1 lib/main.es6

```
import {ComponentRegistry, ExtensionRegistry, PreferencesUIStore} from 'nylas-exports';

import PreferencesComponent from './settings/preferences-component';
import MessageLoaderExtension from './message-loader/message-loader-extension';
import MessageLoaderHeader from './message-loader/message-loader-header';
import WorkerFrontend from './worker-frontend';
import ComposerLoader from './composer/composer-loader';
import KeybaseSidebar from './keybase-sidebar';

class PGPMain {
  config = {
    keybase: {
      type: 'object',
      properties: {
        username: {
          type: 'string',
          default: ''
        },
        uid: {
          type: 'string',
          default: ''
        },
        csrf_token: {
          type: 'string',
          default: ''
        },
        session_token: {
          type: 'string',
          default: ''
        }
      }
    }
  };

  _state = {};
  _tab = null;

  constructor() {
    this.activate = this.activate.bind(this);
    this.serialize = this.serialize.bind(this);
    this.deactivate = this.deactivate.bind(this);
  }

  // Activate is called when the package is loaded. If your package previously
  // saved state using `serialize` it is provided.
  //
  activate(state) {
    let _loadSettings = NylasEnv.getLoadSettings();
    let windowType = _loadSettings.windowType;

    if (windowType === 'default') {
      this._state = state;
      this._tab = new PreferencesUIStore.TabItem({
        tabId: "PGP",
        displayName: "PGP Mail",
        component: PreferencesComponent
      });

      WorkerFrontend.initialize();

      PreferencesUIStore.registerPreferencesTab(this._tab);
      ComponentRegistry.register(MessageLoaderHeader, {role: 'message:BodyHeader'});
      ComponentRegistry.register(KeybaseSidebar, {role: 'MessageListSidebar:ContactCard'});
      ExtensionRegistry.MessageView.register(MessageLoaderExtension);
    }

    if (windowType === 'default' || windowType === 'composer') {
      ComponentRegistry.register(ComposerLoader, {role: 'Composer:ActionButton'});
    }
  }

  // Serialize is called when your package is about to be unmounted.
  // You can return a state object that will be passed back to your package
```

```
  // when it is re-activated.
  serialize() {
  }

  // This **optional** method is called when the window is shutting down,
  // or when your package is being updated or disabled. If your package is
  // watching any files, holding external resources, providing commands or
  // subscribing to events, release them here.
  deactivate() {
    let _loadSettings = NylasEnv.getLoadSettings();
    let windowType = _loadSettings.windowType;

    if (windowType === 'default') {
      PreferencesUIStore.unregisterPreferencesTab(this._tab.tabId);
      ExtensionRegistry.MessageView.unregister(MessageLoaderExtension);
      ComponentRegistry.unregister(MessageLoaderHeader);
    }

    if (windowType === 'default' || windowType === 'composer') {
      ComponentRegistry.unregister(ComposerLoader);
    }
  }
}

export default new PGPMain();
```

2 lib/settings/preferences-component.es6

```
import {React} from 'nylas-exports';
import {Flexbox} from 'nylas-component-kit';

import {KeybaseActions, KeybaseStore} from '../keybase';

class PreferencesComponent extends React.Component {
  static displayName = 'PreferencesComponent';

  constructor(props) {
    super(props);

    this.render = this.render.bind(this);
    this._renderError = this._renderError.bind(this);
    this._renderUserLoginInfo = this._renderUserLoginInfo.bind(this);
    this._renderSigChain = this._renderSigChain.bind(this);
    this.onChangeUsername = this.onChangeUsername.bind(this);
    this.onChangePassphrase = this.onChangePassphrase.bind(this);
    this.loadPreviousLogin = this.loadPreviousLogin.bind(this);
    this.loginToKeybase = this.loginToKeybase.bind(this);
    this.fetchAndVerifySigChain = this.fetchAndVerifySigChain.bind(this);
    this.onKeybaseStore = this.onKeybaseStore.bind(this);

    //this.keybase = new KeybaseIntegration();
    //this.keybase.loadPreviousLogin();

    global.$pgpPref = this;

    this.defaultState = this.state = {
      error: '',
      username: '',
      passphrase: '',
      uid: '',
      csrf_token: '',
      session_token: '',
      userInfo: ''
    };

    this.state = this.loadPreviousLogin();
  }

  componentDidMount() {
    this.unsubscribe = KeybaseStore.listen(this.onKeybaseStore);
  }

  componentDidUnmount() {
```

```
    if (this.unsubscribe) {
      this.unsubscribe();
      this.unsubscribe = null;
    }
  }

  render() {
    return <div className="pgp container-pgp-mail">
      <section>
        <h2>Keybase Login</h2>
        <div><i>Sorry, tab to next field does not work</i></div>
        {this._renderError()}
        <Flexbox className="keybase-username item">
          <div className="setting-name">
            <label htmlFor="account.username">Username/Email:</label>
          </div>
          <div className="setting-value">
            <input id="account.username" type="text" placeholder="(e.g. max)" value={this.state.username} onChange={
          </div>
        </Flexbox>
        <Flexbox className="keybase-password item">
          <div className="setting-name">
            <label htmlFor="account.passphrase">Passphrase:</label>
          </div>
          <div className="setting-value">
            <input id="account.passphrase" type="password" value={this.state.passphrase} onChange={this.onChangePass
          </div>
        </Flexbox>
        {this._renderUserLoginInfo()}
        <button className="btn" onClick={this.loginToKeybase}>Login</button>
      </section>
      <section>
        <h2>SigChain Status</h2>
        <Flexbox className="keybase-sigchain item">
          {this._renderSigChain()}
        </Flexbox>
      </section>
    </div>
  }

  _renderError() {
    if (this.state.error !== '') {
      return <div className="statusBox errorBox">
        Error: {this.state.error}
      </div>
    }
  }

  _renderUserLoginInfo() {
    let { uid, session_token } = this.state;

    if (uid && session_token) {
      let body = `uid: ${uid}\nsession_token: ${session_token}`;

      // Using substitution causes <span>s to be used, causes incorrect line
      // breaks
      return <pre>{body}</pre>
    }
  }

  _renderSigChain() {
    let sigchain = KeybaseStore.getPrimarySigChain();
    if (!sigchain) {
      return "Not loaded yet.";
    }

    let keytype = (kid) => {
      if (kid.startsWith('0101')) {
        return 'PGP';
      } else if (kid.startsWith('0120')) {
        return 'NaCL';
      } else {
        return 'Unknown';
      }
    }
```

```
    return <table>
      <thead>
        <tr>
          <td>#</td>
          <td>Type</td>
          <td>Sig Key Type</td>
          <td>Fingerprint or kid</td>
        </tr>
      </thead>
      <tbody>
      {sigchain.get_links().map((link, i) => {
        return <tr key={i} className="bg-green">
          <td>{link.seqno}</td>
          <td>{link.type}</td>
          <td>{keytype(link.kid)}</td>
          <td>{link.fingerprint || link.kid}</td>
        </tr>;
      })}
      </tbody>
    </table>
  }

  onChangeUsername(e) {
    //console.log('username');

    this.setState({
      username: e.target.value
    });
  }

  onChangePassphrase(e) {
    //console.log('passphrase');

    this.setState({
      passphrase: e.target.value
    });
  }

  loadPreviousLogin() {
    let {
      username =  '',
      uid = '',
      csrf_token = '',
      session_token = ''
    } = NylasEnv.config.get('email-pgp.keybase') || {};

    return {
      error: '',
      username: username,
      passphrase: (csrf_token && session_token) ? '****' : '',
      uid: uid,
      csrf_token: csrf_token,
      session_token: session_token,
      userInfo: null
    };
  }

  loginToKeybase() {
    console.log('[PGP] Keybase Login');

    let { username, passphrase } = this.state;
    //console.log('%s %s', username, passphrase);

    this.setState(Object.assign({}, this.defaultState, {
      username
    }));

    KeybaseActions.login(username, passphrase);
  }

  fetchAndVerifySigChain() {
    let { username, uid } = this.state;
    KeybaseActions.fetchAndVerifySigChain(username, uid);
  }
```

```
  onKeybaseStore({ type, username, uid, res }) {
    if (type === 'LOGIN') {
      let { status: { name } } = res;

      if (name === 'BAD_LOGIN_PASSWORD') {
        return this.setState({ error: 'Bad Passphrase' });
      } else if (name === 'BAD_LOGIN_USER_NOT_FOUND') {
        return this.setState({ error: 'Bad Username or Email' });
      }

      this.setState(Object.assign({}, this.loadPreviousLogin(), {
        userInfo: res.me
      }));
    } else {
      console.log('listen: type=%s, username=%s, uid=%s, res=', type, username, uid, res);
      /* this.setState({
        username,
        uid
      }); */
      this.forceUpdate();
    }
  }
}

export default PreferencesComponent;
```

