def setOnTweetData[A, C](

lens: Lens[TweetData, A],

mkCtx: (TweetData, TweetQuery.Options) => C,

h: ValueHydrator[A, C]

): TweetDataValueHydrator =

ValueHydrator[TweetData, TweetQuery.Options] { (td, opts) =>

h.run(lens.get(td), mkCtx(td, opts)).map { r =>

if (r.state.isEmpty) ValueState.unmodified(td) else r.map(lens.set(td, \_))

}

}

/\*\*

\* Produces an [[Effect]] that can be applied to a [[TweetDataValueHydrator]] to write updated

\* values back to cache.

\*/

def cacheChanges(

cache: LockingCache[TweetId, Cached[TweetData]],

stats: StatsReceiver

): Effect[ValueState[TweetData]] = {

val updatedCounter = stats.counter("updated")

val unchangedCounter = stats.counter("unchanged")

val picker = new TweetRepoCachePicker[TweetData](\_.cachedAt)

val cacheErrorCounter = stats.counter("cache\_error")

val missingCacheableResultCounter = stats.counter("missing\_cacheable\_result")

Effect[TweetResult] { result =>

// cacheErrorEncountered will never be set on `cacheableTweetResult`, so we need to

// look at the outer tweet state.

val cacheErrorEncountered = result.state.cacheErrorEncountered

result.value.cacheableTweetResult match {

case Some(ValueState(td, state)) if state.modified && !cacheErrorEncountered =>

val tweetData = td.addHydrated(state.completedHydrations)

val now = Time.now

val cached = Cached(Some(tweetData), CachedValueStatus.Found, now, Some(now))

val handler = LockingCache.PickingHandler(cached, picker)

updatedCounter.incr()

cache.lockAndSet(tweetData.tweet.id, handler)

case Some(ValueState(\_, \_)) if cacheErrorEncountered =>

cacheErrorCounter.incr()

case None =>

missingCacheableResultCounter.incr()

case \_ =>

unchangedCounter.incr()

}

}

}

/\*\*

\* Wraps a hydrator with a check such that it only executes the hydrator if `queryFilter`

\* returns true for the `TweetQuery.Option` in the `Ctx` value, and the specified

\* `HydrationType` is not already marked as having been completed in

\* `ctx.tweetData.completedHydrations`. If these conditions pass, and the underlying

\* hydrator is executed, and the result does not contain a field-level or total failure,

\* then the resulting `HydrationState` is updated to indicate that the specified

\* `HydrationType` has been completed.

\*/

def completeOnlyOnce[A, C <: TweetCtx](

queryFilter: TweetQuery.Options => Boolean = \_ => true,

hydrationType: HydrationType,

dependsOn: Set[HydrationType] = Set.empty,

hydrator: ValueHydrator[A, C]

): ValueHydrator[A, C] = {

val completedState = HydrationState.modified(hydrationType)

ValueHydrator[A, C] { (a, ctx) =>

hydrator(a, ctx).map { res =>

if (res.state.failedFields.isEmpty &&

dependsOn.forall(ctx.completedHydrations.contains)) {

// successful result!

if (!ctx.completedHydrations.contains(hydrationType)) {

res.copy(state = res.state ++ completedState)

} else {

// forced rehydration - don't add hydrationType or change modified flag

res

}

} else {

// hydration failed or not all dependencies satisfied so don't mark as complete

res

}

}

}.onlyIf { (a, ctx) =>

queryFilter(ctx.opts) &&

(!ctx.completedHydrations.contains(hydrationType))

}

}

/\*\*

\* Applies a `TweetDataValueHydrator` to a `TweetRepository.Type`-typed repository.

\* The incoming `TweetQuery.Options` are first expanded using `optionsExpander`, and the

\* resulting options passed to `repo` and `hydrator`. The resulting tweet result

\* objects are passed to `cacheChangesEffect` for possible write-back to cache. Finally,

\* the tweets are scrubbed according to the original input `TweetQuery.Options`.

\*/

def hydrateRepo(

hydrator: TweetDataValueHydrator,

cacheChangesEffect: Effect[TweetResult],

optionsExpander: TweetQueryOptionsExpander.Type

)(

repo: TweetResultRepository.Type

): TweetResultRepository.Type =

(tweetId: TweetId, originalOpts: TweetQuery.Options) => {

val expandedOpts = optionsExpander(originalOpts)

for {

repoResult <- repo(tweetId, expandedOpts)

hydratorResult <- hydrator(repoResult.value, expandedOpts)

} yield {

val hydratingRepoResult =

TweetResult(hydratorResult.value, repoResult.state ++ hydratorResult.state)

if (originalOpts.cacheControl.writeToCache) {

cacheChangesEffect(hydratingRepoResult)

}

UnrequestedFieldScrubber(originalOpts).scrub(hydratingRepoResult)

}

}

/\*\*

\* A trivial wrapper around a Stitch[\_] to provide a `joinWith`

\* method that lets us choose the precedence of exceptions.

\*

\* This wrapper is useful for the case in which it's important that

\* we specify which of the two exceptions wins (such as visibility

\* filtering).

\*

\* Since this is an [[AnyVal]], using this is no more expensive than

\* inlining the joinWith method.

\*/

// exposed for testing

case class StitchExceptionPrecedence[A](toStitch: Stitch[A]) extends AnyVal {

/\*\*

\* Concurrently evaluate two Stitch[\_] values. This is different

\* from Stitch.join in that any exception from the expression on

\* the left hand side will take precedence over an exception on

\* the right hand side. This means that an exception from the

\* right-hand side will not short-circuit evaluation, but an

\* exception on the left-hand side \*will\* short-circuit. This is

\* desirable because it allows us to return the failure with as

\* little latency as possible. (Compare to lifting \*both\* to Try,

\* which would force us to wait for both computations to complete

\* before returning, even if the one with the higher precedence is

\* already known to be an exception.)

\*/

def joinWith[B, C](rhs: Stitch[B])(f: (A, B) => C): StitchExceptionPrecedence[C] =

StitchExceptionPrecedence {

Stitch

.joinMap(toStitch, rhs.liftToTry) { (a, tryB) => tryB.map(b => f(a, b)) }

.lowerFromTry

}

}

}