Stellar

GUESTBOOK DEMO DAPP WALKTHROUGH

CONTENTS

Section 01

INTRODUCTION

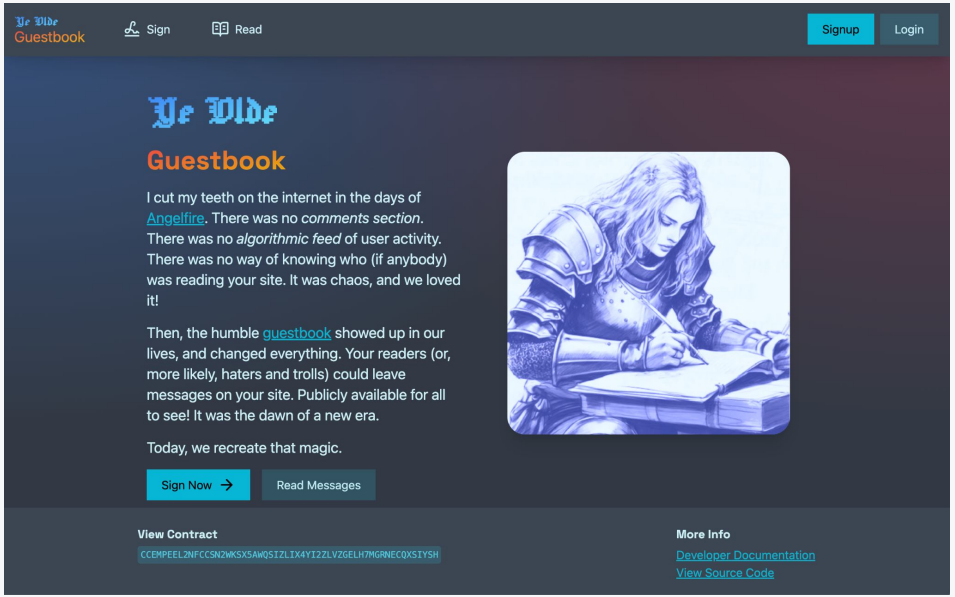
INTRODUCTION

Guestbook Dapp

In this walkthrough, we will take a closer look at the guestbook dapp described in the documentation.

The guestbook dapp is a traditional online guestbook, but it's built with cutting-edge Stellar features, including passkey auth.

We will dive into the architecture of the dapp, the solutions used, and the necessary 3rd party services.

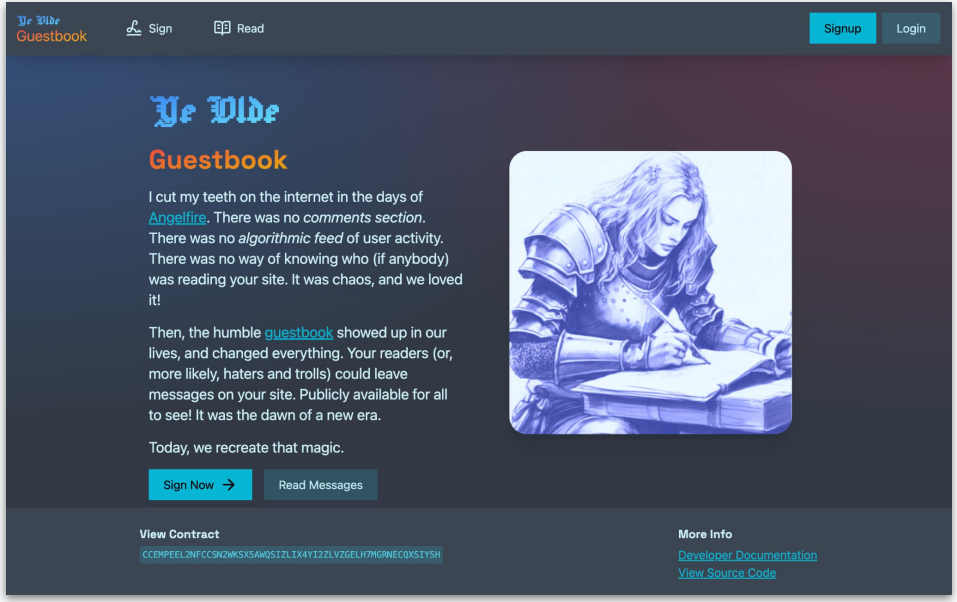


Live Demo: https://ye-olde-guestbook.vercel.app
Documentation: https://developers.stellar.org/docs/build/apps/guestbook

INTRODUCTION

Guestbook Dapp

Let’s try the live demo and sign the guestbook!



Live Demo: https://ye-olde-guestbook.vercel.app
Documentation: https://developers.stellar.org/docs/build/apps/guestbook

