# **EvenTrade** Decentralized Event Ticket Exchange

Marie Hargan, Lukas Zumwalt



## Work Done

Started with some research into React JS and its use for front-end development. Then, created the code skeleton and repository structure to support a single-page React application. After installation of Node JS and all related dependencies needed for this project, I was able to start working on actual front-end development for our application. I created a couple pages with a fully functional navigation bar that automatically adjusts based on window size. This means that when the window is smaller or being viewed on a smart device the navigation bar turns into a drop down menu. Furthermore, all the links on the navigation bar and drop down menu link to the proper pages.

I am currently working on getting login functionality working with secure user authentication. I have run into some roadblocks since the original plan was to use a third party authentication application (Auth0), which doesn't seem to mesh well with React JS. Between trying to get the third party application to work, I have attempted to write our own user authentication with React JS, but security is doubtful. For the next couple of weeks I plan on solving this issue and having a fully functional website that is ready for deployment.

- Marie Hargan

Exploration into the technical deployment of smart contracts generated many questions of unique ticket authenticity and protection; which exposed a flaw in our design, where the valuable information - the ticket's QR code, for example - would be public information by the default nature of a blockchain. In order to protect a user's information, and conform it to a decentralized application, it must be minted to a Non-Fungible Token or NFT. The minting of these NFTs, and the association of them with smart contracts defining the event characteristics and allowing NFTs to be a distributed aggregate, is ***not an easy task***. In fact, it is entirely the wild west out here.

GET Protocol is an organization providing a valuable abstraction of NFT minting and smart contract deployment of exchangeable event media on Ethereum and Polygon blockchains, and it is what we will be integrating into our project. Currently, a suite of commands are prepared and are utilizing a **builder pattern** in order to generate API calls to the GET Ticket Engine API. There is a distinction between event smart contracts and individual ticket NFTs, as many NFTs may associate to one unique smart contract, so they are delegated to separate processors with associated, dynamic instances.

- Lukas Zumwalt

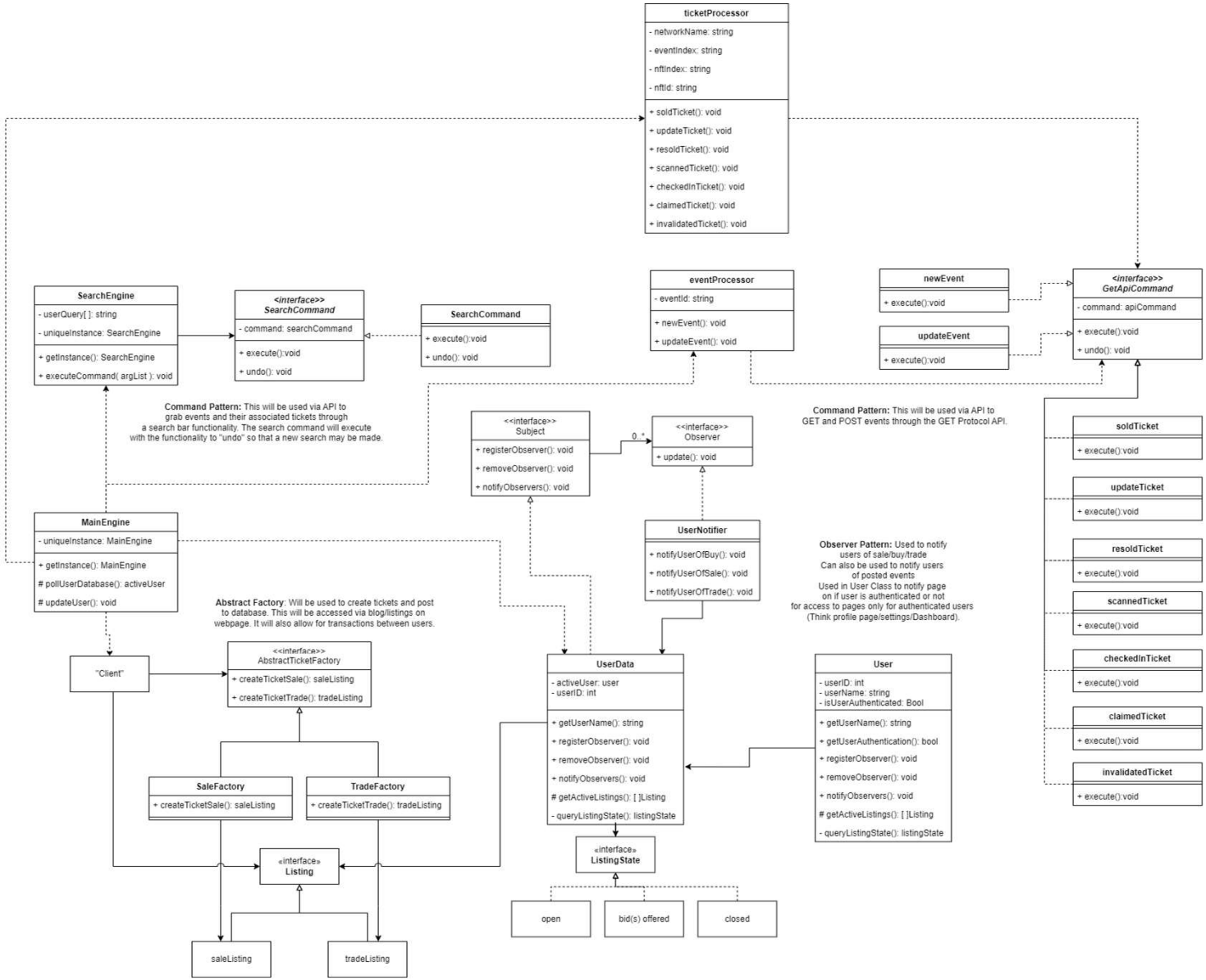
## Changes and/or Issues Encountered

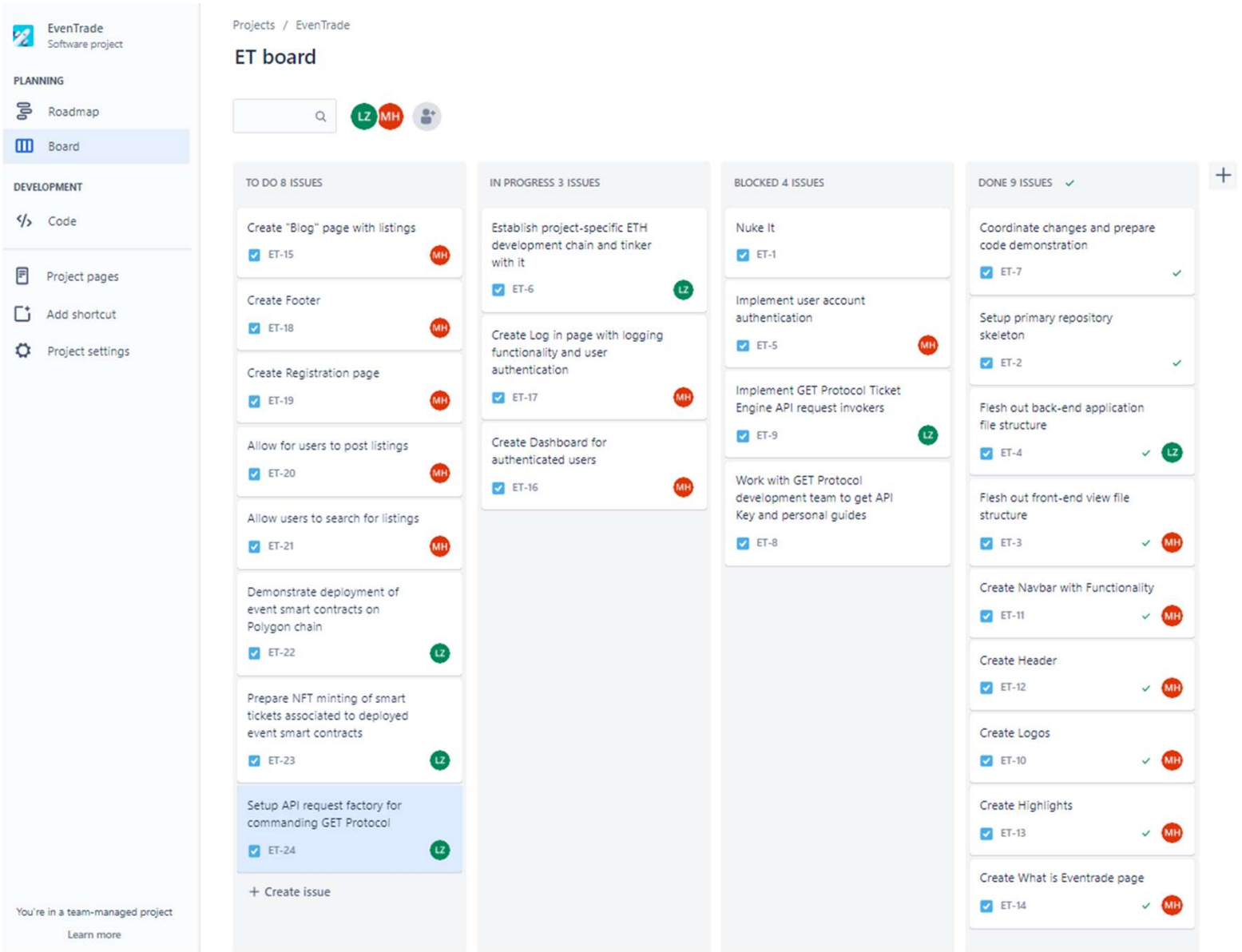
- User authentication Login page stasis
- User-authenticated routes with access to dashboard
    - Account settings page would be a stub
    - User-access would be an access characteristic, though
- Ticket media *CAN'T* be loaded directly onto blockchain smart contracts
    - Must be minted as an NFT *first*
    - May then be associated with an event's smart contract
- Extremely cumbersome to deploy robust smart contracts and mint NFTs manually
    - GET Protocol offers a tangible API solution for this
    - Currently getting in touch with their development team to request an API Key
        - Staging commands to invoke API calls
- MVC seems more powerful than initially treated
    - Considering we will be using an API heavily, will design more of our front/backend tethering with this composite of patterns in mind

## Patterns

Describe use of design patterns *so far* in the prototype and how they are helping the design

I (Lukas) haven't worked heavily with APIs before, so I wasn't sure how to implement them and their calls well into novel code. It became clear that a **command pattern** would benefit our design well, as it allows the API calls to exist in their own domain and associated code can recognize when they are invoked and react accordingly. Also, in order to manifest these API calls, a **builder pattern** is implemented in order to dynamically generate a configuration for an API request based on user input. This allows for each API command to control their own scope


ticketProcessor
- networkName: string
- eventIndex: string
- nftIndex: string
- nftId: string
+ soldTicket(): void
+ updateTicket(): void
+ resoldTicket(): void
+ scannedTicket(): void
+ checkedInTicket(): void
+ claimedTicket(): void
+ invalidatedTicket(): void
SearchEngine
- userQuery[ ]: string
- uniqueInstance: SearchEngine
+ getInstance(): SearchEngine
+ executeCommand( argList ): void
<interface>>
SearchCommand
- command: searchCommand
+ execute():void
+ undo(): void
SearchCommand
+ execute():void
+ undo(): void
eventProcessor
- eventId: string
+ newEvent(): void
+ updateEvent(): void
newEvent
+ execute():void
updateEvent
+ execute():void
<interface>>
GetApiCommand
- command: apiCommand
+ execute():void
+ undo(): void
Command Pattern: This will be used via API to
grab events and their associated tickets through
a search bar functionality. The search command will execute
with the functionality to "undo" so that a new search may be made.
Command Pattern: This will be used via API to
GET and POST events through the GET Protocol API.
<<interface>>
Subject
+ registerObserver(): void
+ removeObserver(): void
+ notifyObservers(): void
0..*
<<interface>>
Observer
+ update(): void
soldTicket
+ execute():void
updateTicket
+ execute():void
resoldTicket
+ execute():void
scannedTicket
+ execute():void
checkedInTicket
+ execute():void
claimedTicket
+ execute():void
invalidatedTicket
+ execute():void
MainEngine
- uniqueInstance: MainEngine
+ getInstance(): MainEngine
# pollUserDatabase(): activeUser
# updateUser(): void
UserNotifier
+ notifyUserOfBuy(): void
+ notifyUserOfSale(): void
+ notifyUserOfTrade(): void
Observer Pattern: Used to notify
users of sale/buy/trade
Can also be used to notify users
of posted events
Used in User Class to notify page
on if user is authenticated or not
for access to pages only for authenticated users
(Think profile page/settings/Dashboard).
Abstract Factory: Will be used to create tickets and post
to database. This will be accessed via blog/listings on
webpage. It will also allow for transactions between users.
"Client"
<<interface>>
AbstractTicketFactory
+ createTicketSale(): saleListing
+ createTicketTrade(): tradeListing
UserData
- activeUser: user
- userID: int
+ getUserName(): string
+ registerObserver(): void
+ removeObserver(): void
+ notifyObservers(): void
# getActiveListings(): [ ]Listing
- queryListingState(): listingState
User
- userID: int
- userName: string
- isUserAuthenticated: Bool
+ getUserName(): string
+ getUserAuthentication(): bool
+ registerObserver(): void
+ removeObserver(): void
+ notifyObservers(): void
# getActiveListings(): [ ]Listing
- queryListingState(): listingState
SaleFactory
+ createTicketSale(): saleListing
TradeFactory
+ createTicketTrade(): tradeListing
«interface»
Listing
«interface»
ListingState
open
bid(s) offered
closed
saleListing
tradeListing


EvenTrade
Software project
PLANNING
Roadmap
Board
DEVELOPMENT
Code
Project pages
Add shortcut
Project settings
You're in a team-managed project
Learn more
Projects / EvenTrade
ET board
LZ
MH
TO DO 8 ISSUES
Create "Blog" page with listings
ET-15
Create Footer
ET-18
Create Registration page
ET-19
Allow for users to post listings
ET-20
Allow users to search for listings
ET-21
Demonstrate deployment of event smart contracts on Polygon chain
ET-22
Prepare NFT minting of smart tickets associated to deployed event smart contracts
ET-23
Setup API request factory for commanding GET Protocol
ET-24
+ Create issue
IN PROGRESS 3 ISSUES
Establish project-specific ETH development chain and tinker with it
ET-6
Create Log in page with logging functionality and user authentication
ET-17
Create Dashboard for authenticated users
ET-16
BLOCKED 4 ISSUES
Nuke It
ET-1
Implement user account authentication
ET-5
Implement GET Protocol Ticket Engine API request invokers
ET-9
Work with GET Protocol development team to get API Key and personal guides
ET-8
DONE 9 ISSUES
Coordinate changes and prepare code demonstration
ET-7
Setup primary repository skeleton
ET-2
Flesh out back-end application file structure
ET-4
Flesh out front-end view file structure
ET-3
Create Navbar with Functionality
ET-11
Create Header
ET-12
Create Logos
ET-10
Create Highlights
ET-13
Create What is Eventrade page
ET-14