```
extern crate crypto;
extern crate rand;
extern crate reqwest;
extern crate rustc_serialize as serialize;
extern crate uuid;

#[macro_use]
extern crate serde_json;

use crypto::buffer::{ReadBuffer, WriteBuffer};
use crypto::hmac;
use crypto::mac::Mac;
use crypto::symmetriccipher;
use serialize::base64;
use serialize::base64::FromBase64;
use serialize::base64::ToBase64;
use std::io;
//use reqwest::header::{Authorization,Basic};
use reqwest::header;

//make a master key.
pub fn make_key(password: &str, salt: &str) -> io::Result<[u8; 32]> {
    // 256-bit derived key
    //  hashlib.pbkdf2_hmac('sha256', password, salt, 5000, dklen=32)
    let mut dk = [0u8; 32];
    let mut mac =
        crypto::hmac::Hmac::new(crypto::sha2::Sha256::new(), &password.as_bytes().to_vec());
    let count = 5000;
    crypto::pbkdf2::pbkdf2(&mut mac, &salt.as_bytes().to_vec(), count, &mut dk);
    return Ok(dk);
}

//# base64-encode a wrapped, stretched password+salt for signup/login
pub fn hashed_password(password: &str, salt: &str) -> String {
    let key = make_key(password, salt);
    let mut derived_key = [0u8; 32];
    let mut mac = crypto::hmac::Hmac::new(crypto::sha2::Sha256::new(), &key.unwrap().to_vec());
    let count = 1;
    crypto::pbkdf2::pbkdf2(
        &mut mac,
        &password.as_bytes().to_vec(),
        count,
        &mut derived_key,
    );
    let mut result = String::from("");
    result.push_str(&derived_key.to_base64(base64::STANDARD)[..]);
    return result;
}

// encode into a bitwarden compatible cipher string.
pub fn encode_cipher_string(enctype: &u8, iv: &[u8], ct: &[u8], mac: &[u8]) -> String {
    let mut result = String::from("");
    result.push_str(&enctype.to_string());
    result.push_str(".");
    if iv.len() > 0 {
        result.push_str(&iv.to_base64(base64::STANDARD)[..]);
    }
    if ct.len() > 0 {
        result.push_str("|");
        result.push_str(&ct.to_base64(base64::STANDARD)[..]);
    }
    if mac.len() > 0 {
        result.push_str("|");
        result.push_str(&mac.to_base64(base64::STANDARD)[..]);
    }
    return result;
}
pub struct Cipherstring {
    encryption_type: u8,
    iv: Vec<u8>,
    ct: Vec<u8>,
    mac: Vec<u8>,
}

//decode a bitwarden cipher string
pub fn decode_cipher_string(cipher_string: &str) -> Cipherstring {
    let pieces: Vec<&str> = cipher_string.split("|").collect();
    let beg = pieces[0];
    let beg_pieces: Vec<&str> = beg.split(".").collect();
    let enc_type = beg_pieces[0];
    let iv = beg_pieces[1];
    let ct = pieces[1];
    let mut mac = vec![0; 0];
    if pieces.len() == 3 {
        mac = pieces[2].from_base64().unwrap();
    } else {
```

```
87          mac = [0; 0].to_vec();
88      };
89      let result = Cipherstring {
90          encryption_type: enc_type.parse::<u8>().unwrap(),
91          iv: iv.from_base64().unwrap(),
92          ct: ct.from_base64().unwrap(),
93          mac,
94      };
95      return result;
96  }
97
98  //create symmetric key (encryption_key and mac_key from secure random bytes
99  pub fn symmetric_key() -> (Vec<u8>, Vec<u8>) {
100     let mut rng = rand::thread_rng();
101     let encryption_key: Vec<u8> = rand::seq::sample_iter(&mut rng, 0..u8::max_value(), 32).unwrap();
102     let mac_key: Vec<u8> = rand::seq::sample_iter(&mut rng, 0..u8::max_value(), 32).unwrap();
103     return (encryption_key, mac_key);
104 }
105
106 // make encryption key
107 pub fn make_encrypted_key(symmetric_key: Vec<u8>, master_key: [u8; 32]) -> String {
108     let mut rng = rand::thread_rng();
109     let iv: Vec<u8> = rand::seq::sample_iter(&mut rng, 0..u8::max_value(), 16).unwrap();
110     let cipher = encrypt_aes_256_cbc(&symmetric_key, &master_key, &iv).unwrap();
111     let mac: [u8; 0] = [];
112     let ret = encode_cipher_string(&0, &iv, &cipher, &mac);
113     return ret;
114 }
115
116 //double hmac compare.