Section 02

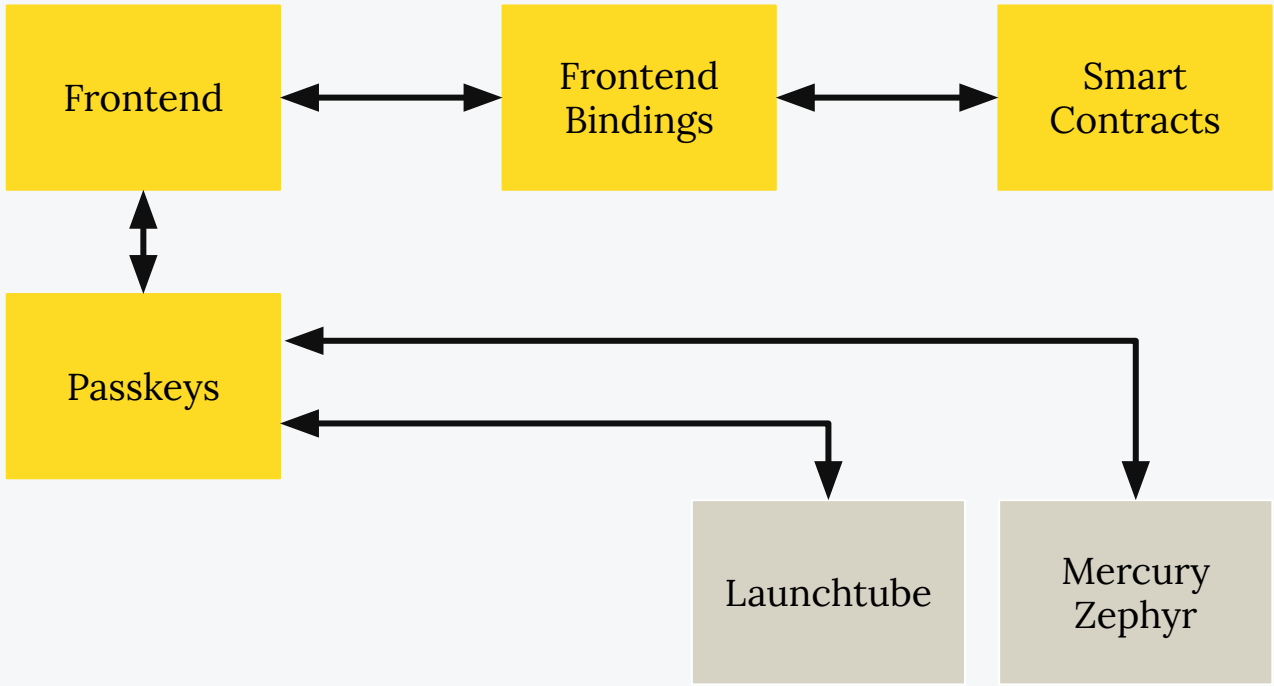
HIGH-LEVEL ARCHITECTURE

HIGH-LEVEL ARCHITECTURE

The guestbook dapp stores messages on-chain by using smart contract functions.

The frontend is connected to the smart contract functions through TypeScript bindings.

The guestbook uses passkeys for convenient user signup and login.

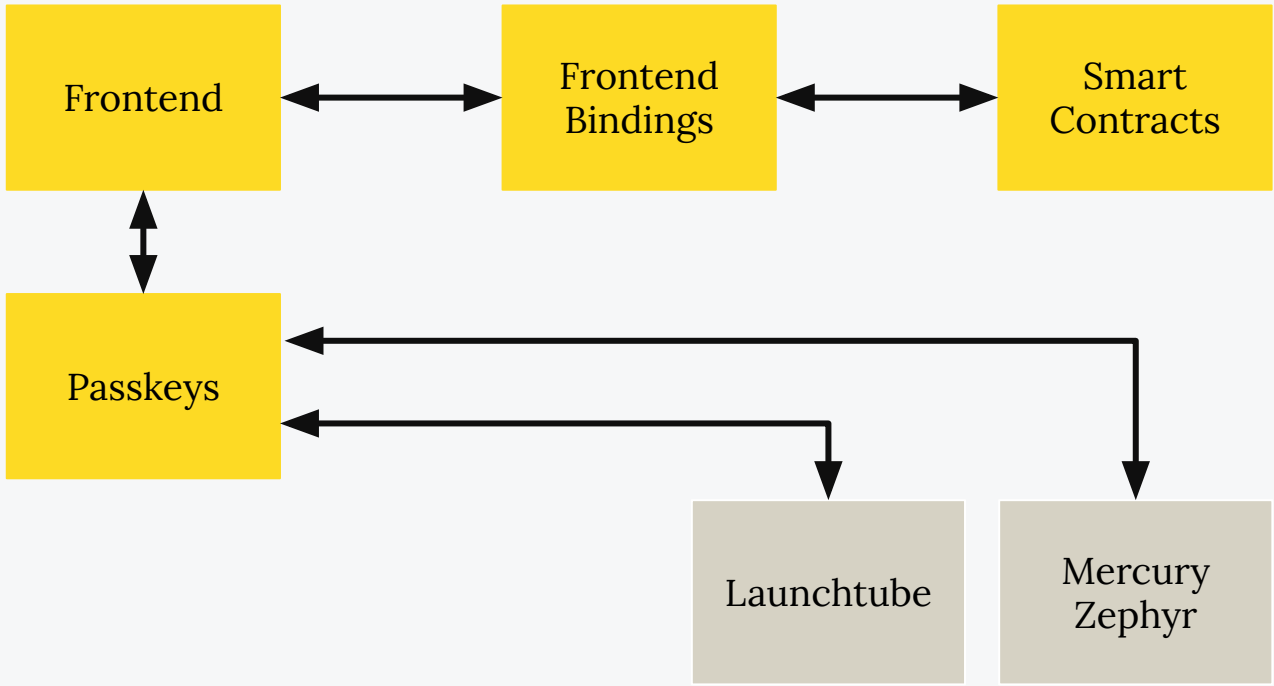


HIGH-LEVEL ARCHITECTURE

Passkeys are a convenient and way for users to sign up and log in, without using a browser wallet plugin. Passkeys require a couple of external services to operate:

Launchtube is similar to a "paymaster" service. The passkey-kit needs it, but we don’t interact with it directly.