3 lib/email-pgp-store.es6

```
import fs from 'fs';
import path from 'path';

import NylasStore from 'nylas-store';
import {Utils, FileDownloadStore, MessageBodyProcessor} from 'nylas-exports';

import MimeParser from 'mimeparser';

import EmailPGPFileDownloadStoreWatcher from './email-pgp-file-download-store-watcher';
import EmailPGPActions from './email-pgp-actions';

import InProcessDecrypter from './decryption/in-process-decrypter';
import WorkerFrontend from './worker-frontend';
import FlowError from './flow-error.es6';
import KeyStore from './worker/kbpgp/key-store';

import smalltalk from 'smalltalk';

// THANK YOU GPGTOOLS! The `MimePart+GPGMail.m` is such a good guide to PGP
// mail decryption
class EmailPGPStore extends NylasStore {
  constructor() {
    super();

    // State-based variables for storing messages when resetting
    // MessageBodyProcessor cache
    this._cachedMessages = {};

    // Store status of message decryption for MessageLoaderHeader
    this._state = {};

    // Binding `this` to each method that uses `this`
    this._encryptMessage = this._encryptMessage.bind(this);
    this._decryptMessage = this._decryptMessage.bind(this);
    this._retryMessage = this._retryMessage.bind(this);
    this.mainDecrypt = this.mainDecrypt.bind(this);
    this._setState = this._setState.bind(this);
    this._retrievePGPAttachment = this._retrievePGPAttachment.bind(this);
    this._extractHTML = this._extractHTML.bind(this);
    this._decryptAndResetCache = this._decryptAndResetCache.bind(this);

    this.listenTo(EmailPGPActions.encryptMessage, this._encryptMessage);
    this.listenTo(EmailPGPActions.decryptMessage, this._decryptMessage);
    this.listenTo(EmailPGPActions.retryMessage, this._retryMessage);
```

```
    global.$pgpEmailPGPStore = this;
  }

  // PUBLIC

  // GPGTools (and other clients) send two attachments, where the first
  // "metadata" attachment contains the string "Version: 1" and the second
  // attachment is the encrypted message.
  //
  // Though, the "metadata" attachment's `contentType` is
  // 'application/pgp-encrypted' and the encrypted message attachment is
  // 'application/octet-stream', which is annoying to deal with.
  shouldDecryptMessage(message) {
    if (message.files.length < 1) {
      console.log(`[PGP] ${message.id}: Failed attachment test`);
      return false;
    }

    let extensionTest = (file) => {
      let ext = file.displayExtension();

      // [@"pgp", @"gpg", @"asc"]
      // https://github.com/GPGTools/GPGMail/blob/master/Source/MimePart%2BGPGMail.m#L643
      if (ext === 'pgp' ||
          ext === 'gpg' ||
          ext === 'asc') {
        return true;
      }

      return false;
    }

    if (!message.files.some(extensionTest)) {
      console.log(`[PGP] ${message.id}: Failed extension test`);
      return false;
    }

    return true;
  }

  haveCachedBody(message) {
    return !!this._cachedMessages[message.id];
  }

  getCachedBody(message) {
    return this._cachedMessages[message.id];
  }

  getState(messageId) {
    return this._state[messageId];
  }

  // PRIVATE

  // ACTION EVENTS

  _encryptMessage(message) {
  }

  _decryptMessage(message) {
    console.log('[PGP] Told to decrypt', message);
    this._decryptAndResetCache(message);
  }

  _retryMessage(message) {
    if (this._state[message.id] && this._state[message.id].decrypting) {
      console.log('[PGP] Told to retry decrypt, but in the middle of decryption');
      return false;
    } else {
      console.log('[PGP] Told to retry decrypt', message);
      delete this._state[message.id];
      this._decryptAndResetCache(message);
    }
  }
```

```
  // Utils

  _setState(messageId, state) {
    this._state[messageId] = Object.assign(this._state[messageId] || {}, state);
    this.trigger(messageId, this._state[messageId]);
  }

  // PGP MAIN

  // The main brains of this project. This retrieves the attachment and secret
  // key (someone help me find a (secure) way to store the secret key) in
  // parallel. We parse the HTML out of the content, then update the state which
  // triggers a page update
  mainDecrypt(message) {
    if (this._state[message.id]) {
      console.log(`[EmailPGPStore] Already decrypting ${message.id}`);
      return Promise.reject(`Already decrypting ${message.id}`);
    }

    console.group(`[PGP] Message: ${message.id}`);

    this._setState(message.id, {
      decrypting: true
    });

    // More decryption engines will be implemented
    const notify = (msg) => this._setState(message.id, { statusMessage: msg });
    const decrypter = this._selectDecrypter().bind(null, notify);
    const startDecrypt = process.hrtime();

    return this._getAttachmentAndKey(message, notify).spread(decrypter).then((result) => {
      const endDecrypt = process.hrtime(startDecrypt);
      console.log(`[EmailPGPStore] %cDecryption engine took ${endDecrypt[0] * 1e3 + endDecrypt[1] / 1e6}ms`, "color:

      this._setState(message.id, {
        rawMessage: result.text,
        signedBy: result.signedBy
      });

      return result;
    }).then(this._extractHTML).then((match) => {
      this._cachedMessages[message.id] = match;

      this._setState(message.id, {
        decrypting: false,
        decryptedMessage: match,
        statusMessage: null
      });

      return match;
    }).catch((error) => {
      if (error instanceof FlowError) {
        console.log(error.title);
      } else {
        console.log(error.stack);
      }
      this._setState(message.id, {
        decrypting: false,
        done: true,
        lastError: error
      });
    }).finally(() => {
      console.groupEnd();
      //delete this._state[message.id];
    });
  }

  // PGP HELPER INTERFACE

  _getKey() {
    var keyLocation = path.join(process.env.HOME, 'pgpkey');
    return fs.readFileAsync(keyLocation);
  }
```

```
  _retrievePGPAttachment(message, notify) {
    console.log("[EmailPGPStore] Attachments: %d", message.files.length);

    // Check for GPGTools-like message, even though we aren't MIME parsed yet,
    // this still applies because the `octet-stream` attachments take
    // precedence
    // https://github.com/GPGTools/GPGMail/blob/master/Source/MimePart%2BGPGMail.m#L665
    var dataPart = null;
    var dataIndex = null;
    var lastContentType = '';
    if (message.files.length >= 1) {
      let {files} = message;

      files.forEach((file, i) => {
        if ((file.contentType === 'application/pgp-signature') || // EmailPGP-style encryption
            ((file.contentType === 'application/octet-stream' && !dataPart) ||
             (lastContentType === 'application/pgp-encrypted')) || // GPGTools-style encryption
            (file.contentType === 'application/pgp-encrypted' && !dataPart)) { // Fallback
          dataPart = file;
          dataIndex = i;
          lastContentType = file.contentType;
        }
      });
    }

    if (dataPart) {
      let path = FileDownloadStore.pathForFile(dataPart);
      console.log(`[EmailPGPStore] Using file[${dataIndex}] = %O`, dataPart);

      // async fs.exists was throwing because the first argument was true,
      // found fs.access as a suitable replacement
      return fs.accessAsync(path, fs.F_OK | fs.R_OK).then(() => {
        return fs.readFileAsync(path, 'utf8').then((text) => {
          console.log("[EmailPGPStore] Read attachment from disk");
          if (!text) {
            throw new FlowError("No text in attachment", true);
          }
          return text;
        });
      }).catch((err) => {
        notify('Waiting for encrypted message attachment to download...');
        console.log('[EmailPGPStore] Attachment file inaccessable, creating pending promise');
        return EmailPGPFileDownloadStoreWatcher.promiseForPendingFile(dataPart.id);
      });
    } else {
      throw new FlowError("No valid attachment");
    }
  }

  // Retrieves the attachment and encrypted secret key for code divergence later
  _getAttachmentAndKey(message, notify) {
    const keys = {};
    const gpg = KeyStore.getKeysGPG();

    for (let key of gpg) {
      if (!key) continue;
      keys[key.key] = `[${key.type}] ${key.fpr}`;
    }

    return Promise.all([
      this._retrievePGPAttachment(message, notify),
      smalltalk.dropdown('PGP Key', 'Which PGP key should be used for decryption of this message?', keys)
    ]).spread((text, pgpkey) => {
      if (!text) {
        throw new FlowError("No text in attachment", true);
      }
      if (!pgpkey) {
        throw new FlowError("No key in pgpkey variable", true);
      }
      return [text, pgpkey];
    });
  }

  _selectDecrypter() {
    const chosen = "WORKER_PROCESS";
```

```
    decrypter = WorkerFrontend; // WORKER_PROCESS

    if (chosen === "IN_PROCESS") {
      ecrypter = new InProcessDecrypter(); // IN_PROCESS
    }

    return decrypter.decrypt;
  }

  // Uses regex to extract HTML component from a multipart message. Does not
  // contribute a significant amount of time to the decryption process.
  _extractHTML(result) {
    return new Promise((resolve, reject) => {
      let parser = new MimeParser();

      // Use MIME parsing to extract possible body
      var matched, lastContentType;
      let start = process.hrtime();

      parser.onbody = (node, chunk) => {
        if ((node.contentType.value === 'text/html') || // HTML body
            (node.contentType.value === 'text/plain' && !matched)) { // Plain text
          matched = new Buffer(chunk).toString('utf8');
          lastContentType = node.contentType.value;
        }
      }
      parser.onend = () => {
        let end = process.hrtime(start);
        console.log(`[EmailPGPStore] %cParsed MIME in ${end[0] * 1e3 + end[1] / 1e6}ms`, "color:blue");
      }

      parser.write(result.text);
      parser.end();

      // Fallback to regular expressions method
      if (!matched) {
        start = process.hrtime();
        let matches = /\n--[^\n\r]*\r?\nContent-Type: text\/html[\s\S]*?\r?\n\r?\n([\s\S]*?)\n\r?\n--/gim.exec(text)
        let end = process.hrtime(start);
        if (matches) {
          console.log(`[EmailPGPStore] %cRegex found HTML in ${end[0] * 1e3 + end[1] / 1e6}ms`, "color:blue");
          matched = matches[1];
        }
      }

      if (matched) {
        resolve(matched);
      } else {
        // REALLY FALLBACK TO RAW
        console.error('[EmailPGPStore] FALLBACK TO RAW DECRYPTED');
        let formatted = `<html><head></head><body><b>FALLBACK TO RAW:</b><br>${text}</body></html>`;
        resolve(formatted);
        //reject(new FlowError("no HTML found in decrypted"));
      }
    });
  }

  _decryptAndResetCache(message) {
    let key = MessageBodyProcessor._key(message);

    return this.mainDecrypt(message).then(() => {
      if (this._state[message.id] && !this._state[message.id].lastError) {
        // Runs resetCache every run, and there can be many messages in a thread
        // that are encrypted. TODO: need a way to track currently processing
        // messages and run resetCache once, or only reprocess one message.
        //MessageBodyProcessor.resetCache();

        var messageIndex = null;
        MessageBodyProcessor._recentlyProcessedA.some(({key: _key, body}, i) => {
          if (key === _key) {
            messageIndex = i;
            return true;
          }

          return false;
```

```
        });

        if (messageIndex !== null) {
          MessageBodyProcessor._recentlyProcessedA.splice(messageIndex, 1);
          delete MessageBodyProcessor._recentlyProcessedD[key];
        }

        let processed = MessageBodyProcessor.process(message);
        MessageBodyProcessor._subscriptions.forEach(({message: _message, callback}) => {
          if (message.id === _message.id) {
            callback(processed);
          }
        });
      }
    }).catch((err) => {
      console.log('[PGP - EmailPGPStore] %s', err);
    });
  }
}

export default new EmailPGPStore();
```

