

BREEZ SDK - NODELESS  
**PLAN ⚡ EDITION**

The background features a dark blue-to-black gradient with three thin, light blue curved lines that sweep across the frame from left to right, creating a sense of motion.

# Prerequisites and Course Information

To be able to get the most out of this lecture, you'll need some prior knowledge on the following topics:

Bitcoin and Layer-2 Networks

**[DEMO ONLY]** Git & Open-Source Platforms

**[DEMO ONLY]** Web development (React+TS)

While we don't necessarily **require** Bitcoin knowledge to use our tools, it's always best to know what you're doing!

## COURSE DETAILS

**Lecturer:** Antonio (**hydra-yse**)

**Duration:** 3-4 hours

**Required Software:** Git, NodeJS (+npm), IDE

## NOTICE ON THE RELIABILITY OF THE MATERIAL

These tools are in continuous development and evolution, so do **NOT** expect the material in this course to be consistent with future versions of the software discussed.

A good approximation for the life-expectancy of this course is **6-12 months**.

# Lecture Overview

By the end of the lecture, you will have extensive knowledge on how to integrate Bitcoin payments into any application using the Breez SDK  
Nodeless - *Liquid Implementation*.

To that extent, the lecture is divided into two parts:

## PART I

### About Us

Introduction to our company and goals

Overview of our SDKs

Preliminaries on the Nodeless implementation

### Step-By-Step

A detailed dive into the SDK interface

How to effortlessly send & receive funds across  
different networks

## PART II

### Hands-On Experience

You will learn how to add the referenced features to a  
demo web-application using Typescript, from start to  
finish

### Comparing Solutions

We will share and compare your different submissions,  
seeing which one would fare best in a real-world  
scenario

# About Us

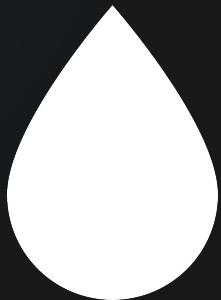


**Our Motto** - *"Make Value Move Like Information"*

**Our Objective** - Enabling access to Bitcoin across the globe, in true peer-to-peer fashion

**Our Way** - Providing the necessary tools for a frictionless, complete developer experience

# Our Implementations



**LIQUID**

Active since April 2024

Uses the *Liquid* network as L2 to interoperate with Lightning

Leverages **Boltz** to seamlessly trade funds between the multiple networks

Feature-rich, wide platform support, actively maintained



**GREENLIGHT**

DEPRECATED

Developed between late-2022 and 2025

Uses cloud-based Lightning nodes to send and receive funds. Keys are stored on-device.

Nodes are hosted on Blockstream's infrastructure, communication occurs via gRPC



**SPARK**

NEW

Active since June 2025

Uses the *Spark* network as L2 to interoperate with Lightning

Feature-rich, latest iteration of our software development kits

# Supported Languages



...and more!

# Before Getting Started

The Nodeless implementation, as well as our other SDKs, follow a specific **paradigm**: connect, prepare, execute. This allows developers with the flexibility to provide different forms of user experiences, catered to different needs and environments.



## Connect

Connects to the SDK, specifying which **configuration options** to use. Background tasks are started, and then the SDK is ready to go.

Available methods are:

- `connect()`
- `disconnect()`

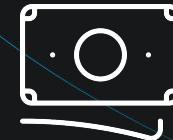


## Prepare

Provides information such as **fees** and **payer/receiver amounts**, which are fundamental for payment confirmation and UX.

Available methods are:

- `prepare_send_payment()`
- `prepare_receive_payment()`
- `prepare_lnurl_pay()`
- `prepare_refund()`



## Execute

Either generates a payment address to **receive funds** to, or uses the provided destination to **send funds**.

Available methods are:

- `send_payment()`
- `receive_payment()`
- `lnurl_pay()`
- `refund()`



DOCUMENTATION

# Step 0 - Installing

Depending on your language of choice, you probably have a specific package manager. The `breez-sdk-liquid` package should be available in most of them.

## Rust

```
cargo add --git https://github.com/breez/breez-sdk-liquid
```

## Golang

```
go get github.com/breez/breez-sdk-liquid-go
```

## JS/TS (WebAssembly)

```
npm install @breeztech/react-native-breez-sdk-liquid
```

## C#

```
dotnet add package Breez.Sdk.Liquid
```

## Python

```
pip install breez_sdk_liquid
```

## Dart/Flutter

```
flutter pub add https://github.com/breez/breez-sdk-liquid-flutter
```

...and more. Check out the docs (icon above) for non-standard installations.