Mercury Zephyr is a data indexer used by the passkey-kit.



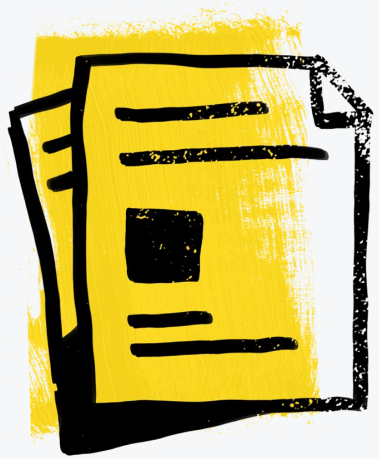
Section 03

SMART CONTRACT

SMART CONTRACT OVERVIEW

The guestbook smart contract consists of five functions:

- **write_message**: This function will allow a message to be written in the guestbook.
- **edit_message**: This function makes it possible for a user to edit an already existing message.
- **read_message**: Reading a message is as simple as querying the contract's persistent storage.
- **read_latest**: Call this function to show the latest message without knowing the message ID.
- **claim_donations**: This function allows users to make a donation.



SMART CONTRACT

`write_message`:

- **author**: The Address of the user
- **title**: The title of the guestbook message
- **text**: The guestbook message

The *write_message* function is used to add a new message to the guestbook.

First, the function checks if the title and text parameters are not empty, and then if the user is authorized to write a message.

An instance of the `Message` struct is created with the message data, including the author and current ledger sequence number, and stored with the utility function *save_message()*.

Source Code

```
    pub fn write_message(
        env: Env,
        author: Address,
        title: String,
        text: String,
    ) -> Result<u32, Error> {
        check_string_not_empty(&env, &title);
        check_string_not_empty(&env, &text);
        author.require_auth();

        let new_message = Message {
            author,
            ledger: env.ledger().sequence(),
            title,
            text,
        };

        let message_id = save_message(&env,
new_message);
        return Ok(message_id);
    }
```

SMART CONTRACT

`edit_message:`

- **message_id**: The ID of the message to edit
- **title**: The title of the guestbook message
- **text**: The guestbook message

The *edit_message* function is used to edit a message.

First, the function checks if either title or text parameters are not empty, gets the current message data, and then checks if the user is authorized to edit a message.

The existing message data is updated with the parameter data. If either title or text is empty, the existing value is used.

Finally, the message is stored.

Source Code

```
pub fn edit_message(
        env: Env,
        message_id: u32,
        title: String,
        text: String,
    ) -> Result<(), Error> {
        if title.is_empty() {
            check_string_not_empty(&env, &text);
        }

        if text.is_empty() {
            check_string_not_empty(&env, &title);
        }

        let mut message = get_message(&env, message_id);
        message.author.require_auth();

        let edited_title = if title.is_empty() {
            message.title
        } else {
            title
        };
        let edited_text = if text.is_empty() { message.text
} else { text };

        message.title = edited_title;
        message.text = edited_text;
        message.ledger = env.ledger().sequence();

        env.storage()
            .persistent()
            .set(&DataKey::Message(message_id), &message);
        return Ok(());
    }
```

SMART CONTRACT

`read_message:`

- **message_id**: The ID of the message to read

The *read_message* function is used to get an existing message from storage.

This function is simple since anyone can read guestbook messages without being authenticated first. The message is read with the utility function *get_message()*.

```
    pub fn read_message(env: Env, message_id: u32)
-> Result<Message, Error> {
        let message = get_message(&env,
message_id);
        Ok(message)
    }
```

Source Code

SMART CONTRACT

`read_latest:`

The *read_latest* function is a convenient way to get the last message added to the guestbook.

First, the function gets the message count to get the ID of the latest message. Messages are never deleted, so the message count will be equal to the ID of the last message.

The latest message is read with the utility function *get_message()* that takes the message ID as an argument.

Like the *read_message()* function, *read_latest()* doesn't require the caller to be authenticated.

[Source Code]

```
    pub fn read_latest(env: Env) ->
Result<Message, Error> {
        let latest_id = env
            .storage()
            .instance()
            .get(&DataKey::MessageCount)
            .unwrap();
        let latest_message = get_message(&env,
latest_id);
        Ok(latest_message)
    }
```

SMART CONTRACT

`claim_donations:`

- **token**: The Address of the token from where the donations should be claimed

Most of the donation functionality is handled by the frontend; the smart contract function just claims a donation.

First, the function creates a token client, checks the contract's balance of that token, and if there's a positive balance, then we send the tokens to the admin address.

This workshop doesn't go into details about the donation feature; see the documentation for more details.