4 lib/flow-error.es6

```
// Error class to not print a stack trace
// Useful for stopping a Promise chain
class FlowError extends Error {
  name = 'FlowError';

  constructor(message, display = false) {
    super(message);
    Error.captureStackTrace(this, arguments.callee);

    this.message = message;
    this.display = display;

    this.title = this.message;
  }
}

export default FlowError;
```

5 lib/composer/mime-writer.es6

```
import mimelib from 'mimelib';
import uuid from 'uuid';

const CR = "\r";
const LF = "\n";
const CRLF = CR+LF;

// MIME Writer to create the MIME encoded emails before encryption. Normally
// the N1 Sync Engine does this itself, but for the case of secrecy of emails
// from the Sync Engine the emails are encoded in the N1 clients
//
// Based on https://github.com/isaacs/multipart-js
export default class MIMEWriter {
  constructor(boundary = `PGP-N1=_${uuid().toUpperCase()}`) {
    this._boundary = boundary;
    this._output = '';

    this.writePart = this.writePart.bind(this);
    this.end = this.end.bind(this);
    this._writeHeader = this._writeHeader.bind(this);

    this._writeHeader();
  }

  writePart(message, {
    encoding = '7bit',
    type = `text/plain; charset="UTF-8"`,
    name,
```

```
    filename
  } = {}) {
    var opener = `--${this._boundary}${CRLF}`;

    opener += `Content-Type: ${type}`;

    if (name)     opener += `; name="${name}"`;
    if (filename) opener += `; filename="${filename}"`;

    opener = mimelib.foldLine(opener);
    opener += CRLF;

    this._output += opener;
    this._output += mimelib.foldLine(`Content-Transfer-Encoding: ${encoding}`);
    this._output += CRLF;
    this._output += CRLF;
    this._output += message + CRLF;

    return this;
  }

  end() {
    this._output = this._output + `${CRLF}--${this._boundary}--${CRLF}`;

    return this._output;
  }

  _writeHeader() {
    var header = `Content-Type: multipart/signed; ${CRLF}`;
    header += `\tboundary="${this._boundary}";${CRLF}`;
    header += `\tprotocol="application/pgp-signature"${CRLF}`;
    header += CRLF + CRLF;

    this._output = header;
  }
}
```

6 lib/composer/composer-loader.es6

```
// Adds a button to encrypt the message body with a PGP user key from Keybase.
// User needs to specify which user to encrypt with. Script will download the
// key and present the user's Keybase profile to ensure verification.

import fs from 'fs';
import path from 'path';

import {Actions, DraftStore, QuotedHTMLTransformer, React, Utils} from 'nylas-exports';
import {Menu, GeneratedForm, Popover, RetinaImg} from 'nylas-component-kit';

import kbpgp from 'kbpgp';
import rimraf from 'rimraf';

import {KeybaseStore} from '../keybase';
import MIMEWriter from './mime-writer';

const NO_OP = () => {};
const SPAN_STYLES = "font-family:monospace,monospace;white-space:pre;";
const rimrafPromise = Promise.promisify(rimraf);

class ComposerLoader extends React.Component {
  static displayName = 'ComposerLoader'

  static propTypes = {
    draftClientId: React.PropTypes.string.isRequired
  }

  constructor(props) {
    super(props);

    this.render = this.render.bind(this);
    this.onChange = this.onChange.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
    this._hidePopover = this._hidePopover.bind(this);
    this._ensureConfigurationDirectoryExists = this._ensureConfigurationDirectoryExists.bind(this);
```

```
    this.temporaryAttachmentLocation = path.join(KeybaseStore._configurationDirPath, 'attachments');
    this._ensureConfigurationDirectoryExists();

    this.state = {
      username: ''
    }

    global.$pgpComposer = this;
  }

  render() {
    let button = <button className="btn btn-toolbar">
      PGP Encrypt
      <RetinaImg mode={RetinaImg.Mode.ContentPreserve}
                 name="toolbar-chevron.png" />
    </button>

    return <Popover ref="popover"
                    className="pgp pgp-menu-picker pull-right"
                    buttonComponent={button}>
      <form className="form col-12 m2">
        <label>Keybase Username:</label>
        <input className="field mb2 block" type="text" placeholder="(e.g. max)" onChange={this.onChange} />
        <button className="btn mb1 block" onClick={this.onSubmit}>Encrypt</button>
      </form>
    </Popover>
  }

  onChange(e) {
    console.log('change', e);
    this.setState({
      username: e.target.value
    });
  }

  onSubmit(e) {
    this._hidePopover();

    let {username, fingerprint} = this.state;

    console.log('submit');
    console.log(username);

    return KeybaseStore.keybaseRemote.publicKeyForUsername(username).then(armoredKey => {
      if (!armoredKey) {
        throw new Error("No public key for username " + username);
      }

      return this._importPublicKey(armoredKey).then(publicKey => {
        return [
          DraftStore.sessionForClientId(this.props.draftClientId),
          publicKey
        ];
      }).spread((session, publicKey) => {
        let draftHtml = session.draft().body;
        let text = QuotedHTMLTransformer.removeQuotedHTML(draftHtml);

        let fingerprint = kbpgp.util.format_fingerprint(publicKey.get_pgp_fingerprint());
        let bodyHeader = this._formatBodyHeader(username, fingerprint);

        return this._encryptMessage(text, publicKey).then(pgpMessage => {
          let temporaryDir = path.join(this.temporaryAttachmentLocation, this.props.draftClientId);
          let attachmentPath = path.join(temporaryDir, 'encrypted.asc');

          return fs.accessAsync(temporaryDir, fs.F_OK).then(() => {
            return rimrafPromise(temporaryDir);
          }, NO_OP).then(() => {
            return fs.mkdirAsync(temporaryDir);
          }).then(() => {
            return fs.writeFileAsync(attachmentPath, pgpMessage);
          }).then(() => {
            Actions.addAttachment({
              messageClientId: this.props.draftClientId,
              filePath: attachmentPath
```

```
            });
          });
        }).then(() => {
          let body = QuotedHTMLTransformer.appendQuotedHTML(bodyHeader, draftHtml);

          session.changes.add({ body });
          session.changes.commit();
        });
      }).catch(err => {
        console.log(err);
      });
    });
  }

  _hidePopover() {
    this.refs.popover.close();
  }

  _formatBodyHeader(username, fingerprint) {
    return `This message is encrypted for <span style="${SPAN_STYLES}">${username}</span> with key fingerprint <spar
  }

  _importPublicKey(publicKey) {
    let import_from_armored_pgp = Promise.promisify(kbpgp.KeyManager.import_from_armored_pgp);

    return import_from_armored_pgp({
      armored: publicKey
    }).then(([ keyManager, warnings ]) => {
      return keyManager;
    });
  }

  _encryptMessage(text, encrypt_for) {
    let box = Promise.promisify(kbpgp.box);

    let writer = new MIMEWriter();

    writer.writePart(text, {
      type: 'text/html; charset="UTF-8"'
    });

    let msg = writer.end();

    return box({ msg, encrypt_for }).then(([ pgpMessage, pgpMessageBuffer ]) => {
      return pgpMessage;
    });
  }

  _ensureConfigurationDirectoryExists() {
    fs.access(this.temporaryAttachmentLocation, fs.F_OK, err => {
      if (err) {
        console.log('[PGP] Temporary attachment directory missing, creating');
        fs.mkdir(this.temporaryAttachmentLocation, err => {
          if (err) {
            console.error('[PGP] Temporary attachment directory creation unsuccessful', err);
          } else {
            console.log('[PGP] Temporary attachment directory creation successful');
          }
        });
      }
    });
  }
}

export default ComposerLoader;
```