117 pub fn macs_equal(mac_key: &[u8], mac1: &[u8], mac2: &[u8]) -> bool {
118     let mut hmac1 = hmac::Hmac::new(crypto::sha2::Sha256::new(), &mac_key);
119     hmac1.input(&mac1);
120     let mut hmac2 = hmac::Hmac::new(crypto::sha2::Sha256::new(), &mac_key);
121     hmac2.input(&mac2);
122     return hmac1.result() == hmac2.result();
123 }
124
125 //decrypt encryption key
126 //pub fn decrypt_encrypted_key(cipher_string: &str, key: &[u8], mac_key: &[u8]) -> ( Vec<u8>, Vec<u8> ) {
127 //    let cipher_struct = decode_cipher_string(cipher_string);
128 //    let iv = cipher_struct.iv.drain(..).collect();
129 //    let encrypted_data = cipher_struct.ct.drain(..).collect();
130 //    assert_eq!(cipher_struct.encryption_type, 0);
131 //    let mut decryptor = crypto::aes::cbc_decryptor(
132 //        crypto::aes::KeySize::KeySize256,
133 //        key,
134 //        iv,
135 //        crypto::blockmodes::PkcsPadding,
136 //    );
137 //    let mut final_result = Vec::<u8>::new();
138 //    let mut read_buffer = crypto::buffer::RefReadBuffer::new(encrypted_data);
139 //    let mut buffer = [0; 4096];
140 //    let mut write_buffer = crypto::buffer::RefWriteBuffer::new(&mut buffer);
141 //
142 //    loop {
143 //        let result = try!(decryptor.decrypt(&mut read_buffer, &mut write_buffer, true));
144 //        final_result.extend(
145 //            write_buffer
146 //                .take_read_buffer()
147 //                .take_remaining()
148 //                .iter()
149 //                .map(|&i| i),
150 //        );
151 //        match result {
152 //            crypto::buffer::BufferResult::BufferUnderflow => break,
153 //            crypto::buffer::BufferResult::BufferOverflow => {}
154 //        }
155 //    }
156 //
157 //    let symmetric_key = final_result.drain(0..32).collect();
158 //    let mac_key = final_result.drain(32..0).collect();
159 //    return (symmetric_key, mac_key);
160 //}
161
162 // decrypt AES-256-CBC
163 pub fn decrypt_aes_256_cbc(
164     encrypted_data: &[u8],
165     key: &[u8],
166     iv: &[u8],
167 ) -> Result<Vec<u8>, symmetriccipher::SymmetricCipherError> {
168     let mut decryptor = crypto::aes::cbc_decryptor(
169         crypto::aes::KeySize::KeySize256,
170         key,
171         iv,
172         crypto::blockmodes::PkcsPadding,
```

```
    );

    let mut final_result = Vec::<u8>::new();
    let mut read_buffer = crypto::buffer::RefReadBuffer::new(encrypted_data);
    let mut buffer = [0; 4096];
    let mut write_buffer = crypto::buffer::RefWriteBuffer::new(&mut buffer);

    loop {
        let result = try!(decryptor.decrypt(&mut read_buffer, &mut write_buffer, true));
        final_result.extend(
            write_buffer
                .take_read_buffer()
                .take_remaining()
                .iter()
                .map(|&i| i),
        );
        match result {
            crypto::buffer::BufferResult::BufferUnderflow => break,
            crypto::buffer::BufferResult::BufferOverflow => {}
        }
    }

    Ok(final_result)
}
// encrypt AES-256-CBC
pub fn encrypt_aes_256_cbc(
    data: &[u8],
    key: &[u8],
    iv: &[u8],
) -> Result<Vec<u8>, symmetriccipher::SymmetricCipherError> {
    //setup
    let mut final_result = Vec::<u8>::new();
    let mut read_buffer = crypto::buffer::RefReadBuffer::new(data);
    let mut buffer = [0; 4096];
    let mut write_buffer = crypto::buffer::RefWriteBuffer::new(&mut buffer);
    let mut encryptor = crypto::aes::cbc_encryptor(
        crypto::aes::KeySize::KeySize256,
        key,
        iv,
        crypto::blockmodes::PkcsPadding,
    );
    loop {
        let result = try!(encryptor.encrypt(&mut read_buffer, &mut write_buffer, true));

        // "write_buffer.take_read_buffer().take_remaining()" means:
        // from the writable buffer, create a new readable buffer which
        // contains all data that has been written, and then access all
        // of that data as a slice.