# Step 1 - Connecting



connecting is the first step when starting to use the SDK. We need to configure the SDK to our needs and bootstrap the necessary services for it to work.

javascript

```
import init, { defaultConfig, connect } from '@breeztech/breez-sdk-liquid'

// Call init when using the SDK in a web environment before calling any
// other SDK methods.
// This is not needed when using the SDK in a Node.js/Deno environment.
await init()

const mnemonic = '<mnemonics words>'
// Create the default config, providing your Breez API key
const config = defaultConfig('mainnet', '<your-Breez-API-key>')
const sdk = await connect({ mnemonic, config })
```

## Usage Notes

**mnemonic** - A 12 or 24-words long string which will map to the seed of your wallet

**config** - The configuration object for the SDK. You can use **defaultConfig** for some sane defaults, depending on your network.

The Breez API key is **mandatory** on mainnet. Get it [here](#).

# Step 2 - Wallet Information

The very first thing you'd want to do is to get the wallet information. Luckily that's pretty straightforward, `getInfo` will do the job:

javascript

```
const info = await sdk.getInfo()  
const walletInfo = info.walletInfo  
const balanceSat = walletInfo.balanceSat  
const pendingSendSat = walletInfo.pendingSendSat  
const pendingReceiveSat = walletInfo.pendingReceiveSat
```

javascript

```
export interface WalletInfo {  
  balanceSat: number;  
  pendingSendSat: number;  
  pendingReceiveSat: number;  
  fingerprint: string;  
  pubkey: string;  
  assetBalances: AssetBalance[];  
}
```

## Usage Notes

The method returns two structures, `walletInfo` and `blockchainInfo`

The pending balance is updated on every new Liquid block is detected

# Step 3 - Parsing Inputs

Now that we have the basics down, let's start receiving some payments! Recalling our [previous slide](#), we first prepare:

## LIGHTNING (BOLT11)

```
javascript

// Fetch the Receive lightning limits
const currentLimits = await sdk.fetchLightningLimits()

// Set the amount you wish the payer to send via lightning,
// which should be within the above limits
const optionalAmount: ReceiveAmount = {
  type: 'bitcoin',
  payerAmountSat: 5_000
}

const prepareResponse = await sdk.prepareReceivePayment({
  paymentMethod: 'bolt11Invoice',
  amount: optionalAmount
})
```

## BITCOIN

```
javascript

// Fetch the Receive onchain limits
const currentLimits = await sdk.fetchOnchainLimits()

// Set the onchain amount you wish the payer to send, which
// should be within the above limits
const optionalAmount: ReceiveAmount = {
  type: 'bitcoin',
  payerAmountSat: 5_000
}

const prepareResponse = await sdk.prepareReceivePayment({
  paymentMethod: 'bitcoinAddress',
  amount: optionalAmount
})
```

When receiving via Lightning or Bitcoin, there is a minimum and maximum amount.

Notice the **differences**:

- The limit fetching changes between `fetchLightningLimits` and `fetchOnchainLimits`
- The `paymentMethod` changes between '`bolt11Invoice`' and '`bitcoinAddress`'

# Step 4 - Receiving Payments - Prepare

Now that we have the basics down, let's start receiving some payments! Recalling our [previous slide](#), we first prepare:

## LIGHTNING (BOLT11)

```
javascript

// Fetch the Receive lightning limits
const currentLimits = await sdk.fetchLightningLimits()

// Set the amount you wish the payer to send via lightning,
// which should be within the above limits
const optionalAmount: ReceiveAmount = {
  type: 'bitcoin',
  payerAmountSat: 5_000
}

const prepareResponse = await sdk.prepareReceivePayment({
  paymentMethod: 'bolt11Invoice',
  amount: optionalAmount
})
```

## BITCOIN

```
javascript

// Fetch the Receive onchain limits
const currentLimits = await sdk.fetchOnchainLimits()

// Set the onchain amount you wish the payer to send, which
// should be within the above limits
const optionalAmount: ReceiveAmount = {
  type: 'bitcoin',
  payerAmountSat: 5_000
}

const prepareResponse = await sdk.prepareReceivePayment({
  paymentMethod: 'bitcoinAddress',
  amount: optionalAmount
})
```

When receiving via Lightning or Bitcoin, there is a minimum and maximum amount.