7 lib/keybase-sidebar.cjsx

```
{Utils,
 React,
 FocusedContactsStore,
 MessageStore} = require 'nylas-exports'
{RetinaImg} = require 'nylas-component-kit'
```

```
_ = require('lodash')
kbpgp = require('kbpgp')
{PKESK} = require('kbpgp/lib/openpgp/packet/sess')

EmailPGPStore = require('./email-pgp-store');
Keybase = new (require './keybase/keybase-integration')
proto = require('./worker/worker-protocol')
WorkerFrontend = require('./worker-frontend')

# TODO: Recode this in es6.
class KeybaseSidebar extends React.Component
  @displayName: 'KeybaseSidebar'

  # Providing container styles tells the app how to constrain
  # the column your component is being rendered in. The min and
  # max size of the column are chosen automatically based on
  # these values.
  @containerStyles:
    order: 1
    flexShrink: 0

  # This sidebar component listens to the FocusedContactStore,
  # which gives us access to the Contact object of the currently
  # selected person in the conversation. If you wanted to take
  # the contact and fetch your own data, you'd want to create
  # your own store, so the flow of data would be:
  #
  # FocusedContactStore => Your Store => Your Component
  #
  constructor: (@props) ->
    @state = @_getStateFromStores()
    @state.data = null

  getMessage: =>
    console.log MessageStore.items()
    return MessageStore.items()[0]

  componentDidMount: =>
    @unsubscribe = []
    @unsubscribe.push FocusedContactsStore.listen @_onChange
    @unsubscribe.push EmailPGPStore.listen @_onPGPStoreChange

  componentWillUnmount: =>
    @unsubscribe?()

  render: =>
    msg = @getMessage()

    if not EmailPGPStore.shouldDecryptMessage msg
      return <span></span>

    if @state.contact
      content = @_renderContent()
    else
      content = @_renderPlaceholder()


    <div className="contact-card-fullcontact">
      {content}
    </div>

  _proofs: =>
    _.map @state.data.by_presentation_group, (proofs, site) ->
      icon = switch
        when site is 'twitter' then 'twitter'
        when site is 'github' then 'github'
        when site is 'reddit' then 'reddit'
        when site is 'hackernews' then 'hackernews'
        else 'globe'

      for proof in proofs
        if proofs[1]?.presentation_tag is 'dns' and proof.presentation_tag is 'dns'
          break

        <div className="social-profile">
```

```
        <i className="social-icon fa fa-#{icon}" style={ marginTop: 2, minWidth: '1em' }></i>

        <div className="social-link">
          <a href=proof.proof_url>{proof.nametag}</a>
        </div>
      </div>

_cryptocoins: =>
  _.map @state.cryptoaddress, (data, type) ->
    icon = switch
      when type is 'bitcoin' then 'btc'
      else 'question-circle'

    for address in data
      <div className="social-profile">
        <i className="social-icon fa fa-#{icon}" style={ marginTop: 2, minWidth: '1em' }></i>

        <div className="social-link">
          {address.address}
        </div>
      </div>

_renderContent: =>
  # Want to include images or other static assets in your components?
  # Reference them using the nylas:// URL scheme:
  #
  # <RetinaImg
  #    url="nylas://<<package.name>>/assets/checkmark_template@2x.png"
  #    mode={RetinaImg.Mode.ContentIsMask}/>
  #
  if not @state.data
    return @_renderPlaceholder()

  proofs = @_proofs()
  coins = @_cryptocoins()

  console.log coins

  <div className="header">
    <a href="https://keybase.io/#{@state.name}" style={textDecoration: 'none'}><h1 className="name">Keybase</h1></

    <div className="social-profiles">
      {proofs}
      {coins}
    </div>
  </div>

_renderPlaceholder: =>
  <div className="header">
    <h1 className="name">Keybase</h1>

    <div className="social-profiles">
      <div className="social-profile">Loading...</div>
    </div>
  </div>

_onChange: =>
  @setState(@_getStateFromStores())

_onPGPStoreChange: (id, state) =>
  console.log '%s, %O', id, state

  if false
      promises = encrypted.map (x) =>
        Keybase.userLookup(
          key_fingerprint: [x]
          fields: ['basics', 'proofs_summary', 'cryptocurrency_addresses']
        ).then (res) =>
          console.log res
          res?.them?[0]
      console.log promises

      Promise.all(promises).then (results) =>
        console.log results
        results.forEach (res) =>
```

```
            ###
            @setState
              data: res.proofs_summary
              name: res.basics.username
              profile: res.profile
              cryptoaddress: res.cryptocurrency_addresses
            ###

  _getStateFromStores: =>
    contact: FocusedContactsStore.focusedContact()


module.exports = KeybaseSidebar
```

8 lib/email-pgp-actions.es6

```
// Expose missing Reflux
Reflux = require('nylas-exports').require('Reflux', '../node_modules/reflux');

var Actions = [
  'encryptMessage',
  'decryptMessage',
  'retryMessage'
];

Actions.forEach(key => {
  Actions[key] = Reflux.createAction(key);
});

export default Actions;
```

9 lib/worker/worker-entry.es6

```
// This is the main entry-point for the worker process. The `compile-cache` is
// used here to speed up the initialization part

var proto = require('./worker-protocol');

if (!process.send) {
  console.error('This is an IPC worker. Use as intended');
  process.exit(1);
}

[
  'PGP_COMPILE_CACHE_MODULE_PATH',
  'PGP_COMPILE_CACHE_PATH',
  'PGP_CONFIG_DIR_PATH'
].forEach(envToCheck => {
  if (!process.env[envToCheck]) {
    var err = new Error('Environment variable ' + envToCheck + ' undefined');
    console.error(err.message);
    console.error(err.stack);
    process.send({
      method: proto.ERROR_OCCURRED,
      err: err,
      errorMessage: err.message
    });
    return process.exit(1);
  }
});

process.on('uncaughtException', err => {
  console.error(err);
  process.send({
    method: proto.ERROR_OCCURRED,
    err: err,
    errorMessage: err.message,
    errorStackTrace: err.stack
  });
  process.exit(1);
});
```

```
process.on('unhandledRejection', err => {
  console.error(err);
  process.send({
    method: proto.ERROR_OCCURRED,
    err: err,
    errorMessage: err.message,
    errorStackTrace: err.stack
  });
});

global.NylasEnv = require('./nylas-env-wrapper');

var compileCacheModulePath = process.env.PGP_COMPILE_CACHE_MODULE_PATH;
var compileCachePath = process.env.PGP_COMPILE_CACHE_PATH;

require(compileCacheModulePath).setCacheDirectory(compileCachePath);
process.send({ method: proto.VERBOSE_OUT, message: 'Required the compile cache' });

// The `compile-cache` module handles initializing Babel so ES6 will work from
// this point on. We now hand off the processing to the `event-processor`

require('./event-processor');
```

10 lib/worker/kbpgp/hkp.es6

```
// HKP Public Key Fetcher

import HKPCacher from './hkp-cacher';
import {log} from '../logger';

export default class HKP {
  constructor(keyServerBaseUrl) {
    this.lookup = this.lookup.bind(this);
    this._makeFetch = this._makeFetch.bind(this);

    this._baseUrl = keyServerBaseUrl ? keyServerBaseUrl : 'https://pgp.mit.edu';
    this._fetch = (typeof window !== 'undefined' && window.fetch) ? window.fetch : this._makeFetch();
  }

  lookup(keyId) {
    var uri = this._baseUrl + `/pks/lookup?op=get&options=mr&search=0x${keyId}`;

    // Really obsure bug here. If we replace fetch(url) later, Electron throws
    // an "Illegal invocation error" unless we unwrap the variable here.
    var fetch = this._fetch;

    return HKPCacher.isCached(keyId).then((result) => {
      if (!result) {
        return fetch(uri).then((response) => response.text()).then((text) => {
          HKPCacher.cacheResult(keyId, text);
          return text;
        });
      }

      return result;
    }).then((publicKeyArmored) => {
      if (publicKeyArmored && publicKeyArmored.indexOf('-----END PGP PUBLIC KEY BLOCK-----') > -1) {
        return publicKeyArmored.trim();
      }
    });
  }

  // For testing without Electron providing fetch API
  _makeFetch() {
    let request = require('request');

    return (uri) => {
      return new Promise((resolve, reject) => {
        request(uri, (error, response, body) => {
          if (!error && response.statusCode == 200) {
            resolve({ text: () => body });
          } else {
            reject(error);
          }
```

```
        });
      });
    }
  }
}
```

11 lib/worker/kbpgp/hkp-cacher.es6

```
// HKP Remote Cacher
//
// Caches the sucessful result from any HKP request in memory and on disk. While
// the in-memory cache may not be entirely useful as KeyStore stores the decoded
// KeyManager in memory, it is still nice to have around.

import fs from 'fs';
import path from 'path';

import {log, error} from '../logger';

class HKPCacher {
  constructor() {
    this._memCache = {};
    this._cacheDirectory = path.join(NylasEnv.getConfigDirPath(), 'email-pgp', 'pubkey-cache');

    this.cacheResult = this.cacheResult.bind(this);
    this.isCached = this.isCached.bind(this);
    this._getFilePath = this._getFilePath.bind(this);
    this._ensureCacheDirectoryExists = this._ensureCacheDirectoryExists.bind(this);

    this._ensureCacheDirectoryExists();
  }

  cacheResult(keyId, result) {
    let filePath = this._getFilePath(keyId);
    this._memCache[keyId] = result;

    return new Promise((resolve, reject) => {
      fs.writeFile(filePath, result, (err) => {
        if (err) {
          reject(err);
        } else {
          resolve();
        }
      });
    });
  }

  isCached(keyId) {
    let memcached = this._memCache[keyId];
    if (memcached) {
      return Promise.resolve(memcached);
    }

    let filePath = this._getFilePath(keyId);
    return new Promise((resolve, reject) => {
      fs.readFile(filePath, 'utf8', (err, result) => {
        if (err) {
          reject(err);
        } else {
          resolve(result);
        }
      });
    }).then((result) => {
      this._memCache[keyId] = result;

      return result;
    }, (err) => {
      error('[HKPCacher] Error checking for cached pubkey, assuming false %s', err.stack);

      return false;
    });
  }

  _getFilePath(keyId) {
```

```
    return path.join(this._cacheDirectory, `pubkey_${keyId}.asc`);
  }

  _ensureCacheDirectoryExists() {
    fs.access(this._cacheDirectory, fs.F_OK, (err) => {
      if (err) {
        log('[PGP - HKPCacher] Pubkey cache directory missing, creating');
        fs.mkdir(this._cacheDirectory, (err) => {
          if (err) {
            error('[PGP - HKPCacher] Pubkey cache directory creation unsuccessful', err);
          } else {
            log('[PGP - HKPCacher] Pubkey cache directory creation successful');
          }
        });
      }
    });
  }
}

export default new HKPCacher();
```