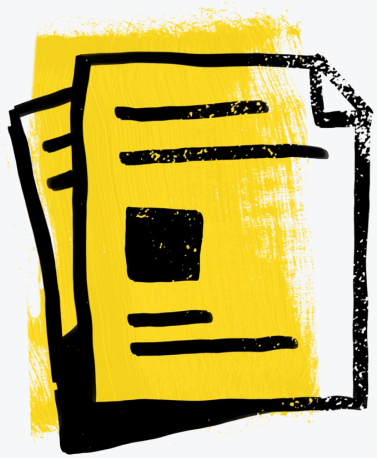
Source Code

```
    pub fn claim_donations(env: Env, token:
Address) -> Result<i128, Error> {
        let token_client =
token::TokenClient::new(&env, &token);
        let contract_balance =
token_client.balance(&env.current_contract_add
ress());

        if contract_balance == 0 {
            panic_with_error!(&env,
Error::NoDonations);
        }

        let admin_address: Address =
env.storage().instance().get(&DataKey::Admin).
unwrap();
        token_client.transfer(
            &env.current_contract_address(),
            &admin_address,
            &contract_balance,
        );

        Ok(contract_balance)
    }
```

SMART CONTRACT UTILITY FUNCTIONS

The guestbook smart contract functions use a few utility functions that are used in the previously mentioned contract functions:

- **check_string_not_empty**: This function is used to check if a string, provided by a user (e.g., title), is empty or not.
- **get_message**: This function reads a guestbook message from storage, based on the message ID.
- **save_message**: This function writes a new guestbook message to storage and returns the message ID of the new message.



SMART CONTRACT UTILITY

`check_string_not_empty`:

- **sus_string**: The string to check

The *check_string_not_empty()* function is used to check if a string is empty or not.

If the string is empty, an error is thrown.

```
fn check_string_not_empty(env: &Env, sus_string:
&String) {
    if sus_string.is_empty() {
        panic_with_error!(env,
Error::InvalidMessage);
    }
}
```

Source Code

SMART CONTRACT UTILITY

`get_message:`

- **message_id**: The ID of the message to read

The *get_message()* function is used to read a message from storage and it contains error handling.

First, the function checks if the message, with the ID provided as a parameter, exists. If it doesn't, an error is thrown.

If the message does exist, the message is read from storage and returned to the calling function.

```
fn get_message(env: &Env, message_id: u32) ->
Message {
    if !env
        .storage()
        .persistent()
        .has(&DataKey::Message(message_id))
    {
        panic_with_error!(env,
Error::NoSuchMessage);
    }

    let message: Message = env
        .storage()
        .persistent()
        .get(&DataKey::Message(message_id))
        .unwrap();
    return message;
}
```

Source Code

SMART CONTRACT UTILITY

`save_message:`

- **message**: The message to save

The *save_message()* function is used to write a new message to storage. Note that this function is only used to store a new message, and not to update an existing message.

First, the function initializes a variable with the current message count and increments the value. Note that the message counter is stored separately from the message storage.

Next, the message is written to storage using the incremented message count as the key and the message from the function parameter as the value. Finally, the incremented message counter is stored and returned.

Source Code

```
fn save_message(env: &Env, message: Message) ->
u32 {
    let mut num_messages = env
        .storage()
        .instance()
        .get(&DataKey::MessageCount)
        .unwrap_or(0 as u32);
    num_messages += 1;

    env.storage()
        .persistent()
        .set(&DataKey::Message(num_messages),
&message);
    env.storage()
        .instance()
        .set(&DataKey::MessageCount,
&num_messages);

    return num_messages;
}
```

Section 04

FRONTEND BINDINGS

FRONTEND BINDINGS

We need a way to invoke the smart contract functions from the web-based frontend. This is where TypeScript bindings come into play.

The Stellar CLI can generate and produce a fully typed NPM package, ready to integrate into your frontend. Add the bindings package to invoke a smart contract and use its interface.

Let’s take a look at how we set up the bindings for the guestbook frontend.

FRONTEND BINDINGS

Prerequisites

- Build smart contract
- Deploy smart contract

Before we can create the bindings, we need to build and deploy the smart contract.

When the smart contract is deployed, a few arguments are mandatory to initialize the guestbook (*admin*, *title,* and *text*).

```
%  stellar contract build
```

```
% stellar contract deploy \
  --wasm target/wasm32-unknown-unknown/release/ye_olde_
    guestbook.wasm \
  --source peter \
  --network testnet \
  -- \
  --admin peter \
  --title "Welcome to the guestbook" \
  --text "Join our guestbook and leave a message"
```

FRONTEND BINDINGS

Generate Bindings

Now that the contract has been built and deployed, we can use the Stellar CLI to generate the TypeScript bindings.

Step 1: Run the Stellar CLI bindings command, using the contract address. Use the contract address returned by the deploy command.

Step 2: The binding package is now generated in the packages/ye_olde_guestbook folder. Now we install and build it.

Step 3: The last step is to import the bindings package as a project dependency so we can use it in the frontend code.

```
% stellar contract bindings typescript \
  --network testnet \
  --id <contract_address_from_deploy_step> \
  --output-dir ./packages/ye_olde_guestbook \
  --overwrite
```

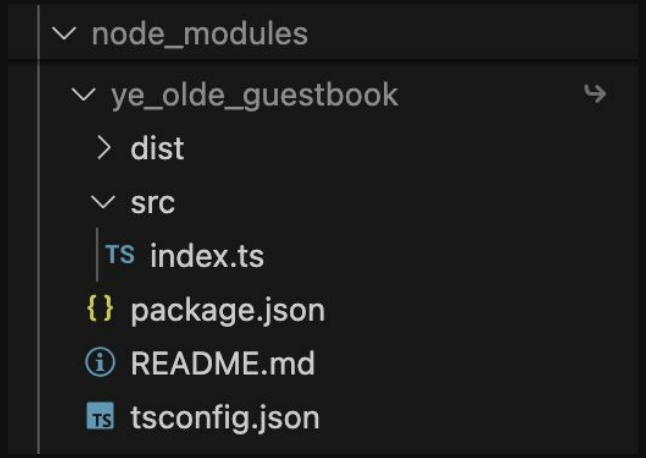
```
%  cd packages/ye_olde_guestbook
%  pnpm install
%  pnpm run build
%  cd ../..
```

```
%  pnpm add file:./packages/ye_olde_guestbook
```

FRONTEND BINDINGS

Evaluate Bindings

After building and installing the TypeScript contract bindings, we have a package in the frontend project's *node_modules* folder, and can use it as any other NPM package.



