Fusion+ test examples

Test 1: Basic Fusion+ order

1. Initialization: set up the environment including initializing chains, wallets, and contracts.
2. Fetching orders: fetch existing orders via 1inch Fusion+ Websocket API.
3. Escrow creation and deposit: create and deposit assets into source and destination chain escrows.
4. Escrow verification and withdrawals: withdraw funds from the escrow after relayer validation, ensuring that both resolver and the maker receive the correct amounts on different chains.

Test 2: Partial fill order

1. Partial fills: create an order with multiple secrets, then execute those fills using each correct secret.

- Creating the order
- Filling the order

Test 3: Order cancellation

Order cancellation or public withdrawal: cancel an order by creating a new order with a hash lock and managing the withdrawal process.

Initialization

```
import 'dotenv/config'
import {expect, jest} from '@jest/globals'
import {createServer, CreateServerReturnType} from 'prool'
import {anvil} from 'prool/instances'

import Sdk from '@1inch/cross-chain-sdk'
import {
    computeAddress,
    ContractFactory,
    JsonRpcProvider,
    MaxUint256,
    parseEther,
    parseUnits,
    randomBytes,
    Wallet as SignerWallet
} from 'ethers'
import {uint8ArrayToHex, UINT_40_MAX} from '@1inch/byte-utils'
import assert from 'node:assert'
import {ChainConfig, config} from './config'
import {Wallet} from './wallet'
import {Resolver} from './resolver'
import {EscrowFactory} from './escrow-factory'
import factoryContract from '../dist/contracts/TestEscrowFactory.sol/TestEscrowFactory.json'
import resolverContract from '../dist/contracts/Resolver.sol/Resolver.json'

const {Address} = Sdk
jest.setTimeout(1000 * 60)

// Private keys for testing (NEVER EXPOSE YOUR REAL PRIVATE KEY)
const userPk = '0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d'
const resolverPk = '0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a'

describe('Resolving example', () => {
    const srcChainId = config.chain.source.chainId
    const dstChainId = config.chain.destination.chainId

    // Define chain-related data
    type Chain = {
        node?: CreateServerReturnType | undefined
        provider: JsonRpcProvider
        escrowFactory: string
        resolver: string
    }

    // Variables for chain and wallet data
    let src: Chain
    let dst: Chain
    let srcChainUser: Wallet
    let dstChainUser: Wallet
    let srcChainResolver: Wallet
    let dstChainResolver: Wallet
    let srcFactory: EscrowFactory
    let dstFactory: EscrowFactory
    let srcResolverContract: Wallet
    let dstResolverContract: Wallet

    let srcTimestamp: bigint // Store the current timestamp

    // Utility to increase time on both chains
    async function increaseTime(t: number): Promise<void> {
        await Promise.all([src, dst].map((chain) => chain.provider.send('evm_increaseTime', [t])))
    }

    // Setup before running tests
    beforeAll(async () => {
        // Initialize chains
        ;[src, dst] = await Promise.all([initChain(config.chain.source), initChain(config.chain.destination)])

        // Initialize wallets
        srcChainUser = new Wallet(userPk, src.provider)
        dstChainUser = new Wallet(userPk, dst.provider)
        srcChainResolver = new Wallet(resolverPk, src.provider)
        dstChainResolver = new Wallet(resolverPk, dst.provider)

        // Initialize factory contracts
        srcFactory = new EscrowFactory(src.provider, src.escrowFactory)
        dstFactory = new EscrowFactory(dst.provider, dst.escrowFactory)

        // Top up user account and approve tokens
        await srcChainUser.topUpFromDonor(
            config.chain.source.tokens.USDC.address,
            config.chain.source.tokens.USDC.donor,
            parseUnits('1000', 6)
        )
        await srcChainUser.approveToken(
            config.chain.source.tokens.USDC.address,
            config.chain.source.limitOrderProtocol,
            MaxUint256
        )

        // Initialize resolver contracts and top up balances
        srcResolverContract = await Wallet.fromAddress(src.resolver, src.provider)
        dstResolverContract = await Wallet.fromAddress(dst.resolver, dst.provider)
        await dstResolverContract.topUpFromDonor(
            config.chain.destination.tokens.USDC.address,
            config.chain.destination.tokens.USDC.donor,
            parseUnits('2000', 6)
        )
        await dstChainResolver.transfer(dst.resolver, parseEther('1'))
        await dstResolverContract.unlimitedApprove(config.chain.destination.tokens.USDC.address, dst.escrow
Factory)

        // Store the current timestamp from the source chain
        srcTimestamp = BigInt((await src.provider.getBlock('latest'))!.timestamp)
    })

   // Retrieve balances for maker and resolver on both chains
async function getBalances(
    srcToken: string,
  dstToken: string
): Promise<{src: {user: bigint; resolver: bigint}; dst: {user: bigint; resolver: bigint}}> {
    return {
        src: {
            user: await srcChainUser.tokenBalance(srcToken),
            resolver: await srcResolverContract.tokenBalance(srcToken)
        },
        dst: {
            user: await dstChainUser.tokenBalance(dstToken),
            resolver: await dstResolverContract.tokenBalance(dstToken)
        }
    }
}
```