12 lib/worker/kbpgp/key-store.es6

```
import kbpgp from 'kbpgp';
import os from 'os';
import child_process from 'child_process';

import HKP from './hkp';

let hexkid = (k) => k.toString('hex');

// Adapted from PgpKeyRing in kbpgp
class KeyStore {
  constructor() {
    this._keys = {};
    this._kms = {};

    this._hkp = new HKP();

    this.addKeyManager = this.addKeyManager.bind(this);
    this.fetchRemotePublicKey = this.fetchRemotePublicKey.bind(this);
    this.fetch = this.fetch.bind(this);
    this.findBestKey = this.findBestKey.bind(this);
    this.lookup = this.lookup.bind(this);
    this.lookupKeyManager = this.lookupKeyManager.bind(this);

    global.$pgpKeyStore = this;
  }

  addKeyManager(km) {
    let keys = km.export_pgp_keys_to_keyring();
    for (var i = 0, _len = keys.length; i < _len; i++) {
      let k = keys[i];
      let kid = hexkid(k.key_material.get_key_id());
      this._keys[kid] = k;
      this._kms[kid] = km;
    }
  }

  fetchRemotePublicKey(keyId) {
    return this._hkp.lookup(keyId).then((armored) => {
      return new Promise((resolve, reject) => {
        kbpgp.KeyManager.import_from_armored_pgp({ armored }, (err, km, warn) => {
          if (err) {
            reject(err, km, warn);
          } else {
            resolve(km, warn);
          }
        });
      });
    });
  }
```

```
  fetch(key_ids, ops, cb) {
    var ret_i, key_material, err, obj, km, _ref;
    var key_material = err = obj = km = null;

    key_ids = (() => {
      var _results = [];
      for (var _i = 0, _len = key_ids.length; _i < _len; _i++) {
        _results.push(hexkid(key_ids[_i]));
      }
      return _results;
    })();

    let check_for_key = () => {
      for (var _i = 0, _len = key_ids.length; _i < _len; _i++) {
        let id = key_ids[_i];
        let k = this._keys[id];
        if (k != null ? (_ref = k.key) != null ? _ref.can_perform(ops) : void 0 : void 0) {
          ret_i = _i;
          km = this._kms[id];
          break;
        }
      }
    }

    check_for_key();

    if (km == null) {
      let promises = key_ids.map((k) => {
        return this.fetchRemotePublicKey(k).then((kmm, warn) => {
          this.addKeyManager(kmm);
        });
      });

      Promise.all(promises).then(() => {
        check_for_key();

        if (km == null) {
          err = new Error(`key not found: ${JSON.stringify(key_ids)}`);
          cb(err, km, ret_i);
        } else {
          cb(err, km, ret_i);
        }
      }).catch((err) => {
        cb(err, km, ret_i);
      });
    } else {
      cb(err, km, ret_i);
    }
  }

  // Pick the best key to fill the flags asked for by the flags.
  // See C.openpgp.key_flags for ideas of what the flags might be.
  findBestKey({key_id, flags}, cb) {
    let kid = hexkid(key_id);
    let km = this._kms[kid];

    if (km == null) {
      err = new Error("Could not find key for fingerprint " + kid);
    } else if ((key = km.find_best_pgp_key(flags)) == null) {
      err = new Error("no matching key for flags: " + flags);
    }

    cb(err, key);
  }

  lookup(key_id) {
    return this._keys[hexkid(key_id)];
  }

  lookupKeyManager(key_id) {
    return this._kms[hexkid(key_id)];
  }

  getKeysGPG() {
    if ((os.platform() === 'linux' || os.platform() === 'darwin') && !process.env.PATH.includes('/usr/local/bin')) {
```

```
      process.env.PATH += ":/usr/local/bin";
    }

    let output = child_process.execSync('gpg --list-secret-keys --fingerprint').toString();

    var keys = [];
    var currentKey = {};

    for (let line of output.split('\n')) {
      line = line.trim();

      if (line.startsWith('sec#')) {
        continue;
      }

      if (line.startsWith('sec') || line.startsWith('ssb')) {
        let parsed = parsed = /^(sec|ssb) +(?:\w+)?([0-9]*)(?:[a-zA-Z]?)\/([a-zA-Z0-9]*) ((2[0-9]{3})-(1[0-2]|0[1-9]

        currentKey = {
          type: parsed[0] == 'sec' ? 'master' : 'subkey',
          size: parsed[1],
          key: parsed[2],
          created: parsed[3],
          expires: parsed[7]
        };
      }

      if (line.startsWith('Key fingerprint')) {
        currentKey.fpr = /^[ ]*Key fingerprint = ((([0-9a-fA-F]{4})[ ]*)+)$/.exec(line)[1];
        keys.push(currentKey);
      }
    }

    return keys;
  }
}

export default new KeyStore();
```

13 lib/worker/kbpgp/kbpgp-decrypt.es6

```
import kbpgp from 'kbpgp';
import os from 'os';
import child_process from 'child_process';

import {log} from '../logger';
import KeyStore from './key-store';

class KbpgpDecryptRoutine {
  constructor(controller, notify) {
    this._controller = controller;
    this._notify = notify;

    this._importKey = this._importKey.bind(this);
    this._checkCache = this._checkCache.bind(this);
    this._decryptKey = this._decryptKey.bind(this);
    this.run = this.run.bind(this);
  }

  _importKey(armored) {
    return new Promise((resolve, reject) => {
      kbpgp.KeyManager.import_from_armored_pgp({
        armored
      }, (err, secretKey) => {
        if (err) {
          reject(err, secretKey);
        } else {
          resolve(secretKey);
        }
      });
    });
  }
```

```
  _checkCache(secretKey) {
    let keyId = secretKey.get_pgp_key_id();
    let keyIdHex = keyId.toString('hex');
    let cachedKey = KeyStore.lookupKeyManager(keyId);
    let isLocked = this._controller.isWaitingForPassphrase(keyIdHex);
    if (cachedKey) {
      log('[InProcessDecrypter] Found cached key for %s', secretKey.get_pgp_key_id().toString('hex'));
      return Promise.resolve(cachedKey);
    } else if (isLocked) {
      return isLocked.promise;
    } else {
      return this._decryptKey(secretKey).then(secretKey => {
        KeyStore.addKeyManager(secretKey);
        return this._controller.completedPassphrasePromise(keyIdHex);
      }, err => {
        this._controller.completedPassphrasePromise(keyIdHex, {err});
        throw err;
      });
    }
  }

  _decryptKey(secretKey) {
    if (!secretKey.is_pgp_locked()) {
      return Promise.resolve(secretKey);
    }

    this._notify('Waiting for passphrase...');

    let keyId = secretKey.get_pgp_key_id().toString('hex');
    let askString = `PGP Key with fingerprint <tt>${keyId}</tt> needs to be decrypted`;
    return this._controller.requestPassphrase(keyId, askString).then(passphrase => {
      return new Promise((resolve, reject) => {
        this._notify('Unlocking secret key...');

        let startTime = process.hrtime();
        secretKey.unlock_pgp({ passphrase }, (err) => {
          if (err) {
            return reject(err);
          }

          let elapsed = process.hrtime(startTime);
          let msg = `Secret key unlocked secret key in ${elapsed[0] * 1e3 + elapsed[1] / 1e6}ms`;

          this._notify(msg);
          log('[KbpgpDecryptRoutine] %s', msg);

          resolve(secretKey);
        });
      });
    }, () => {
      // Since the first argument is undefined, the rejected promise does not
      // propagate to the `catch` receiver in `EventProcessor`. Create an Error
      // here to ensure the error is delivered to `EventProcessor`
      throw new Error('Passphrase dialog cancelled');
    });
  }

  run(armored, identifier) {
    let startTime = process.hrtime();
    let method = 'GPG_DECRYPT';

    if (method === 'GPG_DECRYPT') {
      if ((os.platform() === 'linux' || os.platform() === 'darwin') && !process.env.PATH.includes('/usr/local/bin'))
        process.env.PATH += ":/usr/local/bin";
      }

      //var key = child_process.execSync(`gpg --export-secret-keys -a ${identifier}`);
      const decrypted = child_process.spawnSync('gpg', ['--decrypt'], { input: armored });
      log(decrypted.stdout);
      log(decrypted.stderr.toString());
      const literals = [decrypted.stdout];
      const elapsed = process.hrtime(startTime);

      return Promise.resolve({literals, elapsed});
    } else {
```

```
      return this._importKey(key).then(this._checkCache).then(() => {
        return new Promise((resolve, reject) => {
          log('[KbpgpDecryptRoutine] inside the unbox closure');
          this._notify(null);

          let startDecrypt = process.hrtime();
          kbpgp.unbox({keyfetch: KeyStore, armored}, (err, literals) => {
            if (err) {
              reject(err, literals);
            } else {
              let decryptTime = process.hrtime(startDecrypt);
              let elapsed = process.hrtime(startTime);

              this._notify(`Message decrypted in ${decryptTime[0] * 1e3 + decryptTime[1] / 1e6}ms`);

              const ds = literals[0].get_data_signer();
              let km = signedBy = null;
              if (ds) {
                km = ds.get_key_manager();
              }
              if (km) {
                signedBy = km.get_pgp_fingerprint().toString('hex');
                console.log(`Signed by PGP fingerprint: ${signedBy}`);
              }

              resolve({literals, signedBy, elapsed});
            }
          });
        });
      });
    }
  }
}

// Singleton to manage each decryption session, converts stringified Buffers
// back to Buffers for kbpgp
class KbpgpDecryptController {
  constructor(eventProcessor) {
    this._eventProcessor = eventProcessor;

    this._waitingForPassphrase = {};

    this.decrypt = this.decrypt.bind(this);
    this.isWaitingForPassphrase = this.isWaitingForPassphrase.bind(this);
    this.requestPassphrase = this.requestPassphrase.bind(this);
  }

  // TODO: figure out a way to prompt the user to pick which PGP key to use to
  // decrypt or add a config page to allow them to pick per-email account.
  decrypt({armored, secretKey}, notify) {
    if (armored && armored.type === 'Buffer') {
      armored = new Buffer(armored.data);
    }

    if (secretKey && secretKey.type === 'Buffer') {
      secretKey = new Buffer(secretKey.data);
    }

    return new KbpgpDecryptRoutine(this, notify).run(armored, secretKey);
  }

  isWaitingForPassphrase(keyId) {
    return this._waitingForPassphrase[keyId];
  }

  completedPassphrasePromise(keyId, err) {
    if (!this._waitingForPassphrase[keyId]) {
      throw new Error('No pending promise for that keyId');
    }

    if (err) {
      this._waitingForPassphrase[keyId].reject(err);
      return err;
    }
```

```
    this._waitingForPassphrase[keyId].resolve();
  }

  requestPassphrase(keyId, askString) {
    if (this._waitingForPassphrase[keyId]) {
      return this._waitingForPassphrase[keyId].promise;
    }

    this._waitingForPassphrase[keyId] = {};
    this._waitingForPassphrase[keyId].promise = new Promise((resolve, reject) => {
      this._waitingForPassphrase[keyId].resolve = resolve;
      this._waitingForPassphrase[keyId].reject = reject;
    }).then(() => {
      delete this._waitingForPassphrase[keyId];
    }, err => {
      delete this._waitingForPassphrase[keyId];
      throw err;
    });

    return this._eventProcessor.requestPassphrase(askString);
  }
}

export default KbpgpDecryptController;
```