FRONTEND BINDINGS

Implementing Bindings

The example on the right is a simplified version of a contract function call in the guestbook, but it's a good example of how the bindings can be implemented.

First, the package is imported, and a guestbook client is instantiated with the appropriate network settings: in this case, testnet.

The *read_message()* contract function can now be invoked using the guestbook binding client.

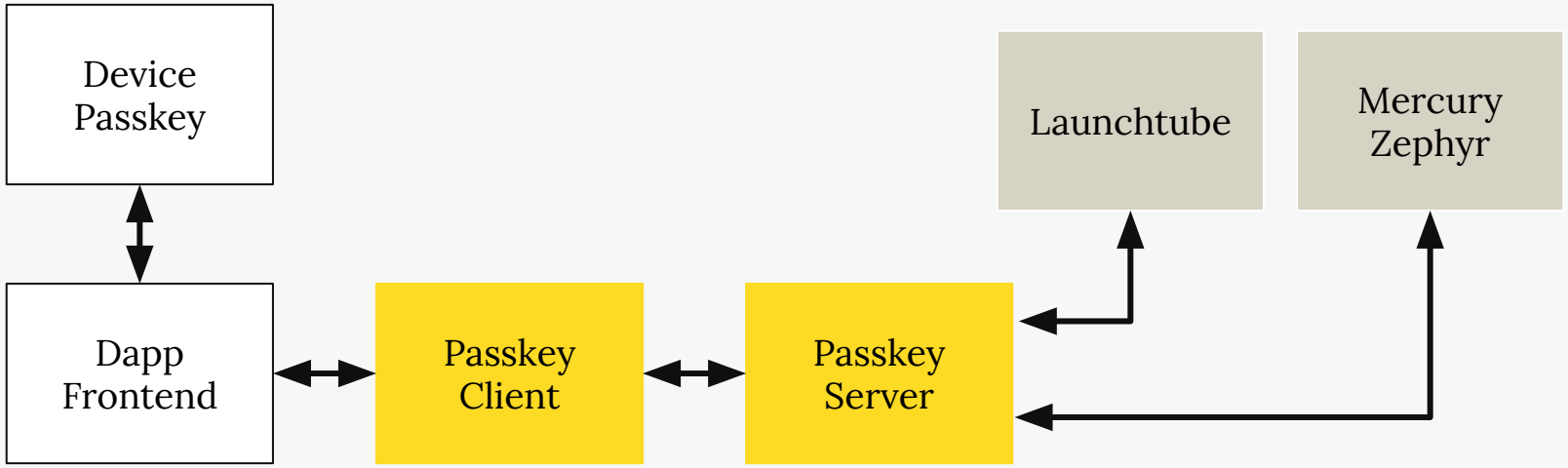
```
import * as Client from 'ye_olde_guestbook';

const guestbook = new Client.Client({
   ...Client.networks.testnet,
   rpcUrl: PUBLIC_STELLAR_RPC_URL,
});

let { result } = await guestbook.read_message({
   message_id: parseInt(params.id),
});
```

Section 05

PASSKEYS

PASSKEYS

How the passkey feature works



PASSKEYS

How it works

The passkey-kit relies on a couple of external services, so access to these services must be set up first.

Launchtube

Launchtube is similar to a "paymaster" service, if you're familiar with account abstraction in EVM networks. Launchtube is needed by the passkey-kit package and is used to submit passkey-signed transactions to the network.

Generate free testnet Launchtube JWT tokens with 100 XLM in credits: https://testnet.launchtube.xyz/gen

Mercury Zephyr

Zephyr is a data indexer service that is used for reverse lookup of smart wallet contract addresses based on passkey IDs, so a user's smart wallet address can be easily retrieved at login.

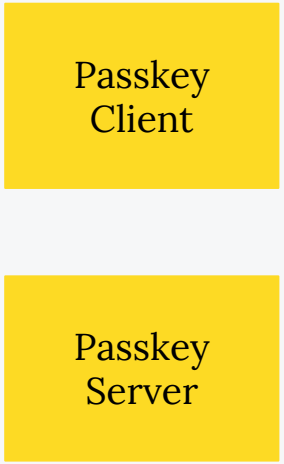
Mercury Zephyr's basic functionality can be used for free. Sign up here: https://www.mercurydata.app and see the documentation for setup instructions.

PASSKEYS

How it works

The guestbook passkey integration is split into two:

- A passkey client that is used by the Svelte frontend that deals with all user interactions.
- A passkey server that is used for secure communication with other services.