Fetching orders

```
// Test suite for filling an order
describe('Fill', () => {
    // Swapping USDC between Ethereum and BSC
    it('should swap Ethereum USDC -> Bsc USDC. Single fill only', async () => {
        const initialBalances = await getBalances(
            config.chain.source.tokens.USDC.address,
            config.chain.destination.tokens.USDC.address
        )

        // Create a cross-chain order
        const secret = uint8ArrayToHex(randomBytes(32))
        const order = Sdk.CrossChainOrder.new(
            new Address(src.escrowFactory),
            {
                salt: Sdk.randBigInt(1000n),
                maker: new Address(await srcChainUser.getAddress()),
                makingAmount: parseUnits('100', 6),
                takingAmount: parseUnits('99', 6),
                makerAsset: new Address(config.chain.source.tokens.USDC.address),
                takerAsset: new Address(config.chain.destination.tokens.USDC.address)
            },
            {
                hashLock: Sdk.HashLock.forSingleFill(secret),
                timeLocks: Sdk.TimeLocks.new({
                    srcWithdrawal: 10n,
                    srcPublicWithdrawal: 120n,
                    srcCancellation: 121n,
                    srcPublicCancellation: 122n,
                    dstWithdrawal: 10n,
                    dstPublicWithdrawal: 100n,
                    dstCancellation: 101n
                }),
                srcChainId,
                dstChainId,
                srcSafetyDeposit: parseEther('0.001'),
                dstSafetyDeposit: parseEther('0.001')
            },
            {
                auction: new Sdk.AuctionDetails({
                    initialRateBump: 0,
                    points: [],
                    duration: 120n,
                    startTime: srcTimestamp
                }),
                whitelist: [
                    {
                        address: new Address(src.resolver),
                        allowFrom: 0n
                    }
                ],
                resolvingStartTime: 0n
            },
            {
                nonce: Sdk.randBigInt(UINT_40_MAX),
                    allowPartialFills: false,
                    allowMultipleFills: false
                }
            )

// Sign the order and calculate its hash
const signature = await srcChainUser.signOrder(srcChainId, order);
const orderHash = order.getOrderHash(srcChainId);
```

Escrow creation and deposit

```
// Initialize resolver contract and log the order fill process
const resolverContract = new Resolver(src.resolver, dst.resolver);
console.log(`[${srcChainId}]`, `Filling order ${orderHash}`);

// Fill the order on the source chain
const fillAmount = order.makingAmount;
const { txHash: orderFillHash, blockHash: srcDeployBlock } =
  await srcChainResolver.send(
    resolverContract.deploySrc(
      srcChainId,
      order,
      signature,
      Sdk.TakerTraits.default()
        .setExtension(order.extension)
        .setAmountMode(Sdk.AmountMode.maker)
        .setAmountThreshold(order.takingAmount),
      fillAmount,
    ),
  );
console.log(
  `[${srcChainId}]`,
  `Order ${orderHash} filled for ${fillAmount} in tx ${orderFillHash}`,
);

// Handle event and deposit on the destination chain
const srcEscrowEvent = await srcFactory.getSrcDeployEvent(srcDeployBlock);
const dstImmutables = srcEscrowEvent[0]
  .withComplement(srcEscrowEvent[1])
  .withTaker(new Address(resolverContract.dstAddress));

console.log(
  `[${dstChainId}]`,
  `Depositing ${dstImmutables.amount} for order ${orderHash}`,
);
const { txHash: dstDepositHash, blockTimestamp: dstDeployedAt } =
  await dstChainResolver.send(resolverContract.deployDst(dstImmutables));
console.log(
  `[${dstChainId}]`,
  `Created dst deposit for order ${orderHash} in tx ${dstDepositHash}`,
);

// Retrieve and calculate escrow contract addresses
```

```
const ESCROW_SRC_IMPLEMENTATION = await srcFactory.getSourceImpl();
const ESCROW_DST_IMPLEMENTATION = await dstFactory.getDestinationImpl();

const srcEscrowAddress = new Sdk.EscrowFactory(
  new Address(src.escrowFactory),
).getSrcEscrowAddress(srcEscrowEvent[0], ESCROW_SRC_IMPLEMENTATION);

const dstEscrowAddress = new Sdk.EscrowFactory(
  new Address(dst.escrowFactory),
).getDstEscrowAddress(
  srcEscrowEvent[0],
  srcEscrowEvent[1],
  dstDeployedAt,
  new Address(resolverContract.dstAddress),
  ESCROW_DST_IMPLEMENTATION,
);
```

Escrow verification and withdrawals

```
await increaseTime(11);
// User shares secret after validation of dst escrow deployment
console.log(
  `[${dstChainId}]`,
  `Withdrawing funds for user from ${dstEscrowAddress}`,
);
await dstChainResolver.send(
  resolverContract.withdraw(
    "dst",
    dstEscrowAddress,
    secret,
    dstImmutables.withDeployedAt(dstDeployedAt),
  ),
);

console.log(
  `[${srcChainId}]`,
  `Withdrawing funds for resolver from ${srcEscrowAddress}`,
);
const { txHash: resolverWithdrawHash } = await srcChainResolver.send(
  resolverContract.withdraw("src", srcEscrowAddress, secret, srcEscrowEvent[0]),
);
console.log(
  `[${srcChainId}]`,
  `Withdrew funds for resolver from ${srcEscrowAddress} to ${src.resolver} in tx ${resolverWithdrawHash}`,
);

const resultBalances = await getBalances(
  config.chain.source.tokens.USDC.address,
  config.chain.destination.tokens.USDC.address,
);

// User transferred funds to resolver on source chain
expect(initialBalances.src.user - resultBalances.src.user).toBe(
  order.makingAmount,
);
expect(resultBalances.src.resolver - initialBalances.src.resolver).toBe(
  order.makingAmount,
);
// Resolver transferred funds to user on destination chain
expect(resultBalances.dst.user - initialBalances.dst.user).toBe(
  order.takingAmount,
);
expect(initialBalances.dst.resolver - resultBalances.dst.resolver).toBe(
  order.takingAmount,
);
```

Partial fills

Creating the order

```
  it("should swap Ethereum USDC -> Bsc USDC. Multiple fills. Fill 100%", async () => {
    const initialBalances = await getBalances(
      config.chain.source.tokens.USDC.address,
      config.chain.destination.tokens.USDC.address
    );

    // User creates order with 11 secrets (10 parts)
    // note: use a cryptographically secure random number for real-life scenarios
    const secrets = Array.from({ length: 11 }).map(() => uint8ArrayToHex(randomBytes(32)));
    const secretHashes = secrets.map((s) => Sdk.HashLock.hashSecret(s));
    const leaves = Sdk.HashLock.getMerkleLeaves(secrets);
    const order = Sdk.CrossChainOrder.new(
      new Address(src.escrowFactory),
      {
        salt: Sdk.randBigInt(1000n),
        maker: new Address(await srcChainUser.getAddress()),
        makingAmount: parseUnits("100", 6),
        takingAmount: parseUnits("99", 6),
        makerAsset: new Address(config.chain.source.tokens.USDC.address),
        takerAsset: new Address(config.chain.destination.tokens.USDC.address)
      },
      {
        hashLock: Sdk.HashLock.forMultipleFills(leaves),
        timeLocks: Sdk.TimeLocks.new({
          srcWithdrawal: 10n, // 10s finality lock for test
          srcPublicWithdrawal: 120n, // 2m for private withdrawal
          srcCancellation: 121n, // 1sec public withdrawal
          srcPublicCancellation: 122n, // 1sec private cancellation
          dstWithdrawal: 10n, // 10s finality lock for test
          dstPublicWithdrawal: 100n, // 100sec private withdrawal
          dstCancellation: 101n // 1sec public withdrawal
        }),
        srcChainId,
        dstChainId,
        srcSafetyDeposit: parseEther("0.001"),
            dstSafetyDeposit: parseEther("0.001")
        },
        {
            auction: new Sdk.AuctionDetails({
                initialRateBump: 0,
                points: [],
                duration: 120n,
                startTime: srcTimestamp
            }),
            whitelist: [
                {
                    address: new Address(src.resolver),
                    allowFrom: 0n
                }
            ],
            resolvingStartTime: 0n
        },
        {
            nonce: Sdk.randBigInt(UINT_40_MAX),
            allowPartialFills: true,
            allowMultipleFills: true
        }
    );

    const signature = await srcChainUser.signOrder(srcChainId, order);
    const orderHash = order.getOrderHash(srcChainId);
```