14 lib/worker/nylas-env-wrapper.js

```
// Basic NylasEnv to fetch configuration directory
function NylasEnvConstructor() {
}

NylasEnvConstructor.prototype.getConfigDirPath = function() {
  return process.env.PGP_CONFIG_DIR_PATH;
}

module.exports = new NylasEnvConstructor();
```

15 lib/worker/event-processor.es6

```
import uuid from 'uuid';

import {log} from './logger';
import proto from './worker-protocol';
import KbpgpDecryptController from './kbpgp/kbpgp-decrypt';

class EventProcessor {
  constructor() {
    this._pendingPromises = {};

    this._kbpgpDecryptController = new KbpgpDecryptController(this);

    this.requestPassphrase = this.requestPassphrase.bind(this);
    this._sendError = this._sendError.bind(this);
    this._handleDecryptMessage = this._handleDecryptMessage.bind(this);
    this._onFrontendMessage = this._onFrontendMessage.bind(this);

    process.on('message', this._onFrontendMessage);
  }

  requestPassphrase(message) {
    let id = uuid();

    return new Promise((resolve, reject) => {
      this._pendingPromises[id] = {resolve, reject};
      process.send({ method: proto.REQUEST_PASSPHRASE, id, message });
    });
  }

  _sendError(err) {
    process.send({ method: proto.ERROR_OCCURRED, err: err, errorMessage: err.message, errorStackTrace: err.stack });
  }
```

```
  _handleDecryptMessage(message) {
    let {id} = message;
    let notify = (result) => {
      process.send({ method: proto.PROMISE_NOTIFY, id, result });
    }

    this._kbpgpDecryptController.decrypt(message, notify).then(({
      literals = [],
      signedBy = '',
      elapsed
    }) => {
      process.send({ method: proto.DECRYPTION_RESULT, id, result: { text: literals[0].toString(), signedBy: signedBy
    }).catch((err) => {
      this._sendError(err);
      process.send({ method: proto.PROMISE_REJECT, id, result: err.message });
    });
  }

  _onFrontendMessage(message) {
    if (message.method === proto.DECRYPT) {
      // DECRYPT
      this._handleDecryptMessage(message);
    } else if (message.method === proto.PROMISE_RESOLVE && this._pendingPromises[message.id]) {
      // PROMISE_RESOLVE
      this._pendingPromises[message.id].resolve(message.result);
      delete this._pendingPromises[message.id];
    } else if (message.method === proto.PROMISE_REJECT && this._pendingPromises[message.id]) {
      // PROMISE_REJECT
      this._pendingPromises[message.id].reject(message.result);
      delete this._pendingPromises[message.id];
    } else if (message.method === proto.LIST_PENDING_PROMISES) {
      // LIST_PENDING_PROMISES
      log(JSON.stringify(this._pendingPromises));
      log(JSON.stringify(this._kbpgpDecryptController._waitingForPassphrase));
    }
  }
}

export default new EventProcessor();
```

16 lib/worker/worker-protocol.js

```
// A file that specifies protocol types between the master and worker processes

exports.DECRYPTION_RESULT       = 1;
exports.DECRYPT                 = 2;
exports.PROMISE_RESOLVE         = 3;
exports.PROMISE_REJECT          = 4;
exports.PROMISE_NOTIFY          = 5;
exports.REQUEST_PASSPHRASE      = 6;
exports.VERBOSE_OUT             = 7;
exports.ERROR_OCCURRED          = 8;
exports.LIST_PENDING_PROMISES   = 9;
```

17 lib/email-pgp-file-download-store-watcher.es6

```
import fs from 'fs';
import {FileDownloadStore} from 'nylas-exports';

import FlowError from './flow-error';

class EmailPGPFileDownloadStoreWatcher {
  constructor() {
    // Object of promises of attachments needed for decryption
    this._pendingPromises = {};
    this._watchingFileIds = {};

    this.promiseForPendingFile = this.promiseForPendingFile.bind(this);
    this.getFilePromise = this.getFilePromise.bind(this);
    this._onDownloadStoreChange = this._onDownloadStoreChange.bind(this);

    this._storeUnlisten = FileDownloadStore.listen(this._onDownloadStoreChange);
```

```
  }

  // PUBLIC

  promiseForPendingFile(fileId) {
    if (this._pendingPromises[fileId]) {
      return this._pendingPromises[fileId];
    }

    return this._pendingPromises[fileId] = new Promise((resolve, reject) => {
      this._watchingFileIds[fileId] = { resolve, reject };
    }).then((text) => {
      delete this._pendingPromises[fileId];
      return text;
    });
  }

  getFilePromise(fileId) {
    return this._pendingPromises[fileId];
  }

  unlisten() {
    if (this._storeUnlisten) {
      this._storeUnlisten();
    }
  }

  // PRIVATE

  _onDownloadStoreChange() {
    let changes = FileDownloadStore.downloadDataForFiles(Object.keys(this._watchingFileIds));
    console.log('Download Store Changes:', changes);
    Object.keys(changes).forEach((fileId) => {
      let file = changes[fileId];

      if (file.state === 'finished' && this._watchingFileIds[file.fileId]) {
        console.log(`Checking ${file.fileId}`);
        // TODO: Dedupe the file reading logic into separate method
        fs.accessAsync(file.targetPath, fs.F_OK | fs.R_OK).then(() => {
          console.log(`Found downloaded attachment ${fileId}`);
          return fs.readFileAsync(file.targetPath, 'utf8').then((text) => {
            if (!this._watchingFileIds[file.fileId] || !this._watchingFileIds[file.fileId].resolve) {
              console.error('watching promise undefined');
            } else {
              this._watchingFileIds[file.fileId].resolve(text);
              delete this._watchingFileIds[file.fileId];
            }
          });
        }).catch((err) => {
          this._watchingFileIds[file.fileId].reject(new FlowError('Downloaded attachment inaccessable', true));
          delete this._watchingFileIds[file.fileId];
        });
      }
    });
  }
}

export default new EmailPGPFileDownloadStoreWatcher();
```

18 lib/message-loader/message-loader-header.es6

```
// PGP Message Loader
//
// Currently for Facebook PGP-encrypted email, this will detect that Facebook
// puts the PGP encrypted document as the second attachment. It will read the
// attachment from disk asynchrnously with background tasks

import {Utils, MessageBodyProcessor, React} from 'nylas-exports';

import EmailPGPActions from '../email-pgp-actions';
import EmailPGPStore from '../email-pgp-store';
import FlowError from '../flow-error';

class MessageLoaderHeader extends React.Component {
```

```
  static displayName = 'MessageLoader'

  static propTypes = {
    message: React.PropTypes.object.isRequired
  }

  constructor(props) {
    super(props);

    // All the methods that depend on `this` instance
    this.componentDidMount = this.componentDidMount.bind(this);
    this.componentWillUnmount = this.componentWillUnmount.bind(this);
    this.shouldComponentUpdate = this.shouldComponentUpdate.bind(this);
    this.render = this.render.bind(this);
    this.retryDecryption = this.retryDecryption.bind(this);
    this._onPGPStoreChange = this._onPGPStoreChange.bind(this);

    this.state = EmailPGPStore.getState(this.props.message.id) || {};
  }

  componentDidMount() {
    this._storeUnlisten = EmailPGPStore.listen(this._onPGPStoreChange);

    window.$pgpLoaderHeader = this;
  }

  componentWillUnmount() {
    if (this._storeUnlisten) {
      this._storeUnlisten();
    }
  }

  shouldComponentUpdate(nextProps, nextState) {
    return !Utils.isEqualReact(nextProps, this.props) ||
           !Utils.isEqualReact(nextState, this.state);
  }

  render() {
    var display = true;
    var decryptingMessage, errorMessage;
    var className = "pgp-message-header";

    if (this.state.decrypting && !this.state.statusMessage) {
      displayMessage = <span>Decrypting message</span>;
    } else if (this.state.decrypting && this.state.statusMessage) {
      className += ' pgp-message-header-info';
      displayMessage = <span>{this.state.statusMessage}</span>;
    } else if (this.state.lastError &&
               ((this.state.lastError instanceof FlowError && this.state.lastError.display) ||
                !(this.state.lastError instanceof FlowError))) {
      className += ' pgp-message-header-error';
      displayMessage = <div>
        <span><b>Error: </b>{this.state.lastError.message}</span>
        <a className="pull-right option" onClick={this.retryDecryption}>Retry Decryption</a>
      </div>
    } else {
      display = false;
    }

    if (display) {
      return <div className={className}>{displayMessage}</div>;
    } else {
      return <div />;
    }
  }

  retryDecryption() {
    EmailPGPActions.retryMessage(this.props.message);
  }

  _onPGPStoreChange(messageId, state) {
    if (messageId === this.props.message.id) {
      console.log('received event', state);
      this.state = state;
```

```
      this.forceUpdate();

      // Fixed in nylas/N1@39a142ddcb80c7e1fce22dfe1e0e628272154523
      //if (state.decryptedMessage) {
      //  this.props.message.body = state.decryptedMessage;
      //}
    }
  }
}

export default MessageLoaderHeader;
```

