Kickstarter App Design Document

1. Introduction

The Kickstarter is a simple crowdfunding solution that lets projects raise capital from diverse sources. A project manager will start a campaign, regular users will add their contributions, and eventually the campaign will end. Depending on whether the target amount was reached, the money will be distributed or sent back to contributors.
This project is done as part of an interview process with a hard deadline, therefore, there will be some flexibility in the project scope - this will be pointed out below.

2. Design Overview

Let's talk about the "big picture" of the kickstarter system and the constraints we have due to blockchain being involved or due to the time allowed for implementation.

2.1. Bitcoin Blockchain Constraints

The problem mentions that this system should allow to raise funds via cryptocurrency, specifically, BTC. While my knowledge of the BTC blockchain is limited, I know at least some things that can complicate the seemingly simple nature of this project.

2.1.1. Nondeterministic Time for Transactions

Unlike transactions that are written directly into a database, it takes some nondeterministic time for a transaction on a blockchain to complete. According to Trust Machines own website, it takes at least 10 minutes for a block to be generated. Based on my own experience with blockchain, it is not unusual for a transaction to take up to several hours. So, typically, the contribution will not "happen" nearly instantaneously.

In this project (if time allows) we can approximate this by putting a contribution into the PENDING state to represent the transaction being submitted to the chain and executed there. After some random amount of time we can set the contribution to the SUCCESS or FAIL state to represent the corresponding transaction being successfully completed or not completed on the chain.

2.1.2. Transactions Have Fees

Unlike donating cash to Girl Scouts when buying cookies, the contributions in cryptocurrency require fees to be paid. (So we can say that a contribution consists of the "net contribution

amount" and "fees".) Furthermore, when the campaign goal is not reached, refunding funds to contributors via crypto will require more fees to be paid. Someone will have to pay those fees - the campaign owner or the contributors (or split between both parties).

I am not aware how this issue is handled in real life, this sounds like a business decision. For the purpose of this project, to keep things simple, we can pretend that the fees don't exist, so the amount contributed is the same amount that the campaign receives.

2.2. Time Constraints: Approximation vs "Real Thing"

Since this is a short-term project, there will be more constraints. Let's look at those.

2.2.1. Blockchain and Blockchain Listener Component

A kickstarter system that uses blockchain would be aware of the existence of the blockchain and feature a component (Kickstarter ChainListener) that would subscribe to the chain and get notifications for specific transaction hashes.

I know this is possible with the Ethereum blockchain, but I was not sure if the BTC chain allows this publish/subscribe mechanism.
Based on my online search it is possible:

- https://bitcoin.stackexchange.com/questions/108412/how-to-set-up-a-listener-on-an-address-and-trigger-events-upon-receiving-transa
- https://bitcoindev.network/accessing-bitcoins-zeromq-interface/

For this project I will not attempt to connect my system to an actual blockchain, so there will be no Kickstarter ChainListener.