PASSKEYS - CLIENT INSTANCES

Instances

- **account**: an instance of the passkey-kit class, used for signing transactions, creating the wallet, and authenticating users
- **rpc**: for handling RPC interactions
- **sac**: an instance of passkey-kit's SAC client
- **native**: using the SAC instance for interacting with the native XLM asset contract

[Source Code]

```
import { Server } from '@stellar/stellar-sdk/rpc';
import { PasskeyKit, SACClient } from 'passkey-kit';
import type { Tx } from '@stellar/stellar-sdk/contract';


export const account = new PasskeyKit({
   rpcUrl: PUBLIC_STELLAR_RPC_URL,
   networkPassphrase: PUBLIC_STELLAR_NETWORK_PASSPHRASE,
   factoryContractId: PUBLIC_FACTORY_CONTRACT_ADDRESS,
});

export const rpc = new Server(PUBLIC_STELLAR_RPC_URL);

export const sac = new SACClient({
   rpcUrl: PUBLIC_STELLAR_RPC_URL,
   networkPassphrase: PUBLIC_STELLAR_NETWORK_PASSPHRASE,
});

export const native =
sac.getSACClient(PUBLIC_NATIVE_CONTRACT_ADDRESS);
```

PASSKEYS - CLIENT HELPERS

`send:`

- **tx**: Transaction

This function will send a transaction to the network via Launchtube using the server *send* endpoint.

```
export async function send(tx: Tx) {
    return fetch('/api/send', {
        method: 'POST',
        body: JSON.stringify({
            xdr: tx.toXDR(),
        }),
    }).then(async (res) => {
        if (res.ok) return res.json();
        else throw await res.text();
    });
}
```

Source Code

PASSKEYS - CLIENT HELPERS

`getContractId`:

- **signer**: Passkey ID

This function takes a passkey ID as a parameter and calls the server endpoint that will retrieve the contract address from Mercury Zephyr.

`fundContract`:

- **address**: User wallet address

This function is used to fund user wallets, since Friendbot can't fund C-addresses.

Source Code

```
export async function getContractId(signer:
string) {
   return
fetch(`/api/contract/${signer}`).then(async (res)
=> {
        if (res.ok) return res.text();
        else throw await res.text();
   });
}
```

```
export async function fundContract(address:
string) {
   return
fetch(`/api/fund/${address}`).then(async (res) =>
{
        if (res.ok) return res.json();
        else throw await res.text();
   });
}
```

PASSKEYS - SERVER

The passkey server gives us an importable server instance that can be accessed and used in other server-side logic.

This server instance will be used for sending transactions (via Launchtube) and reverse-looking-up contract addresses from a known passkey ID (via Mercury).

The passkey server creates the endpoints mentioned in the client helper functions. See the documentation and code for a deeper understanding of the passkey server.

Also, Svelte gives us the added benefit of keeping the code in this directory safe, so sensitive credentials, tokens, etc. are not exposed.

```
import { PasskeyServer } from 'passkey-kit';


export const server = new PasskeyServer({
   rpcUrl: PUBLIC_STELLAR_RPC_URL,
   launchtubeUrl: PUBLIC_LAUNCHTUBE_URL,
   launchtubeJwt: PRIVATE_LAUNCHTUBE_JWT,
   mercuryProjectName:
'smart-wallets-next-dima',
   mercuryUrl: PUBLIC_MERCURY_URL,
   mercuryKey: PRIVATE_MERCURY_KEY,
});
```

Source Code

Section 06

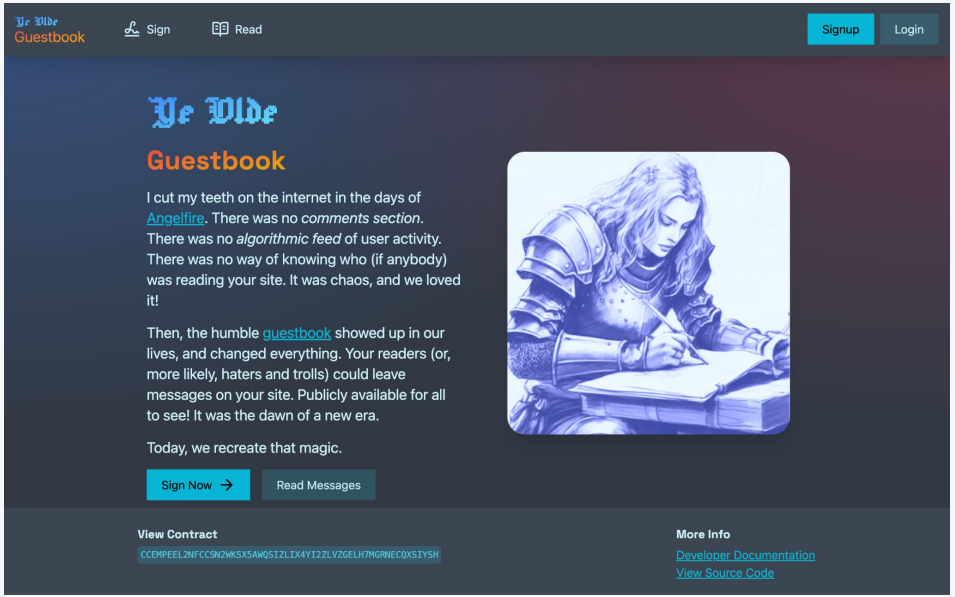
DAPP FRONTEND

DAPP FRONTEND

High-level description of the frontend:

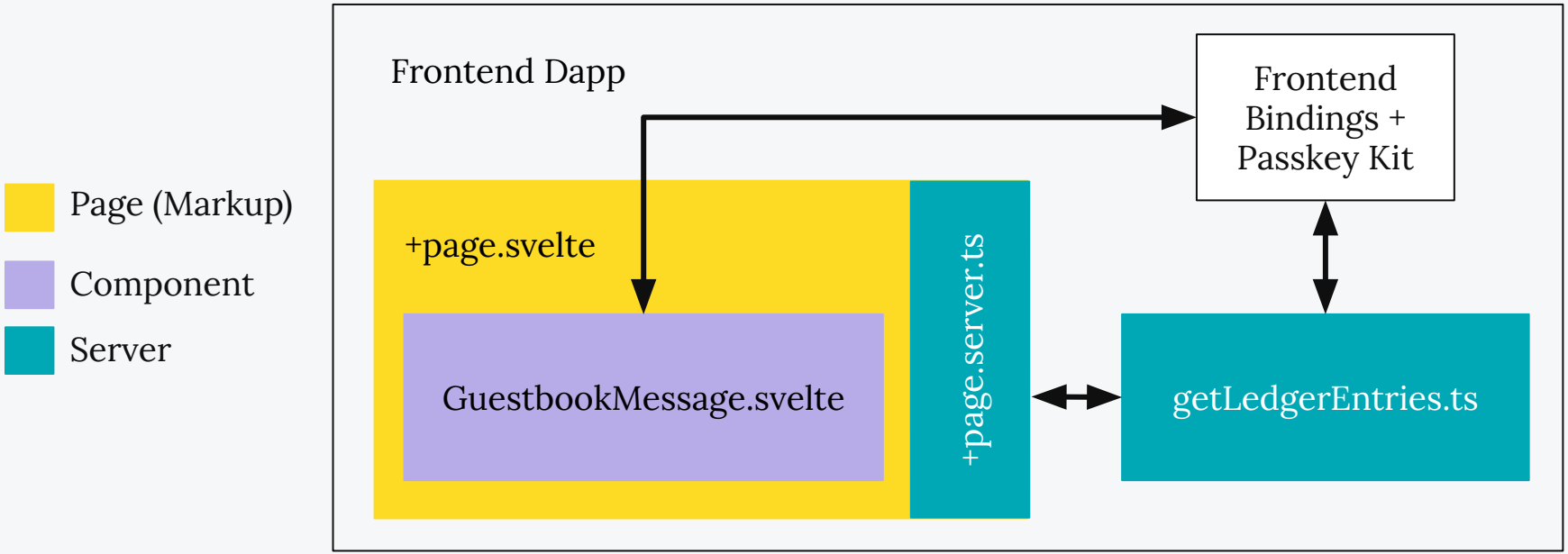
- The frontend is based on the Svelte framework
- The smart contract functions are invoked through the generated frontend bindings
- The passkey-kit is imported and provides the passkey authentication functionality

Most of the frontend is not different from a Web2 frontend, so let's dive into some of the Stellar integrations.



DAPP FRONTEND - READ MESSAGES

How it works



DAPP FRONTEND - READ MSG - GET LEDGER ENTRIES

`getWelcomeMessage:`

The welcome message is the first message stored by the admin when the contract was deployed from the ledger. The function uses RPC to get the message with `getLedgerEntries`.

`getAllMessages:`

This function will get all messages after the first message. It uses a function called `getMessageCount` to get the total number of messages stored, and then iterates over the message count to populate an array of ledger keys representing the messages. Finally, the messages are retrieved from the ledger, using `getLedgerEntries`.

Source Code

```
import { rpc } from '$lib/passkeyClient';
import { Address, networks, Contract, type Message, xdr,
scValToNative } from 'ye_olde_guestbook';

export async function getWelcomeMessage(): Promise<Message> {
   const result = await
rpc.getLedgerEntries(buildMessageLedgerKey(1));
   return
scValToNative(result.entries[0].val.contractData().val());
}

export async function getAllMessages(): Promise<Message[]> {
   const totalCount = await getMessageCount();
   const ledgerKeysArray = [];
   for (let messageId = 2; messageId <= totalCount;
messageId++) {
       ledgerKeysArray.push(buildMessageLedgerKey(messageId));
   }

   const result = await
rpc.getLedgerEntries(...ledgerKeysArray);
   const messages = result.entries.map((message) => {
       return {
           ...scValToNative(message.val.contractData().val()),
       };
   });

   return messages;
}
```

DAPP FRONTEND - READ MSG - +PAGE.SERVER.TS

`+page.server.ts:`

The messages are loaded when the page is loaded, and this is handled by the page's server script.

The message data will be available for use in +page.svelte.

```
import { getAllMessages, getWelcomeMessage } from
'$lib/server/getLedgerEntries' ;
import type { PageServerLoad } from './$types';

export const load: PageServerLoad = async () => {
    return {
        welcomeMessage: await getWelcomeMessage(),
        messages: await getAllMessages(),
    };
};
```

Source Code

DAPP FRONTEND - READ MSG - +PAGE.SVELTE

`+page.svelte`:

- **data**: Message data

This page renders the /read page, which shows the welcome message and the complete list of messages. The code on the right is reduced; see the source code for the full code.