19 lib/message-loader/message-loader-extension.es6

```
import {MessageViewExtension} from 'nylas-exports';

import EmailPGPStore from '../email-pgp-store';
import Actions from '../email-pgp-actions';

class MessageLoaderExtension extends MessageViewExtension {
  // CANNOT crash here. If we do, the whole app stops working
  // properly and the main screen is stuck with the message
  // viewer
  static formatMessageBody(message) {
    // Check for a cached message body for a decrypted message
    // If we have one we should return the cached message so the
    // proper message body is displayed
    let cached = EmailPGPStore.getCachedBody(message);
    if (cached) {
      console.log(`Have cached body for ${message.id}`);
      return message.body = cached;
    }

    // If we don't have a cached copy and the message matches the parameters for
    // decryption, then signal the `EmailPGPStore` to decrypt the message and
    // pass on the cloned message
    if (EmailPGPStore.shouldDecryptMessage(message)) {
      console.log(`[PGP] MessageLoaderExtension formatting ${message.id}`);
      Actions.decryptMessage(message);
    }
  }
}

export default MessageLoaderExtension;
```

20 lib/worker-frontend.es6

```
import child_process from 'child_process';
import readline from 'readline';
import nylasExports from 'nylas-exports';

import debugSettings from './debug-settings';
import debugInitialize from 'debug/browser';
import smalltalk from 'smalltalk';
import uuid from 'uuid';

import proto from './worker/worker-protocol';
import FlowError from './flow-error';

const debug = debugInitialize('WorkerFrontend');

class WorkerFrontend {
  constructor() {
    this._workerEntryScriptPath = path.join(__dirname, 'worker', 'worker-entry.js');
    this._pendingPromises = {};

    this.decrypt = this.decrypt.bind(this);
    this.initialize = this.initialize.bind(this);
    this._forkProcess = this._forkProcess.bind(this);
    this._requestPassphrase = this._requestPassphrase.bind(this);

    global.$pgpWorkerFrontend = this;
```

```
}

decrypt(notify, armored, secretKey) {
  let id = uuid();

  return new Promise((resolve, reject) => {
    this._pendingPromises[id] = {resolve, reject, notify};

    this._child.send({ method: proto.DECRYPT, id, armored, secretKey });
  });
}

// Called by `main.es6` when the `windowType` matches either `default` or
// `composer`
initialize() {
  this._forkProcess();
}

_forkProcess() {
  // We need to find out the path of the compile-cache module so we can
  // pass it on to the worker process, use the hijacked require to ensure it
  // is in the module cache
  let compileCache = nylasExports.require('PGP-CompileCache', 'compile-cache');
  let compileCachePath = compileCache.getCacheDirectory();

  var modulePath = '';
  Object.keys(require.cache).some((module) => {
    if (module.match(/compile-cache/)) {
      modulePath = module;
      return true;
    }
  });

  this._child = child_process.fork(this._workerEntryScriptPath, {
    env: Object.assign({}, process.env, {
      DEBUG: '*',
      PGP_COMPILE_CACHE_MODULE_PATH: modulePath,
      PGP_COMPILE_CACHE_PATH: compileCachePath,
      PGP_CONFIG_DIR_PATH: NylasEnv.getConfigDirPath()
    }),
    silent: true
  });

  const rlOut = readline.createInterface({
    input: this._child.stdout,
    terminal: false
  });
  const rlErr = readline.createInterface({
    input: this._child.stderr,
    terminal: false
  });

  rlOut.on('line', (data) => {
    debug('[child.stdout] %s', data);
  });
  rlErr.on('line', (data) => {
    debug('[child.stderr] %s', data);
  });

  this._child.on('message', (message) => {
    if (message.method === proto.ERROR_OCCURRED) {
      // ERROR_OCCURRED
      let error = new FlowError(message.errorMessage || 'unknown error, check error.childStackTrace', true);
      error.childStackTrace = message.errorStackTrace;
      console.error('[PGP - WorkerFrontend] Error from worker:', error);
      console.error(error.childStackTrace);
    } else if (message.method === proto.VERBOSE_OUT) {
      // VERBOSE_OUT
      debug('[Verbose] %s', message.message);
    } else if (message.method === proto.REQUEST_PASSPHRASE) {
      // REQUEST_PASSPHRASE
      this._requestPassphrase(message.id, message.message);
    } else if (message.method === proto.DECRYPTION_RESULT) {
      // DECRYPTION_RESULT
      if (this._pendingPromises[message.id]) {
```

```
          this._pendingPromises[message.id].resolve(message.result);
          delete this._pendingPromises[message.id];
        }
      } else if (message.method === proto.PROMISE_REJECT && this._pendingPromises[message.id]) {
        // PROMISE_REJECT
        this._pendingPromises[message.id].reject(new FlowError(message.result, true));
        delete this._pendingPromises[message.id];
      } else if (message.method === proto.PROMISE_NOTIFY && this._pendingPromises[message.id]) {
        // PROMISE_NOTIFY
        this._pendingPromises[message.id].notify(message.result);
      } else {
        debug('Unknown Message Received From Worker: %O', message);
      }
    });
  }

  _requestPassphrase(id, msg) {
    smalltalk.passphrase('PGP Passphrase', msg || '').then((passphrase) => {
      this._child.send({ method: proto.PROMISE_RESOLVE, id, result: passphrase });
      debug('Passphrase entered');
    }, () => {
      this._child.send({ method: proto.PROMISE_REJECT, id });
      debug('Passphrase cancelled');
    });
  }
}

export default new WorkerFrontend();
```

21 lib/keybase/keybase-integration.es6

```
import Keybase from 'node-keybase';
import request from 'request';

const API = 'https://keybase.io/_/api/1.0';

class KeybaseRemote {
  constructor() {
    this.keybase = new Keybase();

    this.login = Promise.promisify(this.keybase.login.bind(this.keybase));
    this.userLookup = Promise.promisify(this.keybase.user_lookup);
    this.publicKeyForUsername = Promise.promisify(this.keybase.public_key_for_username);

    this.loadPreviousLogin = this.loadPreviousLogin.bind(this);
    this.fetchAndVerifySigChain = this.fetchAndVerifySigChain.bind(this);
  }

  loadPreviousLogin() {
    let { username, uid, csrf_token, session_token } = NylasEnv.config.get('email-pgp.keybase') || {};

    if (username && uid && csrf_token && session_token) {
      console.log('[PGP] Found Keybase stored login, loading into node-keybase');
      this.keybase.usernameOrEmail = username;
      this.keybase.session = session_token;
      this.keybase.csrf_token = csrf_token;
    } else {
      console.log('[PGP] Previous Keybase login not found');
    }
  }

  sigChainForUid(uid) {
    let queryString = '?uid=' + uid;

    return new Promise((resolve, reject) => {
      request.get({
        url: API + '/sig/get.json' + queryString,
        json: true
      }, (err, res, body) => {
        if (err) {
          return reject(err);
        } else if (body.status.code === 200 && body.status.name === 'OK') {
          return reject(body.status);
```

```
        }
        return resolve(body);
      });
    });
  }

  fetchAndVerifySigChain(username, uid) {
    console.warn('Please use `KeybaseStore` with `KeybaseActions` and listen to store for events');
    KeybaseStore._fetchAndVerifySigChain(username, uid);
  }
}

export default KeybaseRemote;
```

22 lib/keybase/index.es6

```
// There is a bug where the Object.defineProperty on the first line of a ES6
// module with Babel transpiling to ES5 causes an 'Unexpected reserved word'
// error when loading in Electron

import KeybaseIntegration from './keybase-integration';

import KeybaseActions from './store/keybase-actions';
import KeybaseStore from './store/keybase-store';

export { KeybaseIntegration };
export { KeybaseActions, KeybaseStore };

let ffs = ([ hello, world, foobar ]) => {
  return;
}
```

23 /home/mbilker/.nylas/dev/packages/email-pgp/lib/keybase/store/keybase-actions.es6

```
// Expose missing Reflux
Reflux = require('nylas-exports').require('Reflux', '../node_modules/reflux');

var Actions = [
  'login',
  'fetchAndVerifySigChain'
];

Actions.forEach((key) => {
  Actions[key] = Reflux.createAction(name);
  Actions[key].sync = true;
});

export default Actions;
```