        final_result.extend(
            write_buffer
                .take_read_buffer()
                .take_remaining()
                .iter()
                .map(|&i| i),
        );

        match result {
            crypto::buffer::BufferResult::BufferUnderflow => break,
            crypto::buffer::BufferResult::BufferOverflow => {}
        }
    }

    Ok(final_result)
}

// api/accounts/register implementation.
pub fn register(url: &reqwest::Url, email: &String, password: &String) -> String {
    let master_password_hash = hashed_password(password, email);
    let master_key = make_key(password, &email);
    let (val0, val1) = symmetric_key();
    let mut sym_key = Vec::new();
    sym_key.clone_from_slice(&val0);
    sym_key.clone_from_slice(&val1);
    let protected_key = make_encrypted_key(sym_key, master_key.unwrap());
    let signup_json = json!({
        "name": null,
        "email": email,
        "masterPasswordHash": master_password_hash,
        "masterPasswordHint":null,
        "key": protected_key,
    });
    //let url = reqwest::url::Url::parse(url);
    //let register_url = format!("{}/api/accounts/register", url);
    let client = reqwest::Client::new();
    let mut response = client
        .post(url.join("/api/accounts/register").unwrap())
```

```
        .body(signup_json.to_string())
        .send()
        .unwrap();
    // println!("register post:{:?}", res);
    if response.status().is_success() {
        println!("success!");
    } else if response.status().is_server_error() {
        println!("server error!");
    } else {
        println!("Something else happened. Status: {:?}", response.status());
    }
    let response_body = response.text();
    return response_body.unwrap();
}

//api//sync
pub fn sync(url: &reqwest::Url, access_token: &str) -> String {
    let mut header = reqwest::header::Headers::new();
    header.set(reqwest::header::Authorization(reqwest::header::Bearer {
        token: access_token.to_owned(),
    }));
    let client = reqwest::Client::new();
    let mut response = client
        .get(url.join("/api/sync").unwrap())
        .headers(header)
        .send()
        .unwrap();
    return response.text().unwrap();
}
// api/connect/token implementation.
pub fn login(url: &reqwest::Url, email: &String, password: &String) -> String {
    //let internal_key = make_key(password, &email);
    let master_password_hash = hashed_password(&password, &email);
    drop(password);
    //let id = uuid::Uuid::new_v5(&uuid::NAMESPACE_DNS, "foo").to_string();
    let id = "49a521be-c920-4cba-b6ba-f170c3993669";
    //println!("uuid:{}", id);
    let params = [
        ("grant_type", "password"),
        ("username", email),
        ("password", &master_password_hash),
        ("scope", "api offline_access"),
        ("client_id", "browser"),
        ("deviceType", "3"),
        ("deviceIdentifier", id),
        ("deviceName", "firefox"),
        ("devicePushToken", ""),
    ];
    //let url = reqwest::url::Url::parse(url);
    //let register_url = format!("{}/api/accounts/register", url);
    let client = reqwest::Client::new();
    let mut response = client
        .post(url.join("/identity/connect/token").unwrap())
        .form(&params)
        .send()
        .unwrap();
    // println!("register post:{:?}", res);
    if response.status().is_success() {
        println!("success!");
    } else if response.status().is_server_error() {
        println!("server error!");
    } else {
        println!("Something else happened. Status: {:?}", response.status());
    }
    let response_body = response.text();
    return response_body.unwrap();
}

#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn test_hashed_password() {
        let result = hashed_password("password", "nobody@example.com");
        let expected = "2cj6A0brDusMjVlVqcBW2a+kiOQDqZDCEB40NshJE7o=";
        assert_eq!(expected, result);
    }
    #[test]
    fn test_make_key() {
        let expected = b"\x95\xa9\xc3\xb6W\xfb\xa7r\x80\xbfY\xdf\xfc\x18S\x81\x9e+\xf7W\xd0\x1db\x92$\x1bN\x05\
xf5\xb8s\xe7";
        let result = make_key("password", "nobody@example.com").unwrap();
        assert_eq!(expected, &result);
    }
    #[test]
    fn test_decrypt_encrypted_key() {
```

```
344          let expected = b"";
345          //let result = decrypt_encrypted_key("0.QjjRqI96zTTB7/z3wHInzg==|WHl3wQjcPmZJ4wgADXywOhMB6RILrqPcivCJc5
    0OkivznCRaFTBXVe6MudDxYcJEu6M7RMVQfz7lLEcmcy/DFOT5veHR9YCdp4kQj3t4Tx0=",);
346          //assert_eq!(expected, &result);
347      }
348  }
349
```