*/\*\**

*\* --- DRAFT ---*

*\**

*\* A library of utilities useful for adding DeRec to wallets, password managers, or other apps that have*

*\* secrets to protect from loss.*

*\**

*\* This class defines data types and method signatures, to serve as an API for the library. It isn't currently*

*\* implemented, but is provided as a standard so that different implementations of this library in different languages*

*\* can be structured similarly. Any implementation should be written so that it communicates in accordance with the*

*\* protocols defined in DecentralizedRecoveryRFC.txt. Many of the classes here are marked abstract,*

*\* but would no longer need to be marked abstract after being completely implemented.*

*\**

*\* All of the protocols assume messages are passing between parties with the messages themselves having encryption and*

*\* authentication, so there is no need for additional security such as SSL. This is so that the parties can*

*\* communicate by any method, such as a message store-and-forward system, and not be limited to real-time connections*

*\* such as TCP/IP or UDP. Each message will serialized and deserialized according to the format defined in the RFC.*

*\**

*\* In the demo app, the initial pairing of two devices happens by one displaying a QR code and the other using its*

*\* camera. Both show the code and both use cameras, so the users can pair in either direction, but only one direction*

*\* is needed. That initial message is therefore sent by encoded it in the QR code. Other messages might be sent by*

*\* TCP/IP or by SMS text or email or by sending to a server that holds it until the other party connects to the server*

*\* to receive it.*

*\**

*\* Note that the RFC typically uses unsigned types, but Java has only signed types. So this file uses signed*

*\* byte/short/int/long for the RFCs Uint8/Uint16/Uint32/Uint64. Implementations should be careful to ensure the*

*\* behavior matches the unsigned types.*

*\**

*\* This library does not actually contain the methods for sending and receiving the messages. There can be a separate*

*\* library dealing with generating and recognizing QR codes, or sending and receiving SMS texts, or communicating with*

*\* TCP/IP sockets, or using servers to store and forward messages. Some of those will be specific to iOS or to*

*\* Android or to other operating systems. But that is beyond the scope of this library. This library is limited to*

*\* deciding what byte array message to send, and responding appropriately to each byte array message that is received.*

*\**

*\* For convenience of reading these examples, all of these classes are inside a single DeRecSdk class. When they are*

*\* actually implemented as a library, it will make more sense to make each class its own file, so it is no longer an*

*\* inner class, and remove the "static" modifier from each one.*

*\*/*

*public class DeRecSdk {*

*/\*\* the protocols defined in RFC section 4 \*/*

*enum ProtocolType {*

*PAIRING\_PROTOCOL,*

*LOCKBOX\_SHARES\_UPDATE\_PROTOCOL,*

*KEEP\_ALIVE\_PROTOCOL,*

*RECOVERY\_PROTOCOL;*

*}*

*/\*\* the current status of a given protocol state \*/*

*enum ProtocolStatus {*

*/\*\* the protocol conversation hasn't yet started \*/*

*NOT\_STARTED,*

*/\*\* the protocol is currently ongoing (more messages will be needed) \*/*

*ACTIVE,*

*/\*\* the protocol conversation finished successfully \*/*

*DONE\_SUCCESS,*

*/\*\* the protocol conversation failed (the only way to retry is to start a new conversation) \*/*

*DONE\_FAILURE;*

*}*

*/\*\* the message types defined in RFC section 5.1.1 \*/*

*enum MessageType {*

*PAIRING\_REQUEST(0x0000),*

*PAIRING\_RESPONSE(0x0001),*

*PAIRING\_ACK(0x0002),*

*LOCKBOX\_SHARE\_RETRIEVAL\_REQUEST(0x0100),*

*LOCKBOX\_SHARE\_RETREIVAL\_RESPONSE(0x0101),*

*OPERATING\_MODE\_UPDATE(0x0102),*

*KEEP\_ALIVE\_REQUEST(0x0200),*

*KEEP\_ALIVE\_RESPONSE(0x0201),*

*STORE\_LOCKBOX\_SHARE\_REQUEST(0x0300),*

*STORE\_LOCKBOX\_SHARE\_RESPONSE(0x0301),*

*LOCKBOX\_UPDATE\_REQUEST(0x0302);*

*/\*\* constructor for a message type, given its code as defined in the RFC \*/*

*MessageType(int code) {*

*this.code = code;*

*}*

*/\*\* the code number representing this message type, as defined in the RFC \*/*