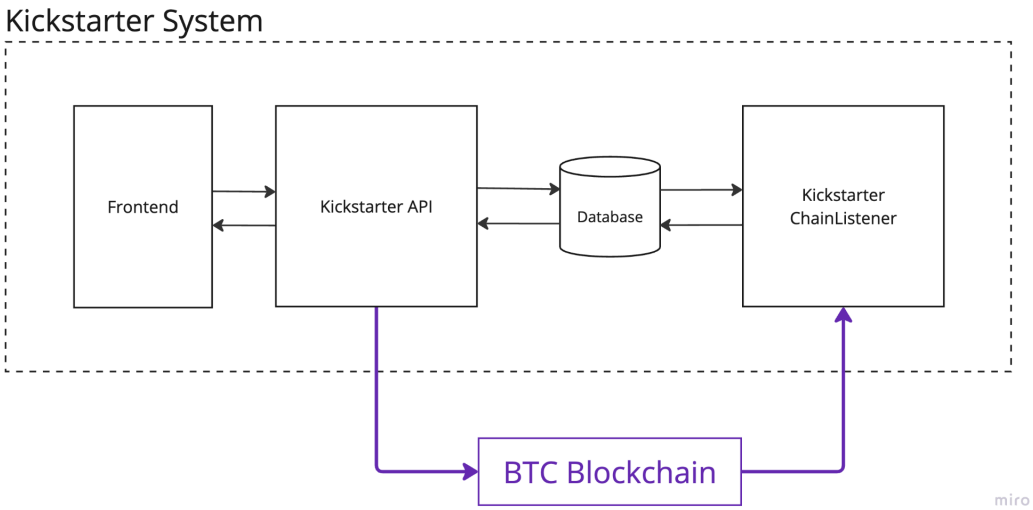
2.2.2. Postman Will Be The Frontend

In real life users would interact with a proper UI either on the web or in an app. Since I am more familiar with a backend, I will not attempt to write any fancy UI and will find my happiness interacting with Postman.

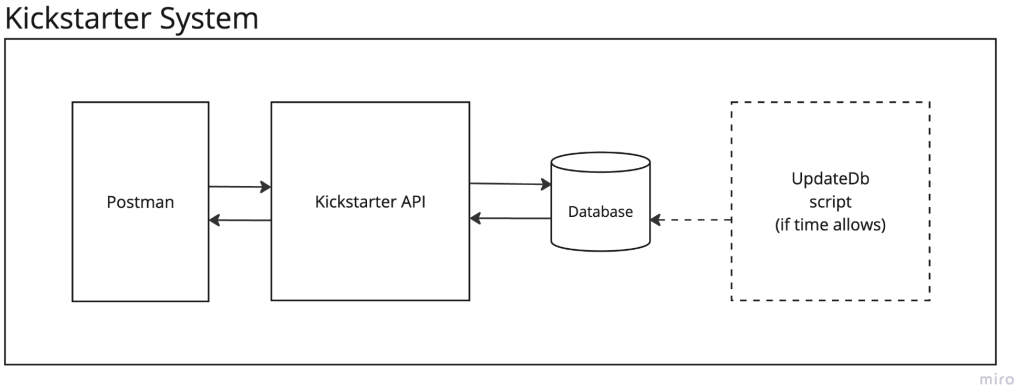
2.3. Summary In Two Pictures

The following two pictures show everything I said above.

1. A simplified picture of what a real system might look like.



2. What I will try to implement.



3. Detailed Design

Let's look at the system closer. There are at least three areas / entities that stand out: users (that can have roles of an Admin, Manager or Donor), campaign and donations.

Logically, we can even think of this as of **three possible services** that can exist, if we follow a **microservices approach**:
- user service,
- campaign service and
- (maybe?) donations service.

Due to the limited scope of this project, I am not going to build any microservices and will go with a simple monolith system. Still, I would like to point out that in real life the microservices would be my preferred approach.

3.1. Endpoints

Below is a table of possible endpoints that we can expose to the front end.

Notice that **`users` endpoints** are all grayed out - this is because **I don't plan on implementing them**. I mention them for the sake of logical completeness. In real life we normally allow users to create an account ("create a user"), edit it, etc. For this project I plan to manually enter some users with various roles into the database - some donors, a couple managers and perhaps an admin.

M = manager
D = donor

API Endpoints (what I plan to implement)				
Verb	URL	RequestBody	Response	User
POST	/campaigns/create	CreateCampaignRequest	None	M
GET	/campaigns/{campaign_id}	None	CampaignObject	D, M
PUT	/campaigns/{campaign_id}	UpdateCampaignRequest	None	M
DELETE	/campaigns/{campaign_id}	None	None	M
GET	/campaigns/{campaign_id}/donations	None	[] of DonationObjects	D, M
GET	/campaigns	None	[] of CampaignObjects	D, M

Not Part of This Project				
POST	/campaigns/{campaign_id}/vote	VoteRequest	None	D
POST	/donations/make	MakeDonationRequest	None	D
GET	/donations/{donation_id}	None	DonationObject	D, M
POST	/users/create	CreateUserRequest	None	D
GET	/users/{userid}	None	UserObject	D
PUT	/users/{userid}	UpdateUserRequest	None	D
DELETE	/users/{userid}	None	None	Admin
GET	/users	None	[] of UserObjects	Admin

All endpoints should return 4xx status codes in case of an error or a 2xx http status code in case of a success (200 for GET requests, 204 for the rest).

