```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.16;

contract Crowdfunding {
struct Transaction {
address Sender;
address Receiver;
uint256 AmountInEther;
string EventType;
}

struct Campaign {
address organizer;
string title;
string description;
uint256 target;
uint256 deadline;
uint256 amountCollected;
string image;
address[] donators;
uint256[] donations;
}

mapping(address => uint256) public Funders;
mapping(address => bool) public Is_Funder_Registered;
Campaign[] public campaigns;
Transaction[] public TransactionHistory;

address public FundOrganizer;
bool public FundsWithdrawn;
uint256 public FundTransactionCount;
uint256 public DonateTransactionCount;

event Funded(address indexed funder, uint256 amountInEther);
event OwnerWithdraw(uint256 amountInEther);
event FundBack(address indexed funder, uint256 amountInEther);
event CampaignCreated(uint256 campaignId);

constructor() {
FundOrganizer = msg.sender;
}
```

```
function registerAsFunder() public {
Is_Funder_Registered[msg.sender] = true;
}

modifier onlyRegisteredFunder() {
require(Is_Funder_Registered[msg.sender], "Not a registered funder");
_;
}

function fund() public payable onlyRegisteredFunder {
require(isFundEnabled(), "Funding is now disabled");
Funders[msg.sender] += msg.value;
emit Funded(msg.sender, msg.value);

FundTransactionCount++;

Transaction memory newTransaction = Transaction({
Sender: msg.sender,
Receiver: address(this),
AmountInEther: msg.value / 1 ether,
EventType: "Funded"
});
TransactionHistory.push(newTransaction);
}

function withdrawOwner() public {
require(msg.sender == FundOrganizer, "Not authorized");
require(isFundSuccess(), "Cannot withdraw");
uint256 amountToSend = address(this).balance;
(bool success, ) = payable(FundOrganizer).call{value: amountToSend}("");
require(success, "Unable to send");
FundsWithdrawn = true;
emit OwnerWithdraw(amountToSend);

FundTransactionCount++;

Transaction memory newTransaction = Transaction({
Sender: FundOrganizer,
Receiver: FundOrganizer,
AmountInEther: amountToSend / 1 ether,
EventType: "OwnerWithdraw"
});
TransactionHistory.push(newTransaction);
}

function isFundEnabled() public view returns (bool) {
if (block.timestamp > campaigns[0].deadline || FundsWithdrawn) {
return false;
} else {
return true;
}
}

function getCampaignCount() public view returns (uint256) {
return campaigns.length;
}

function createCampaign(
string memory _title,
string memory _description,
uint256 _target,
uint256 _durationInSeconds,
string memory _image
) public returns (uint256) {
uint256 _deadline = block.timestamp + _durationInSeconds;

require(_deadline > block.timestamp, "The deadline should be a date in the future.");

Campaign memory newCampaign;
newCampaign.organizer = msg.sender;
newCampaign.title = _title;
newCampaign.description = _description;
newCampaign.target = _target;
newCampaign.deadline = _deadline;
newCampaign.amountCollected = 0;
newCampaign.image = _image;

campaigns.push(newCampaign);
emit CampaignCreated(campaigns.length - 1);

return campaigns.length - 1;
```

```
}

function donateToCampaign(uint256 campaignId) public payable {
uint256 amount = msg.value;
Campaign storage campaign = campaigns[campaignId];
require(campaign.deadline > block.timestamp, "Campaign deadline has passed");
campaign.donators.push(msg.sender);
campaign.donations.push(amount);

Funders[address(this)] += amount; // Send the donation to the contract's address
emit Funded(msg.sender, amount);

DonateTransactionCount++;
}

function getDonators(uint256 campaignId) public view returns (address[] memory, uint256[] memory) {
Campaign storage campaign = campaigns[campaignId];
return (campaign.donators, campaign.donations);
}

function isFundSuccess() internal view returns (bool) {
if (Funders[address(this)] >= campaigns[0].target || FundsWithdrawn) {
return true;
} else {
return false;
}
}
function getDonationHistory(uint256 campaignId, uint256 donationIndex) public view returns (address, uint256,
address) {
require(campaignId < campaigns.length, "Campaign ID out of range");
require(donationIndex < campaigns[campaignId].donations.length, "Donation index out of range");

Campaign storage campaign = campaigns[campaignId];

return (
campaign.donators[donationIndex],
campaign.donations[donationIndex] / 1 ether, // Convert to ether
campaign.organizer
);
}
}
```

1. Inside Contract

.This is a Solidity smart contract for a crowdfunding platform. The contract allows users to create campaigns, donate to campaigns, and withdraw funds as an organizer. Here's an overview of the key features and functions of the contract:

State Variables:

Funders: A mapping that keeps track of the amount of Ether each funder has contributed.