*private final int code;*

*/\*\* get the code for this message type, as defined in the RFC \*/*

*public int getCode() {*

*return code;*

*}*

*}*

*/\*\* Each message inherits from this class. Each protocol works by repeatedly sending and receiving messages. Each*

*\* message is sent as a byte array that comes from serializing this message, and each received message is*

*\* instantiated by deserializing a received byte array.*

*\* \*/*

*public static abstract class Message {*

*/\*\**

*\* Serialize to a byte array, suitable for sending as a message.*

*\**

*\* For any field that is a string, it is first converted to bytes as UTF-8 with NFC normalization, then*

*\* serialized as a Uint16 length followed by the bytes.*

*\**

*\* @return create a message from this object, serialized in the format defined by the RFC*

*\*/*

*abstract byte[] serialize();*

*/\*\**

*\* @return a new object instantiated with the data from this message,*

*\* parsed according to the format defined by the RFC*

*\*/*

*abstract Message deserialize(byte[] message);*

*/\*\**

*\* @return the type of message*

*\*/*

*abstract MessageType getMessageType();*

*/\*\**

*\* @return the protocol that uses this message type*

*\*/*

*abstract ProtocolType getProtocolType();*

*/\*\**

*\* @return true if this message is the first message starting a new protocol conversation*

*\*/*

*abstract boolean firstInProtocol();*

*}*

*/\*\**

*\* This is the PairingRequest message, defined in RFC section 5.2.1*

*\*/*

*public static abstract class PairingRequestMessage extends Message {*

*private final int protocolVersion;*

*private final MessageType messageType = MessageType.PAIRING\_REQUEST;*

*private final byte counter;*

*private final byte nonce;*

*private final byte operatingMode;*

*private final String name; // in the RFC this is two fields: the name and the length of the name*

*private final short phoneNumberLength;*

*private final short ecdhPublicKeyLen;*

*private final byte[] ecdhPublicKey;*

*private final ProtocolType protocolType = ProtocolType.PAIRING\_PROTOCOL;*

*/\*\* constructor, with parameters for all fields except MessageType \*/*

*PairingRequestMessage (*

*int protocolVersion,*

*byte counter,*

*byte nonce,*

*byte operatingMode,*

*String name,*

*short phoneNumberLength,*

*short ecdhPublicKeyLen,*

*byte[] ecdhPublicKey*

*) {*

*this.protocolVersion = protocolVersion;*

*this.counter = counter;*

*this.nonce = nonce;*

*this.operatingMode = operatingMode;*

*this.name = name;*

*this.phoneNumberLength = phoneNumberLength;*

*this.ecdhPublicKeyLen = ecdhPublicKeyLen;*

*this.ecdhPublicKey = ecdhPublicKey;*

*}*

*/\* getters for all the fields \*/*

*public int getProtocolVersion() {return protocolVersion;}*

*public MessageType getMessageType() {return messageType;}*

*public byte getCounter() {return counter;}*

*public byte getNonce() {return nonce;}*

*public byte getOperatingMode() {return operatingMode;}*

*public String getName() {return name;}*

*public short getPhoneNumberLength() {return phoneNumberLength;}*

*public short getEcdhPublicKeyLen() {return ecdhPublicKeyLen;}*

*public byte[] getEcdhPublicKey() {return ecdhPublicKey;}*

*public ProtocolType getProtocolType() {return protocolType;}*

*}*

*/\* These classes should be implemented from the definitions in RFC section 5, similarly to PairingRequestMessage \*/*

*public static abstract class PairingResponseMessage extends Message {}*

*public static abstract class PairingAckMessage extends Message {}*

*public static abstract class LockboxShareRetreivalRequestMessage extends Message {}*

*public static abstract class OperatingModeUpdateMessage extends Message {}*

*public static abstract class KeepAliveRequestMessage extends Message {}*

*public static abstract class KeepAliveRequestResponseMessage extends Message {}*

*public static abstract class StoreLockboxShareRequestMessage extends Message {}*

*public static abstract class StoreLockboxShareResponseMessage extends Message {}*

*public static abstract class LockboxUpdateRequestMessage extends Message {}*

*/\*\* While a protocol conversation is in progress, all needed info is kept in this state. The state for each protocol*

*\* type should be implemented as a different class that inherits from this one.*

*\**

*\* It is possible for multiple protocols to be happening simultaneously. For example, Alice might be*