Filling the order

```
// Resolver fills order
const resolverContract = new Resolver(src.resolver, dst.resolver);

console.log(`[${srcChainId}]`, `Filling order ${orderHash}`);

const fillAmount = order.makingAmount;
const idx = secrets.length - 1; // last index to fulfill

const { txHash: orderFillHash, blockHash: srcDeployBlock } = await srcChainResolver.send(
    resolverContract.deploySrc(
        srcChainId,
        order,
        signature,
        Sdk.TakerTraits.default()
            .setExtension(order.extension)
            .setInteraction(
                new Sdk.EscrowFactory(new Address(src.escrowFactory)).getMultipleFillInteraction(
                    Sdk.HashLock.getProof(leaves, idx),
                    idx,
                    secretHashes[idx]
                )
            )
            .setAmountMode(Sdk.AmountMode.maker)
            .setAmountThreshold(order.takingAmount),
        fillAmount,
        Sdk.HashLock.fromString(secretHashes[idx])
    )
);

console.log(`[${srcChainId}]`, `Order ${orderHash} filled for ${fillAmount} in tx ${orderFillHash}`);

const srcEscrowEvent = await srcFactory.getSrcDeployEvent(srcDeployBlock);

const dstImmutables = srcEscrowEvent[0]
  .withComplement(srcEscrowEvent[1])
  .withTaker(new Address(resolverContract.dstAddress));

console.log(`[${dstChainId}]`, `Depositing ${dstImmutables.amount} for order ${orderHash}`);
const { txHash: dstDepositHash, blockTimestamp: dstDeployedAt } = await dstChainResolver.send(
  resolverContract.deployDst(dstImmutables)
);
console.log(`[${dstChainId}]`, `Created dst deposit for order ${orderHash} in tx ${dstDepositHash}`);

const secret = secrets[idx];

const ESCROW_SRC_IMPLEMENTATION = await srcFactory.getSourceImpl();
const ESCROW_DST_IMPLEMENTATION = await dstFactory.getDestinationImpl();

const srcEscrowAddress = new Sdk.EscrowFactory(new Address(src.escrowFactory)).getSrcEscrowAddress(
  srcEscrowEvent[0],
  ESCROW_SRC_IMPLEMENTATION
);

const dstEscrowAddress = new Sdk.EscrowFactory(new Address(dst.escrowFactory)).getDstEscrowAddress(
  srcEscrowEvent[0],
  srcEscrowEvent[1],
  dstDeployedAt,
  new Address(resolverContract.dstAddress),
  ESCROW_DST_IMPLEMENTATION
);

await increaseTime(11); // finality lock passed
// User shares secret after validation of dst escrow deployment
console.log(`[${dstChainId}]`, `Withdrawing funds for user from ${dstEscrowAddress}`);
await dstChainResolver.send(
  resolverContract.withdraw("dst", dstEscrowAddress, secret, dstImmutables.withDeployedAt(dstDeployedAt))
);

console.log(`[${srcChainId}]`, `Withdrawing funds for resolver from ${srcEscrowAddress}`);
const { txHash: resolverWithdrawHash } = await srcChainResolver.send(
  resolverContract.withdraw("src", srcEscrowAddress, secret, srcEscrowEvent[0])
);
console.log(
  `[${srcChainId}]`,
  `Withdrew funds for resolver from ${srcEscrowAddress} to ${src.resolver} in tx ${resolverWithdrawHash}`
);

const resultBalances = await getBalances(
  config.chain.source.tokens.USDC.address,
  config.chain.destination.tokens.USDC.address
);

// User transferred funds to resolver on the source chain
```

```
    expect(initialBalances.src.user - resultBalances.src.user).toBe(order.makingAmount);
    expect(resultBalances.src.resolver - initialBalances.src.resolver).toBe(order.makingAmount);
    // Resolver transferred funds to user on destination chain
    expect(resultBalances.dst.user - initialBalances.dst.user).toBe(order.takingAmount);
    expect(initialBalances.dst.resolver - resultBalances.dst.resolver).toBe(order.takingAmount);
    });
```

Order cancellation or public withdrawal

```
    // ---- Order cancellation ----
    describe('Cancel', () => {
        it('should cancel swap Ethereum USDC -> Bsc USDC', async () => {
            const initialBalances = await getBalances(
                config.chain.source.tokens.USDC.address,
                config.chain.destination.tokens.USDC.address
            )

            // User creates order
            // note: use a cryptographically secure random number for real-life scenarios
            const hashLock = Sdk.HashLock.forSingleFill(uint8ArrayToHex(randomBytes(32)))
            const order = Sdk.CrossChainOrder.new(
                new Address(src.escrowFactory),
                {
                    salt: Sdk.randBigInt(1000n),
                    maker: new Address(await srcChainUser.getAddress()),
                    makingAmount: parseUnits('100', 6),
                    takingAmount: parseUnits('99', 6),
                    makerAsset: new Address(config.chain.source.tokens.USDC.address),
                    takerAsset: new Address(config.chain.destination.tokens.USDC.address)
                },
                {
                    hashLock,
                    timeLocks: Sdk.TimeLocks.new({
                        srcWithdrawal: 0n, // no finality lock for test
                        srcPublicWithdrawal:
```