Is_Funder_Registered: A mapping that keeps track of whether an address is a registered funder.

campaigns: An array of Campaign structs, where each struct represents a crowdfunding campaign.

TransactionHistory: An array of Transaction structs, storing transaction history.

FundOrganizer: The address of the crowdfunding organizer.

FundsWithdrawn: A boolean indicating whether funds have been withdrawn by the organizer.

FundTransactionCount: Count of fund-related transactions.

DonateTransactionCount: Count of donation-related transactions.

Events:

Funded: Fired when a funder contributes Ether to a campaign.

OwnerWithdraw: Fired when the organizer withdraws funds from a successful campaign.

FundBack: Fired when a funder is refunded due to campaign failure.

CampaignCreated: Fired when a new campaign is created.

Modifiers:

onlyRegisteredFunder: A modifier that restricts certain functions to registered funders.

Constructor:

Sets the FundOrganizer to the address that deploys the contract.

Public Functions:

registerAsFunder: Allows an address to register as a funder.

fund: Allows registered funders to contribute Ether to a campaign.

withdrawOwner: Allows the organizer to withdraw funds from a successful campaign.

isFundEnabled: Checks if funding is currently enabled for the campaign.

getCampaignCount: Retrieves the total number of campaigns.

createCampaign: Allows users to create a new campaign.

donateToCampaign: Allows users to donate to a specific campaign.

getDonators: Retrieves the list of donators and their donations for a campaign.

getDonationHistory: Retrieves donation history for a specific campaign and donation index.

Internal Function:

isFundSuccess: Checks if a campaign has met its funding target.

The contract is designed for crowdfunding campaigns where registered funders can contribute Ether, campaigns have a funding target and deadline, and the organizer can withdraw funds if the campaign is successful. Additionally, there's support for refunding funds to contributors if the campaign fails.

Please note that this contract has some limitations and potential security concerns that should be addressed before deployment, such as handling multiple campaigns, managing campaign states, and ensuring secure fund transfers. Security audits and testing are essential steps when deploying a smart contract to the Ethereum blockchain.

2. Working of the contract

The contract you provided is designed to work as a crowdfunding platform on the Ethereum blockchain. Here's how it works:

Initialization:

The contract is deployed to the Ethereum blockchain, and the address deploying the contract becomes the FundOrganizer.

Registration as Funder:

Any Ethereum address can call the registerAsFunder function to register themselves as a funder.

Campaign Creation:

Users (including the organizer) can create new campaigns by calling the createCampaign function.

When creating a campaign, users provide details such as the campaign title, description, funding target (in Ether), duration (in seconds), and an image.

Donation:

After a campaign is created, users can donate Ether to the campaign using the donateToCampaign function.

Users specify the campaign ID to which they want to donate, and the specified amount of Ether is transferred to the contract and recorded in the campaign's donations array.

Campaign Status and Funding:

The isFundEnabled function checks if a campaign is still open for funding. It ensures that the campaign's deadline has not passed and that funds haven't been withdrawn.

If a campaign is still open, funders can continue to donate to it.

Campaign Success:

The isFundSuccess function checks whether a campaign has met its funding target or if funds have already been withdrawn by the organizer.

If a campaign has met its funding target, it is considered successful.

Withdrawal by Organizer:

The organizer can call the withdrawOwner function to withdraw funds from a successful campaign.

The entire balance of the contract is transferred to the organizer's address.

Refunding Funders:

If a campaign fails (doesn't meet its funding target), the organizer has the option to refund funders by calling the fundBack function.

Funds are refunded to the funders who contributed to the campaign.

Event Logging:

Throughout the contract, various events (e.g., Funded, OwnerWithdraw, FundBack, CampaignCreated) are emitted.

These events allow external applications to listen for updates and provide transparency about contract actions.

Transaction History:

Transaction history is recorded in the TransactionHistory array, including details about who sent funds, how much, and for what purpose (e.g., funding or withdrawal).

It's important to note that this contract allows for the creation of multiple campaigns, each with its own funding target and deadline. However, in its current form, the contract assumes that there is only one ongoing campaign at a time. Handling multiple campaigns simultaneously would require additional logic to manage and distinguish between them.

Before deploying a contract like this to the Ethereum blockchain, it's crucial to perform thorough testing and consider potential security vulnerabilities, such as reentrancy attacks, to ensure the safety of funds and users' assets. Additionally, contract updates or improvements may be necessary based on specific use cases and requirements.