The GuestbookMessage component is used twice, first for showing the welcome message (messageId = 1) and second for showing the complete list of messages (messageId > 1). The message data is loaded by +page.server.ts when the page is loaded.

Source Code

```
<script lang="ts">
    import GuestbookMessage from
'$lib/components/GuestbookMessage.svelte';

    import type { PageData } from './$types';
    export let data: PageData;

    let messages = data.messages;
</script>

<div class="flex flex-col md:flex-row justify-start
md:justify-between space-y-4">
      …
</div>

<GuestbookMessage message={data.welcomeMessage}
messageId={1} />
<hr class="!border-t-2" />

{#each messages as message, i (message.ledger)}
    <GuestbookMessage {message} messageId={i + 2}
/>
{/each}
```

DAPP FRONTEND - READ MSG - COMPONENT

`GuestbookMessage.svelte`:

- **props**: message, messageId

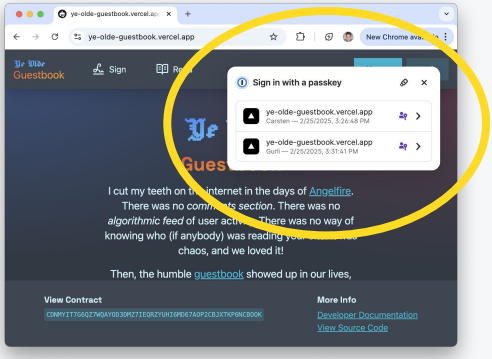
This component renders a guestbook entry, and takes *message* and *messageId* as props.

The code is reduced down to just the script and HTML to show the guestbook entry, the complete component includes both controls and functions to edit the message (if the user is the author of the message).

Source Code

```
<script lang="ts">
	import type { Message } from 'ye_olde_guestbook';

	export let message: Message;
	export let messageId: number;

	let messageTitle = message.title;
	let messageText = message.text;

	const cancelEdit = () => { … }
	const submitEdit = async () => { … }
</script>

<section class="card w-full variant-soft-primary">
	<div class="p-4 space-y-4">
		…
		<div class="flex gap-8 justify-between">
			<div class="space-y-2 grow">
				<h3 class="h3">
					{messageTitle}
				</h3>
				<article>
					<p>{messageText}</p>
				</article>
			</div>
		</div>
		…
	</div>
</section>
```

DAPP FRONTEND - PASSKEY



The Passkey Kit is handling the passkey prompt, so we don't need to do anything besides calling the Passkey Kit.

The code here shows how the Passkey Kit is implemented for the login function. By calling connectWallet() the passkey prompt will automatically appear.

[Source Code]

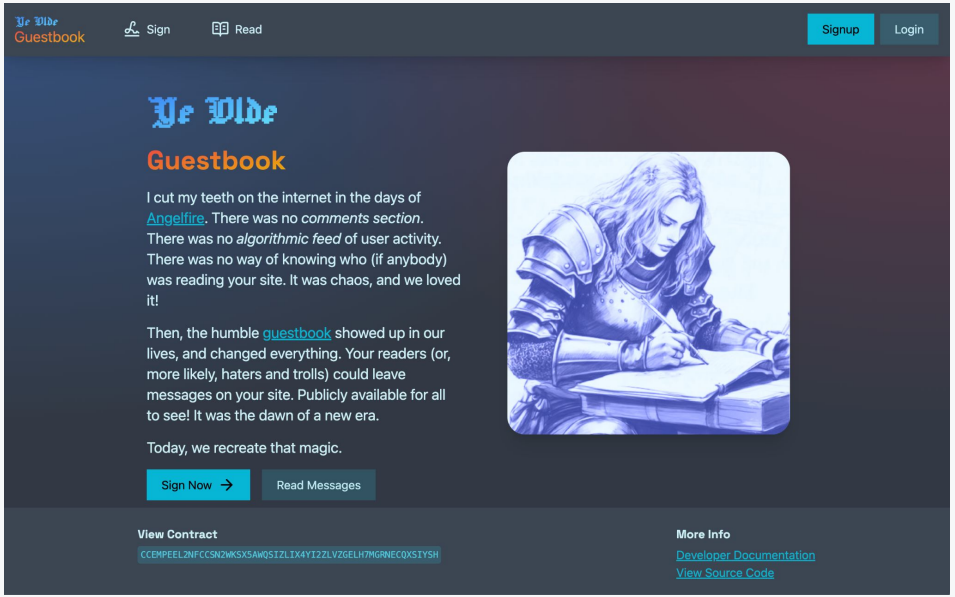
```
async function login() {
    console.log('logging in');
    try {
        const { keyId_base64, contractId: cid } = await
account.connectWallet({
            getContractId,
        });

        keyId.set(keyId_base64);
        console.log($keyId);

        contractId.set(cid);
        console.log($contractId);
    } catch (err) {
        console.log(err);
        toastStore.trigger({
            message: 'Something went wrong logging in. Please try
again later.',
            background: 'variant-filled-error',
        });
    }
}
```

Section 07

TRY IT OUT!

TRY IT OUT!

Try the live version of the guestbook and take a look at the documentation and the source code. You can even deploy it locally to experiment with the project.



Live Demo: https://ye-olde-guestbook.vercel.app
Documentation: https://developers.stellar.org/docs/build/apps/guestbook
Source Code: https://github.com/ElliotFriend/ye-olde-guestbook

Stellar

THANK YOU FOR YOUR TIME!

stellar.org