3.1.1.Request Objects

Examples of `CreateCampaignRequest` and `UpdateCampaignRequest` request objects are below

CreateCampaingRequest

```
CreateCampaignRequest:
{
  "name": "my campaign",
  "description": "money for my taco party",
  "deadline": "Noon of August 19, 2024, PST", // the timestamp
equivalent
  "target_amount": 12345,
  "installment_data": {}
}
```

Backend would also generate or compute these fields:

`"id": "campaign-7659",`
`"user_id": "user-5678",`
`"status": "created",`
	generated when campaign is created
	extracted from the session / header
	eventually "open" -> "funded" or "closed" -> "completed" or "canceled"

`"amount_raised": 0,`
 	
`"created_at": "Noon of August 11, 2024, PST",`
`"modified_at": "Noon of August 12, 2024, PST",`
	initially 0, incremented with each donation made
	
	the timestamp equivalent
	the timestamp equivalent

Update Campaign Request

```
UpdateCampaignRequest:
{
  "id": "campaign-7659",
  "name": "my campaign",
  "description": "money for my taco party",
  "deadline": "Noon of August 19, 2024, PST", // the timestamp
equivalent
  "target_amount": 54321,
  "installment_data":
  {
  "installments": [
    "installment": {
      "amount": 123,
      "deadline": timestamp_of_the_date,
      "is_locked": true,
      "voting_data": {} // no voting happened yet
```

```
    },
    "installment": {
      "amount": 321,
      "deadline": timestamp_of_the_date,
      "is_locked": true,
      "voting_data": {} // no voting happened yet
    }
  ]
  }

}
```

Backend would also generate or compute these fields:

`"user_id": "user-5678",` `"status": "open",`	extracted from the session / header eventually going to "funded" or "closed" then to "completed" or "canceled"

`"amount_raised": 0,`
 	
`"modified_at": "Noon of August 12, 2024, PST",`
	initially 0, incremented with each donation made
	
	the timestamp equivalent

3.2. Database Design

Once we start thinking about what a `CreateCampaignRequest` contains, for example, we have to think about what kind of information we plan to store in our database and how we organize it.
I propose these three tables: users, campaigns and donations. The primary key in each table would be `id` (unless stated otherwise).

3.2.1. Users

I plan to manually populate this table:

Field	Data Type
id name created_at	UUID String Timestamp Timestamp

modified_at	

`name`: keep it simple here - no first, last, etc. - as long as it is unique and at least 1 character long, we are ok

3.2.2. Campaigns

I plan to focus in my project on this table.

Field	Data Type
id name description user_id // index	UUID String String UUID
deadline target_amount amount_raised installment_data	Timestamp Integer Integer JSON String
status created_at modified_at	String Timestamp Timestamp

`user_id`: this is the id of the campaign manager - we will only allow 1 manager per campaign to keep things simple in this project. In real life we can have a joint table to associate multiple managers to campaigns.

Here is an example of what installment_data JSON will look like:

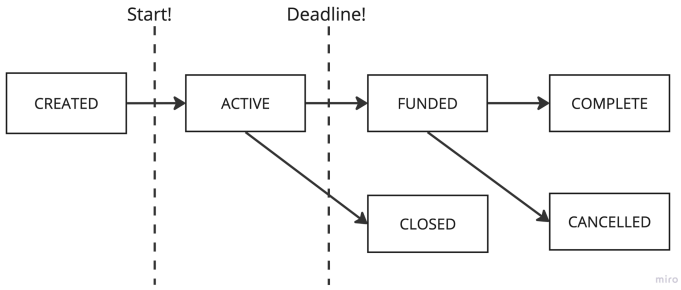
`installment_data`:

```
{
  "installments": [
    "installment": {
      "order": 1,
      "amount": 123,
      "deadline": timestamp_of_the_date,
      "is_unlocked": true,
      "voting_data": {
        "confident": 23,
        "not_confident": 4,
      }
    },
    "installment": {
      "order": 2,
```

```
      "amount": 321,
      "deadline": timestamp_of_the_date,
      "is_unlocked": false,
      "voting_data": {} // no voting happened yet
    }
  ]
}
```