3. Line by line explannation of code

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.16;
```

- Solidity pragma statement specifies version of the Solidity compiler to be used and sets a license identifier.

```
contract Crowdfunding {
```

- This line defines the start of a new Solidity smart contract named "Crowdfunding."

```
struct Transaction {
address Sender;
address Receiver;
uint256 AmountInEther;
string EventType;
}
```

- Here, a Transaction struct is defined to represent a transaction with fields like sender, receiver, amount in Ether, and event type.

```
struct Campaign {
address organizer;
string title;
string description;
uint256 target;
uint256 deadline;
uint256 amountCollected;
string image;
address[] donators;
uint256[] donations;
}
```

- This Campaign struct is defined to represent a crowdfunding campaign with fields like the organizer's address, title, description, funding target, deadline, amount collected, image, an array of donators, and an array of donations.

```
mapping(address => uint256) public Funders;
mapping(address => bool) public Is_Funder_Registered;
Campaign[] public campaigns;
Transaction[] public TransactionHistory;
```

- Several state variables are declared here:
- Funders: A mapping that tracks the total amount each funder has contributed.
- Is_Funder_Registered: A mapping that checks if an address is a registered funder.
- campaigns: An array of Campaign structs to store information about all the campaigns.
- TransactionHistory: An array of Transaction structs to store transaction history.

```
address public FundOrganizer;
bool public FundsWithdrawn;
uint256 public FundTransactionCount;
uint256 public DonateTransactionCount;
```

- More state variables are declared:
- FundOrganizer: The address of the crowdfunding organizer.
- FundsWithdrawn: A boolean to track whether funds have been withdrawn by the organizer.
- FundTransactionCount: A counter for fund-related transactions.
- DonateTransactionCount: A counter for donation-related transactions.

```
event Funded(address indexed funder, uint256 amountInEther);
event OwnerWithdraw(uint256 amountInEther);
event FundBack(address indexed funder, uint256 amountInEther);
event CampaignCreated(uint256 campaignId);
```

- Event declarations for emitting specific events:
- Funded: Fired when a funder contributes to a campaign.
- OwnerWithdraw: Fired when the organizer withdraws funds.
- FundBack: Fired when a funder is refunded.
- CampaignCreated: Fired when a new campaign is created.

```
constructor() {
FundOrganizer = msg.sender;
}
```

- The constructor function is executed once when the contract is deployed. It sets the FundOrganizer to the address of the account that deploys the contract.

```
function registerAsFunder() public {
```

```
Is_Funder_Registered[msg.sender] = true;
}
```

- registerAsFunder is a public function that allows any Ethereum address to register as a funder by setting Is_Funder_Registered to true for their address.

```
modifier onlyRegisteredFunder() {
require(Is_Funder_Registered[msg.sender], "Not a registered funder");
_;
}
```

- onlyRegisteredFunder is a modifier that restricts access to certain functions to only registered funders. If the sender is not a registered funder, the function will revert with the message "Not a registered funder."

```
function fund() public payable onlyRegisteredFunder {
require(isFundEnabled(), "Funding is now disabled");
Funders[msg.sender] += msg.value;
emit Funded(msg.sender, msg.value);
```

- The fund function allows registered funders to contribute Ether to a campaign.
- It checks whether funding is enabled, and if so, it records the funder's contribution in the Funders mapping and emits a Funded event.

```
FundTransactionCount++;

Transaction memory newTransaction = Transaction({
Sender: msg.sender,
Receiver: address(this),
AmountInEther: msg.value / 1 ether,
EventType: "Funded"
});
TransactionHistory.push(newTransaction);
}
```

- The function increments the FundTransactionCount to keep track of fund-related transactions.
- It also creates a Transaction struct to record the details of the fund transaction and pushes it into the TransactionHistory array.

```
function withdrawOwner() public {
require(msg.sender == FundOrganizer, "Not authorized");
require(isFundSuccess(), "Cannot withdraw");
uint256 amountToSend = address(this).balance;
(bool success, ) = payable(FundOrganizer).call{value: amountToSend}("");
require(success, "Unable to send");
FundsWithdrawn = true;
emit OwnerWithdraw(amountToSend);
```

- The withdrawOwner function allows the organizer to withdraw funds from a successful campaign.
- It checks if the sender is the organizer and if the campaign is successful before proceeding with the withdrawal.
- The entire balance of the contract is transferred to the organizer's address, and the OwnerWithdraw event is emitted.