24 lib/keybase/store/keybase-store.es6

```
import fs from 'fs';
import path from 'path';

import libkb from 'libkeybase';
import NylasStore from 'nylas-store';

import KeybaseActions from './keybase-actions';
import KeybaseRemote from '../keybase-integration';

class KeybaseStore extends NylasStore {
  constructor() {
    super();

    this.keybaseRemote = new KeybaseRemote();
    this.keybaseRemote.loadPreviousLogin();

    this._cachedPrimarySigChain = null;

    this._configurationDirPath = path.join(NylasEnv.getConfigDirPath(), 'email-pgp');
```

```
    this.getPrimarySigChain = this.getPrimarySigChain.bind(this);
    this.getTrackedUsers = this.getTrackedUsers.bind(this);
    this._login = this._login.bind(this);
    this._fetchAndVerifySigChain = this._fetchAndVerifySigChain.bind(this);
    this._ensureConfigurationDirectoryExists = this._ensureConfigurationDirectoryExists.bind(this);
    this._loadSavedCredentials = this._loadSavedCredentials.bind(this);

    this.listenTo(KeybaseActions.login, this._login);
    this.listenTo(KeybaseActions.fetchAndVerifySigChain, this._fetchAndVerifySigChain);

    this._ensureConfigurationDirectoryExists();
    this._loadSavedCredentials();

    global.$pgpKeybaseStore = this;
  }

  // Helper methods

  // SigChain for the stored login
  getPrimarySigChain() {
    return this._cachedPrimarySigChain;
  }

  getPrimaryTrackedUsers() {
    return this.getTrackedUsers(this._cachedPrimarySigChain);
  }

  getTrackedUsers(sigchain) {
    if (!sigchain) {
      throw new Error('No sigchain provided');
    }

    let trackingStatus = sigchain
      .get_links()
      .filter((a) => a.type === 'track' || a.type === 'untrack')
      .reduce((origValue, value) => {
        origValue[value.payload.body[value.type].basics.username] = origValue[value.payload.body[value.type].basics.
        if (value.type === 'track') {
          origValue[value.payload.body[value.type].basics.username] += 1;
        } else if (value.type === 'untrack') {
          origValue[value.payload.body[value.type].basics.username] -= 1;
        }
        return origValue;
      }, {});

    return Object.keys(b).reduce((array, name) => {
      if (b[name] % 2 === 0) {
        array.push(name);
      }
    }, []);
  }

  // Action Trigges

  _login(username, passphrase) {
    this.keybaseRemote.login(username, passphrase).then((res) => {
      console.log(res);

      let promise = Promise.resolve(true);
      let { status: { name } } = res;

      if (name === 'BAD_LOGIN_PASSWORD') {
        console.log('[PGP] Keybase login error: Bad Passphrase');
        promise = Promise.resolve(false);
      } else if (name === 'BAD_LOGIN_USER_NOT_FOUND') {
        console.log('[PGP] Keybase login error: Bad Username or Email');
        promise = Promise.resolve(false);
      } else {
        NylasEnv.config.set('email-pgp.keybase.username', username);
        NylasEnv.config.set('email-pgp.keybase.uid', res.uid);
        NylasEnv.config.set('email-pgp.keybase.csrf_token', res.csrf_token);
        NylasEnv.config.set('email-pgp.keybase.session_token', res.session);

        promise = fs.writeFileAsync(path.join(this._configurationDirPath, 'keybase_login.json'), JSON.stringify({
          username: username,
```

```
        uid: res.uid,
        csrf_token: res.csrf_token,
        session_token: res.session
      }));

      this._loadSavedCredentials();
    }

    this.trigger({ type: 'LOGIN', username, res });

    return promise;
  });
}

_fetchAndVerifySigChain(username, uid) {
  let parseAsync = Promise.promisify(libkb.ParsedKeys.parse);
  let replayAsync = Promise.promisify(libkb.SigChain.replay);

  let cachedPublicKeys = `${username}.${uid}.public_keys.json`;
  let cachedSigchain = `${username}.${uid}.sigchain.json`;

  return this.keybaseRemote.userLookup({
    usernames: [ username ],
    fields: [ 'public_keys' ]
  }).then((result) => {
    return result.them[0].public_keys;
  }, (err) => {
    console.error('There was an error', err);
    console.log('Attempting to load from cache, if exists');

    let cachedFile = path.join(this._configurationDirPath, cachedPublicKeys);
    return fs.accessAsync(cachedFile, fs.F_OK).then(() => {
      return fs.readFileAsync(cachedFile).then(JSON.parse);
    });
  }).then((public_keys) => {
    let cachedFile = path.join(this._configurationDirPath, cachedPublicKeys);
    fs.writeFileAsync(cachedFile, JSON.stringify(public_keys)).then(() => {
      console.log('Wrote user public_keys to cache file successfully');
    }, (err) => {
      console.error('Unable to write public_keys cache file', err);
    });

    let key_bundles = public_keys.all_bundles;
    return [
      public_keys.eldest_kid,
      parseAsync({ key_bundles })
    ];
  }).spread((eldest_kid, [ parsed_keys ]) => {
    let log = (msg) => console.log(msg);

    return this.keybaseRemote.sigChainForUid(uid).then(({ sigs: sig_blobs }) => {
      let cachedFile = path.join(this._configurationDirPath, cachedSigchain);
      fs.writeFileAsync(cachedFile, JSON.stringify(sig_blobs)).then(() => {
        console.log('Wrote user sigchain to cache file successfully');
      }, (err) => {
        console.error('Unable to write sigchain cache file', err);
      });

      return sig_blobs;
    }, (err) => {
      let cachedFile = path.join(this._configurationDirPath, cachedSigchain);
      return fs.accessAsync(cachedFile, fs.F_OK).then(() => {
        return fs.readFileAsync(cachedFile).then(JSON.parse);
      });
      //throw err;
    }).then((sig_blobs) => {
      return replayAsync({
        sig_blobs, parsed_keys,
        username, uid,
        eldest_kid,
        log
      });
    }).then((res) => {
      if (username === this.username & uid === this.uid) {
        this._cachedPrimarySigChain = res;
```

```
        }

        this.trigger({ username, uid, res });
        return res;
      });
    });
  }

  // Private methods

  _ensureConfigurationDirectoryExists() {
    fs.access(this._configurationDirPath, fs.F_OK, (err) => {
      if (err) {
        console.log('[PGP] Configuration directory missing, creating');
        fs.mkdir(this._configurationDirPath, (err) => {
          if (err) {
            console.error('[PGP] Configuration directory creation unsuccessful', err);
          } else {
            console.log('[PGP] Configuration directory creation successful');
          }
        });
      }
    });
  }

  _loadSavedCredentials() {
    let { username, uid, csrf_token, session_token } = NylasEnv.config.get('email-pgp.keybase') || {};
    this.username = username;
    this.uid = uid;
    this.csrf_token = csrf_token;
    this.session_token = session_token;

    if (this.username && this.uid) {
      this._fetchAndVerifySigChain(this.username, this.uid);
    }
  }
}

export default new KeybaseStore();
```

25 spec/message-loader-header-spec.es6

```
import {Contact, File, Message, React} from 'nylas-exports';

import EmailPGPStore from '../lib/email-pgp-store';
import MessageLoaderHeader from '../lib/message-loader/message-loader-header';

let ReactTestUtils = React.addons.TestUtils;
let me = new Contact({
  name: TEST_ACCOUNT_NAME,
  email: TEST_ACCOUNT_EMAIL
});

describe("MessageLoaderHeader", function() {
  beforeEach(function() {
    this.message = new Message({
      from: [me],
      to: [me],
      cc: [],
      bcc: []
    });
    this.component = ReactTestUtils.renderIntoDocument(
      <MessageLoaderHeader message={this.message} />
    );
  });

  it("should render into the page", function() {
    expect(this.component).toBeDefined();
  });

  it("should have a displayName", function() {
    expect(MessageLoaderHeader.displayName).toBe('MessageLoader');
  });
```

```
  describe("when not decrypting", function() {
    beforeEach(function() {
      this.component.setState({decrypting: false});
    });

    it("should have no child elements", function() {
      expect(React.findDOMNode(this.component).childElementCount).toEqual(0);
    });
  });

  describe("when decrypting", function() {
    beforeEach(function() {
      this.component.setState({decrypting: true});
    });

    it("should have one single child element", function() {
      expect(React.findDOMNode(this.component).childElementCount).toEqual(1);
    });
  });

  it("should throw when text input to multipart parser is null", function() {
    let text = null;
    expect(() => this.component._extractHTML(text)).toThrow();
  });

  //it "should show a dialog box when clicked", function() {
    //spyOn(@component, '_onClick');
    //buttonNode = React.findDOMNode(this.component.refs.button);
    //ReactTestUtils.Simulate.click(buttonNode);
    //expect(@component._onClick).toHaveBeenCalled();
  //});
});
```

26 spec/main-spec.es6

```
import {ComponentRegistry, ExtensionRegistry, PreferencesUIStore} from 'nylas-exports';

import PGPMain from '../lib/main';
import MessageLoaderExtension from '../lib/message-loader/message-loader-extension';
import MessageLoaderHeader from '../lib/message-loader/message-loader-header';
import WorkerFrontend from '../lib/worker-frontend';
import ComposerLoader from '../lib/composer/composer-loader';

describe("PGPMain", () => {
  describe("::activate(state)", () => {
    it("should register the preferences tab, message header, message loader, and composer button", () => {
      spyOn(PreferencesUIStore, 'registerPreferencesTab');
      spyOn(ComponentRegistry, 'register');
      spyOn(ExtensionRegistry.MessageView, 'register');
      spyOn(WorkerFrontend, 'initialize');

      PGPMain.activate();

      expect(PGPMain._tab).not.toBeNull();
      expect(PreferencesUIStore.registerPreferencesTab).toHaveBeenCalledWith(PGPMain._tab);
      expect(ComponentRegistry.register).toHaveBeenCalledWith(MessageLoaderHeader, {role: 'message:BodyHeader'});
      expect(ExtensionRegistry.MessageView.register).toHaveBeenCalledWith(MessageLoaderExtension);
      expect(WorkerFrontend.initialize).toHaveBeenCalled();
      expect(ComponentRegistry.register).toHaveBeenCalledWith(ComposerLoader, {role: 'Composer:ActionButton'});
    });
  });

  describe("::deactivate()", () => {
    it("should unregister the preferences tab, message header, message loader, and composer button", () => {
      spyOn(PreferencesUIStore, 'unregisterPreferencesTab');
      spyOn(ComponentRegistry, 'unregister');
      spyOn(ExtensionRegistry.MessageView, 'unregister');

      PGPMain.deactivate();

      expect(PreferencesUIStore.unregisterPreferencesTab).toHaveBeenCalledWith(PGPMain._tab.tabId);
      expect(ExtensionRegistry.MessageView.unregister).toHaveBeenCalledWith(MessageLoaderExtension);
      expect(ComponentRegistry.unregister).toHaveBeenCalledWith(MessageLoaderHeader);
```

```
      expect(ComponentRegistry.unregister).toHaveBeenCalledWith(ComposerLoader);
    });
  });
});
```