In real life I might choose to keep the disbursement data in its own table, but to keep things simple and small, I will put it in a JSON String in this project.

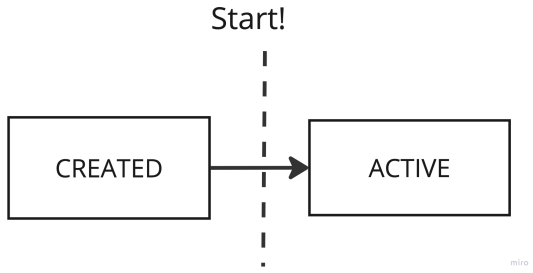
`status`: Each campaign will go through this lifecycle, so here is a possible status it may be in:



Once a campaign has been created, it is in CREATED status and can be edited or deleted until the manager decides that it looks good. At this point they call `/campaigns/{campaign_id}/start` endpoint and “locks” the campaign (so the campaign manager cannot change anything after donations started).

While the donation period lasts, the campaign is considered “ACTIVE”. After the deadline, if the target_amount was reached, the campaign will be FUNDED, it will stay in this status until completion. If at any point a vote of no confidence is passed, the campaign status becomes CANCELED and stays that way. At some point (after the project for which funds were raised is done), we can mark the campaign as COMPLETE.

For my project, I will only implement the first two states: CREATED and ACTIVE along with the `/campaigns/{campaign_id}/start` endpoint:

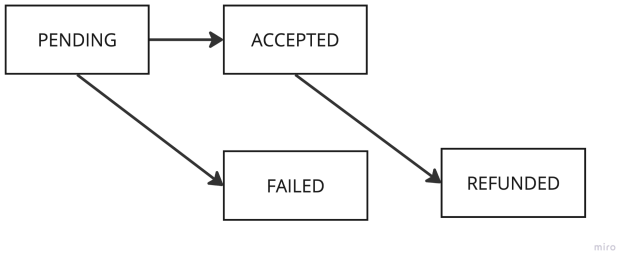


3.2.3. Donations

I plan to manually populate this table:

Field	Data Type
id amount user_id // index campaign_id // index	UUID Integer UUID UUID
transaction_hash // index, not part of this project created_at modified_at	UUID Timestamp Timestamp

`transaction_hash`: since there will be no `transactions` table in this project, this field can be removed.
`status` field: here is another tricky point. A typical donation tied to a real blockchain transaction might progress in this sort of fashion:



Ideally, I would like to implement these transitions but, after looking at the size of this document,

I am going to skip this complexity and assume **the status of a donation is always "ACCEPTED**". So, there will be no "status" field for a donation for now.

Summary

What I plan to have in my project:

- Frontend = Postman
- Backend = endpoints for CRUD operations on a campaign + display the donations attached to this campaign:

API Endpoints (what I plan to implement)

Verb	URL	RequestBody	Response	User
POST	/campaigns/create	CreateCampaignRequest	String	M
GET	/campaigns/{campaign_id}	None	None	D, M
PUT	/campaigns/{campaign_id}/start	None	CampaignObject	M
PUT	/campaigns/{campaign_id}	UpdateCampaignRequest	None	M
DELETE	/campaigns/{campaign_id}/delete	None	None	M
GET	/campaigns	None	[] of CampaignObjects	A
GET	/campaigns/active	None	[] of CampaignObjects	D, M

M = manager
D = donor

- Tables in DB: `users`, `campaign` and `donations`. Table users and donations will be populated manually.