```
FundTransactionCount++;

Transaction memory newTransaction = Transaction({
Sender: FundOrganizer,
Receiver: FundOrganizer,
AmountInEther: amountToSend / 1 ether,
EventType: "OwnerWithdraw"
});
TransactionHistory.push(newTransaction);
}
```

- Similar to the fund function, the withdrawOwner function records the withdrawal transaction in the TransactionHistory array.

```
function isFundEnabled() public view returns (bool) {
if (block.timestamp > campaigns[0].deadline || FundsWithdrawn) {
return false;
} else {
return true;
}
}
```

- The isFundEnabled function checks whether funding is currently enabled for the campaign.
- It returns true if the current block's timestamp is before the campaign's deadline and funds haven't been withdrawn; otherwise, it returns false.

```
function getCampaignCount() public view returns (uint256) {
return campaigns.length;
}
```

- The getCampaignCount function allows external callers to retrieve the total number of campaigns created.

```
function createCampaign(
string memory _title,
string memory _description,
uint256 _target,
uint256 _durationInSeconds,
string memory _image
) public returns (uint256) {
```

- The createCampaign function allows users to create a new campaign with specified parameters such as title, description, funding target, duration in seconds, and an image.
- It returns the index of the newly created campaign in the campaigns array.

```
uint256 _deadline = block.timestamp + _durationInSeconds;
require(_deadline > block.timestamp, "The deadline should be a date in the future.");
```

- The function calculates the campaign's deadline by adding the duration to the current block's timestamp and ensures that the deadline is set in the future.

```
Campaign memory newCampaign;
newCampaign.organizer = msg.sender;
newCampaign.title = _title;
newCampaign.description = _description;
newCampaign.target = _target;
newCampaign.deadline = _deadline;
newCampaign.amountCollected = 0;
newCampaign.image = _image;

campaigns.push(newCampaign);
emit CampaignCreated(campaigns.length - 1);

return campaigns.length - 1;
}
```

- The function creates a new Campaign struct with the provided details and initializes the amount collected to 0.
- The new campaign is added to the campaigns array, and the CampaignCreated event is emitted with the index of the newly created campaign.
- Finally, the function returns the index of the newly created campaign.

```
function donateToCampaign(uint256 campaignId) public payable {
uint256 amount = msg.value;
Campaign storage campaign = campaigns[campaignId];
require(campaign.deadline > block.timestamp, "Campaign deadline has passed");
campaign.donators.push(msg.sender);
campaign.donations.push(amount);
```

- The donateToCampaign function allows users to donate to a specific campaign.
- Users specify the campaign ID they want to donate to and send Ether along with the transaction.
- The function checks if the campaign's deadline has not passed.
- It records the donor's address and the donation amount in the campaign's donators and donations arrays.

```
Funders[address(this)] += amount; // Send the donation to the contract's address
emit Funded(msg.sender, amount);
```

```
DonateTransactionCount++;
}
```

- The function also updates the total amount donated to the contract's address in the Funders mapping.
- It emits a Funded event to notify that a donation has been made.
- The DonateTransactionCount is incremented to keep track of donation-related transactions.

```
function getDonators(uint256 campaignId) public view returns (address[] memory, uint256[] memory) {
Campaign storage campaign = campaigns[campaignId];
return (campaign.donators, campaign.donations);
}
```

- The getDonators function allows external callers to retrieve the list of donators and their respective donation amounts for a specific campaign.

```
function isFundSuccess() internal view returns (bool) {
if (Funders[address(this)] >= campaigns[0].target || FundsWithdrawn) {
return true;
} else {
return false;
}
}
```

- The isFundSuccess function checks if a campaign is considered successful.
- It returns true if either the total amount donated to the contract (address(this)) is greater than or equal to the campaign's target amount or if funds have already been withdrawn by the organizer.

```
function getDonationHistory(uint256 campaignId, uint256 donationIndex) public view returns (address, uint256,
address) {
require(campaignId < campaigns.length, "Campaign ID out of range");
require(donationIndex < campaigns[campaignId].donations.length, "Donation index out of range");

Campaign storage campaign = campaigns[campaignId];

return (
campaign.donators[donationIndex],
campaign.donations[donationIndex] / 1 ether, // Convert to ether
campaign.organizer
);
}
}
```

- The getDonationHistory function allows external callers to retrieve donation history for a specific campaign and donation index.
- It checks if the specified campaign ID and donation index are within valid ranges.

- It returns a tuple containing the donor's address, the donation amount in Ether (converted from Wei), and the organizer's address for the specified donation index in the specified campaign.