*\* simultaneously engaging in one of these protocols with Bob, while also engaging in a different protocol with*

*\* Carol. In that case, Alice will instantiate two ProtocolState objects, one for each conversation, feeding each*

*\* one the next message received from Bob or Carol, and sending to Bob or Carol the messageToSend() that it*

*\* generates.*

*\**

*\* When Alice receives a message from Bob, she should call ProtocolState.startNew(message). If this message was*

*\* from Bob trying to start a new protocol conversation, then the factory method will return a new ProtocolState*

*\* for that conversation. Otherwise, it will return null. If it returns null, then she can call the*

*\* processMessage(message) method on all of the existing ProtocolState objects in memory, until she finds one that*

*\* returns true, indicating that this message was actually an expected next message in that protocol. If all of*

*\* them return false, then this was an unexpected message, and it can be discarded.*

*\**

*\* If the message is successfully processed, either by ProtocolState.startNew(message) returning a new object, or*

*\* by processMessage(message) returning true, then Alice can call messageToSend() on that state object to retrieve*

*\* the next message to send Bob. And she can call serialize() on that message, to get the actual byte array to send.*

*\**

*\* Each protocol will implement a state that extends ProtocolState, and which will include getters and setters*

*\* specific to that protocol. For example, when Alice initiates a protocol conversation with Bob, she will first*

*\* call startProtocol(protocolType) to get a new state for a new conversation. Depending on the protocol, she then*

*\* might call several setters on the resulting state, then call nextMessageToSend() and send it.*

*\**

*\*/*

*public static class ProtocolState {*

*/\*\* the current status of this protocol (whether it is started, in progress, failed, etc) \*/*

*private ProtocolStatus protocolStatus = ProtocolStatus.NOT\_STARTED;*

*/\*\* the next message to send (or null if none). \*/*

*private Message nextMessageToSend = null;*

*/\*\* do not call constructors to instantiate this class. Only use the two static factory methods. \*/*

*private ProtocolState () {};*

*/\*\* If Alice is starting a new protocol conversation with Bob, she should call startProtocol. Then, the first*

*\* message to send Bob will either be immediately available from messageToSend(), or she may have to call some*

*\* methods on the state to provide needed data first, depending on which protocol it is.*

*\**

*\* @param protocolType which protocol to start*

*\* @return a new state for this new protocol conversation that is starting.*

*\*/*

*public static ProtocolState startProtocol(ProtocolType protocolType) {*

*return new PairingProtocolState();*

*};*

*/\*\**

*\* If Alice receives a message from Bob, it is possible that this message is a request from him to start a new*

*\* protocol conversation. Alice should call this factory method with the message. If message is starting a*

*\* new conversation, then this returns a new state for that conversation. Otherwise, it returns null.*

*\**

*\* @return the new state for the new protocol conversation started by this message, or null if none*

*\*/*

*public static ProtocolState possibleStartProtocol(Message message) {*

*if (message.firstInProtocol()) {*

*return ProtocolState.startProtocol(message.getProtocolType());*

*} else {*

*return null;*

*}*

*}*

*/\*\* The next message to send, or null if there is none. Calling this removes the message, so an immediate*

*\* second call will always return null.*

*\**

*\* @return the message to send next, or null if there is none*

*\*/*

*public Message nextMessageToSend() {*

*Message message = nextMessageToSend;*

*nextMessageToSend = null;*

*return message;*

*}*

*/\*\* the current status of this protocol \*/*

*public ProtocolStatus protocolStatus() {*

*return protocolStatus;*

*};*

*}*

*/\**

*\* each of the four protocol states should be implemented, inheriting from ProtocolState, and giving startProtocol*

*\* () a more specific return type (its own class).*

*\*/*

*public static class PairingProtocolState extends ProtocolState {*

*public PairingProtocolState startProtocol() {return new PairingProtocolState();}*

*}*

*public static class LockboxSharesUpdateProtocolState extends ProtocolState {*

*public LockboxSharesUpdateProtocolState startProtocol() {return new LockboxSharesUpdateProtocolState();}*

*}*

*public static class KeepAliveProtocolState extends ProtocolState {*

*public KeepAliveProtocolState startProtocol() {return new KeepAliveProtocolState();}*

*}*

*public static class RecoveryProtocolState extends ProtocolState {*

*public RecoveryProtocolState startProtocol() {return new RecoveryProtocolState();}*

*}*

*}*