Notice the **differences**:

- The limit fetching changes between `fetchLightningLimits` and `fetchOnchainLimits`
- The `paymentMethod` changes between '`bolt11Invoice`' and '`bitcoinAddress`'

# Step 4 - Receiving Payments - Execute

## LIQUID

```
javascript
// Note: Not setting the amount will generate a plain Liquid
address
const optionalAmount: ReceiveAmount = {
  type: 'bitcoin',
  payerAmountSat: 5_000
}

const prepareResponse = await sdk.prepareReceivePayment({
  paymentMethod: 'liquidAddress',
  amount: optionalAmount
})
```

Finally, once we've prepared our request and validated the fees, we proceed with the **execution**:

```
javascript
const optionalDescription = '<description>'
const res = await sdk.receivePayment({
  prepareResponse,
  description: optionalDescription
})
const destination = res.destination
```

## LIGHTNING (BOLT12)

```
javascript
const prepareResponse = await sdk.prepareReceivePayment({
  paymentMethod: 'bolt12Offer'
})
```

## Usage Notes

The destination returned is a string representing our output. Depending on what `paymentMethod` we chose, we may get a Lightning invoice/offer, or a Liquid/Bitcoin address (BIP21 is supported as well)

# Step 5 - Sending Payments - Liquid/Lightning



## Preparing

```
javascript
const optionalAmount: PayAmount = {
  type: 'bitcoin',
  receiverAmountSat: 5_000
}
const prepareResponse = await sdk.prepareSendPayment({
  destination: '<offer/invoice/address>',
  amount: optionalAmount
})
// If the fees are acceptable, continue to create the
// Send Payment
const sendFeesSat = prepareResponse.feesSat
```

## Executing

```
javascript
const optionalPayerNote = '<payer note>'
const sendResponse = await sdk.sendPayment({
  prepareResponse,
  payerNote: optionalPayerNote
})
const payment = sendResponse.payment
```

## Usage Notes

The `destination` passed can be either a BOLT11 invoice, BOLT12 offer or Liquid address/BIP21.

If the destination already includes an amount, the `amount` field becomes redundant and optional. If it doesn't match the embedded destination amount, an error is thrown.

# Step 5 - Sending Payments - Bitcoin



## Preparing

javascript

```
const optionalSatPerVbyte = 21
const prepareResponse = await
sdk.preparePayOnchain({
  amount: {
    type: 'bitcoin',
    receiverAmountSat: 5_000
  },
  feeRateSatPerVbyte: optionalSatPerVbyte
})

// Check if the fees are acceptable before
proceeding
const claimFeesSat = prepareResponse.claimFeesSat
const totalFeesSat = prepareResponse.totalFeesSat
```

## Executing

javascript

```
const destinationAddress = 'bc1..'

const payOnchainRes = await
sdk.payOnchain({
  address: destinationAddress,
  prepareResponse
})
```

# Step 6 - Listing Payments



After having executed our payments, we can then list and filter through them using the `listPayments` method:

javascript

```
const payments = await sdk.listPayments({  
  filters: ['send'],  
  fromTimestamp: 1696880000,  
  toTimestamp: 1696959200,  
  offset: 0,  
  limit: 50  
})
```

javascript

```
export interface ListPaymentsRequest {  
  filters?: PaymentType[];  
  states?: PaymentState[];  
  fromTimestamp?: number;  
  toTimestamp?: number;  
  offset?: number;  
  limit?: number;  
  details?: ListPaymentDetails;  
  sortAscending?: boolean;  
}
```

# Step 7 - Event Listeners



User interfaces require a way to **react** to events. Event **listeners** are made specifically for that purpose: they allow us to get nudged by the SDK whenever something important has been completed/needs attention:

```
javascript

class JsEventListener {
  onEvent = (event: SdkEvent) => {
    console.log(`Received event: ${JSON.stringify(event)}`)
  }
}

const eventListener = new JsEventListener()
const listenerId = await
sdk.addEventListener(eventListener)

// Later on ...

await sdk.removeEventListener(listenerId)
```

```
javascript

export type SdkEvent = {
  type: "paymentFailed"; details: Payment
} | {
  type: "paymentPending"; details: Payment
} | {
  type: "paymentRefundable"; details: Payment
} | {
  type: "paymentRefunded"; details: Payment
} | {
  type: "paymentRefundPending"; details: Payment
} | {
  type: "paymentSucceeded"; details: Payment
} | {
  type: "paymentWaitingConfirmation"; details: Payment
} | {
  type: "paymentWaitingFeeAcceptance"; details: Payment
} | {
  type: "synced"
} | {
  type: "syncFailed"; error: string
} | {
  type: "dataSynced"; didPullNewRecords: boolean
};
```

# Hands-On Experience

Has anyone of you played *Space Invaders* as a kid? Well, it's time to reinvoke old times and rewrite the version of our good old game with some Lightning functionality **embedded** in it!

To get started, simply scan the following QR code:

